A Comparative Study of Conservative and Optimistic Trace-driven Simulations *

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

e-mail: {baer,xqin}@cs.washington.edu
phone: (206)-685-1376(baer), (206)-685-4087(xqin)

Abstract

In this paper, we consider the evaluation of the memory hierarchy of multiprocessor systems via parallel trace-driven simulation. We study two parallel simulation schemes: a conservative one using an algorithm proposed by Lin et al. [10], whose main characteristic is to insert the shared references from every trace in all other traces, and an optimistic one using a Time Warp-like [9] algorithm. We compare, qualitatively and quantitatively, the major causes of overhead and the overall performance of the two methods. In addition, we discuss the trade-offs in terms of implementation and debugging effort and of application to more general architectural simulation. The optimistic scheme is more complex but, in general, has slightly better performance, is more general, and does not require preprocessing.

*This work was supported in part by NSF Grants CCR-91-01541, 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. One would expect that the existing parallel systems could be exploited to predict the performance of their successors by taking advantage of their own ability to perform various tasks concurrently. In other words, parallel architectures should be simulated using parallel simulation. Although simulation of parallel architectures and parallel simulation sound similar, they designate two distinct entities. The former refers to the system that is the target of the simulation while the latter defines the medium on which the simulation is performed. The main reason why we would like to use parallel machines for simulation is that the 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 can provide us with higher computation and storage capabilities. Moreover, an additional motivation is that the functioning of the target system exhibits natural parallelism: instructions from distinct simulated processors may be issued and carried out independently and concurrently.

State of the art simulators of parallel architectures such as Proteus[2] and Tango[8] run on single processor workstations. Recently execution-driven parallel simulators for parallel architectures [13, 3] have been implemented on specific parallel architectures (the TMI CM-5 and BBN Butterfly respectively). The challenge in these parallel simulators is to provide effective means to simulate the communication among processors. To elaborate on this point, assume that one processor of the simulation system is used to simulate one processor of the target system. During the simulation, interprocessor communication will consist not only of the explicit communication between two nodes in the simulated system but also of many operations that involve parts of the target system such as the interconnection network or the cache coherence mechanism. Since the simulation is software-based, the slow-down due to the simulation of communication can erase, or even outweigh, the benefits of having simulation processes running in parallel. It is therefore critical to keep the amount and cost of communication as low as possible if we want to achieve good performance.

Parallel simulation methods can be broadly classified into two categories [7]: conservative methods and optimistic methods. A fundamental difference between the two approaches is that in a conservative scheme, correct computation is guaranteed at an arbitrary point of the simulation, while in an optimistic scheme speculative errors may occur, but they will be corrected before the simulation completes. Chandy-Misra-Bryant methods [5, 4] are the best-known conservative simulation 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.

Protocol	Shared				Inserted	
	Read		Write		Read	Write
	hit	miss	hit	miss		
Berkeley						
Illinois		Y			X	
Firefly		Y		Y	X	X

Table 1: Communication requirements for different protocols in the conservative simulation. An entry with "Y" means that synchronization is required. An entry with X means that communication might be necessary but detection is impossible using only local information.

In this paper, we consider the evaluation of the memory hierarchy of multiprocessor systems via parallel trace-driven simulation. We have designed and implemented a conservative simulator based on a method proposed by Lin et al. [10, 11], and an optimistic simulator based on the Time Warp algorithm [12]. In Section 2, we describe the ideas, algorithms and important data structures for the conservative and the optimistic simulations respectively. A qualitative comparison between the two schemes is then presented in Section 3. In Section 4, we report on the performance of these two simulators running on a KSR system and using traces of real applications. We further compare quantitatively the overall speedups and the major causes of overhead that contribute to less than ideal speedups. Finally we give conclusion in Section 5.

2 Parallel Trace-driven Simulation

Our goal is parallel trace-driven simulation of shared-memory multiprocessors. Each processor of the target system has a private cache memory. Processors are connected to each other and to global memory via an interconnection network. We simulate three different protocols – Berkeley, Illinois, and Firefly [1] – that vary in the type and amount of communication involved in the target system and henceforth will impact the performance of the parallel simulations. We focus our attention on snoopy shared-bus systems although the techniques we describe can be adapted to directory-based systems with other types of interconnection network. The input to the simulation is a multiprocess trace – a set of memory address trace files, each of which consists of two types of references: private references and shared references with only the shared references having potential effects on the status of other processors' caches. Since our main interest is in comparing the feasibility and performance of the methods, we restrict ourselves to computing the hit ratio of each cache as the output of the simulations. In both methods, we used the natural mapping of one physical processor and its cache to one logic process. Each simulation process receives as input a memory trace of a physical processor.

2.1 The Conservative Trace-driven Simulation

Figure 1 shows a conservative parallel simulation diagram proposed by Lin et al. [10]. The basic idea of the algorithm is to preprocess input traces by inserting, for each of

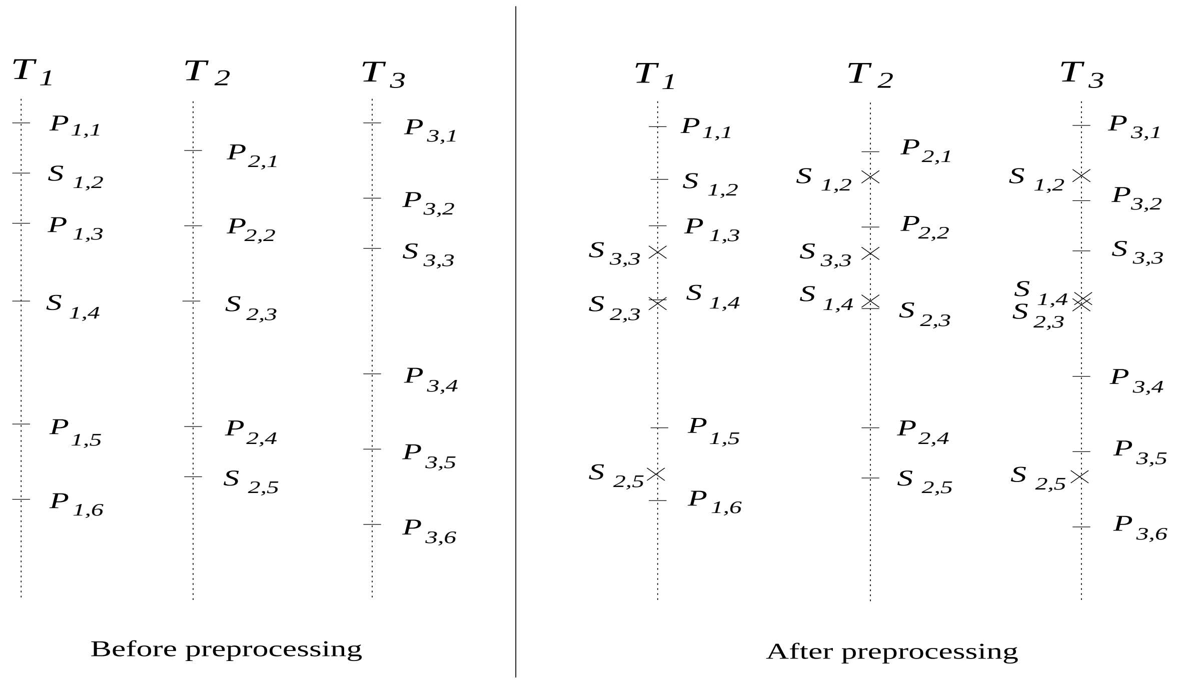

Figure 1: Preprocessing a multiprocessor trace. P and S stand for private and shared reference respectively.

them, their shared references into all the other trace files. The benefits of preprocessing are twofold. Firstly, all the points where interactions are potentially needed are identified before a simulation starts. As a result, communication and synchronization can be performed at very low cost. Secondly, after the preprocessing, part of the cache coherence events, i.e., actions that involve the interconnection network in target systems, can be treated locally by the simulation of the inserted references. How many of these events are local and how many require communication is protocol dependent. For example, in simulating the Berkeley protocol, suppose R_i is a shared write miss, meaning it should invalidate the cache line corresponding to R_i in other processors' caches. With preprocessing, R_i will appear in every trace file. Other processes can perform locally the invalidation when they encounter the inserted R_i. As a second example, let R_i be a shared miss in cache C_m in the Illinois protocol. The state of the line where R_i is mapped depends on the states of the corresponding line in the other caches. Thus the simulation of C_m is at a synchronization point and must wait for messages from other caches. The other caches upon processing the inserted reference corresponding to R_i will send a message to C_m independently of whether or not their own clock is ahead or behind C_m's and whether the message is useful or not (e.g., redundant). To facilitate fast message passing and processing, we assigned a dedicated message queue M_{ij} to each pair of simulation processes C_i (sender) and C_j (receiver) and placed M_{ij} on C_i's processor since it is used more often by the sender. Further communication optimization can be found in [11]. Table 1 displays where communication might be necessary in simulating various cache coherence protocols. Figure 2 sketches the conservative simulation algorithm.

```
 paralleldo i from 1..N
     while ( not  end of trace input i )
       read in a memory reference event R ;
       simulate R ;

       case (R.type)
         private:     noop;
         shared :     if  ( R is a synchronous point )
                      then  wait until it  receives a message about
                            the corresponding inserted reference
                            from each other cache simulation process ;
         inserted:    if  ( R is a synchronous point )
                      then  send a message to the simulation process
                            whose input trace contains the
                            corresponding shared reference;
      endcase ;

      update the status of cache i ;
    endwhile
endparalleldo
```

Figure 2: The conservative simulation algorithm for N traces.

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

Table 2: Messages used in optimistic simulation.

2.2 The Optimistic Trace-driven Simulation

For a given logical process in the optimistic simulation, 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. Table 2 lists the messages generated for the three cache coherence protocols.

The basic idea of adapting 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 advance their local simulation clocks separately, synchronizing with (a subset of) other processes only when the references or the messages require to do so. 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 of whether the message will cause a rollback is needed. Rollback calls for recovery of the simulation 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.

Figure 3 displays 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 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. While saving all the past simulation events (references and incoming messages) and outgoing messages is necessary, the state vector can be saved only at important events, say at a cache miss. The negative side effect of this strategy is that the rollback distance may be increased significantly due to sparse state saving for applications exhibiting good data locality. The augmentation in the rollback distance can be regulated by a timeout parameter $T_{timeout}$ which forces state saving if cache states have not changed for $T_{timeout}$.

One of the important data structures supporting the state saving is the log stack (LS).

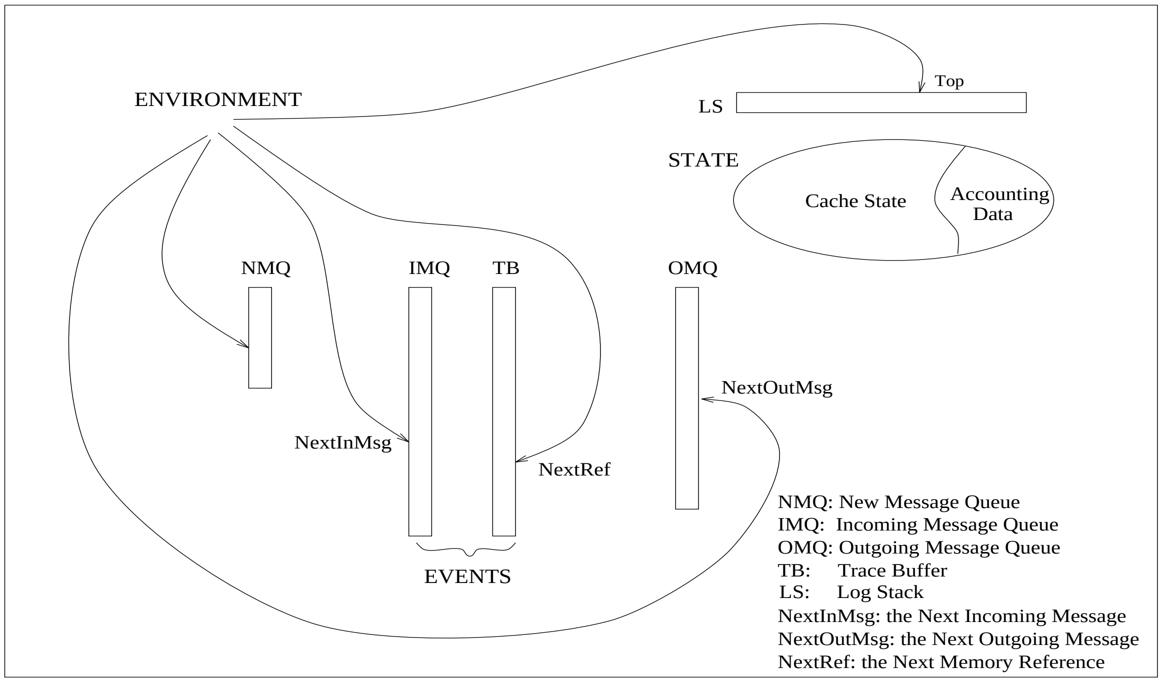
ENVIRONMENT
Top
LS
STATE
Cache State
Accounting Data
NMQ
IMQ
TB
OMQ
NextOutMsg
NextInMsg
NextRef
EVENTS
NMQ: New Message Queue
IMQ: Incoming Message Queue
OMQ: Outgoing Message Queue
TB: Trace Buffer
LS: Log Stack
NextInMsg: the Next Incoming Message
NextOutMsg: the Next Outgoing Message
NextRef: the Next Memory Reference

Figure 3: The data structures of a logic process in optimistic trace-driven simulation.

Every change on the state of a cache line 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 invoked, the process pops entries from the log stack and reconstructs the state vector of an earlier time based on the recorded changes.

The events to simulate are memory references from the input trace and incoming messages from the incoming message queue (IMQ). IMQ is used to store in timestamp order "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). Similarly, the outgoing message queue (OMQ) is used to store all the positive messages being sent out.

Figure 4 sketches the optimistic trace-driven simulation algorithm. The algorithm incorporates a lazy re-evaluation test for rollback and lazy cancellation during rollback. Lazy re-evaluation relies on a more sophisticated test than the simple comparison of timestamps. The test determines not only that there is a late message but also whether this late message will change the state of the cache line that it refers to. Lazy cancellation, during rollback, prevents the sending of unnecessary anti-messages.

3 Qualitative Comparison of the Conservative and the Optimistic Methods

A major difference between the conservative and optimistic approaches to trace-driven simulation is that a fair amount of the communication burden is avoided in the conservative

```
while ( more references or more messages ?) {
   if ( new message in NMQ? ) {
     //  yes, there are new messages
      . Insert new messages in timestamp order in IMQ.
        // Anti-messages are not inserted into IMQ
        // Those messages that anti-messages mean to cancel will be
        //   removed  from IMQ
      . Examine new messages in timestamp order to see if any of
        the new messages causes rollback. If rollback is in order:
          . Restore state vector and environment to the time
            of the late message.
   }

   if ( timestamp of next reference < timestamp of next message ) {
        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 already in OMQ )
             do not send the message
        else send it
        // some anti-messages may be sent out in the matching process.
        // anti-message do not enter OMQ. They remove their
        //   positive counterparts from OMQ.
   }
}
```

Figure 4: The optimistic simulation algorithm.

method by use of preprocessing. Although preprocessing is time consuming since all shared references have to be gathered and sorted in time-stamp order, it has to be done only once and its cost can be amortized over subsequent simulations. There is also a space issue since it appears that all traces are becoming longer but in fact this can be mitigated to a great extent by sharing the shared reference file (implementation details can be found in [11]).

Preprocessing adds a significant advantage to the conservative method and makes it unique in a number of ways. In typical Chandy-Misra-Bryant conservative methods, frequent synchronizations are required to decide when it is safe for a process to proceed. In the conservative approach here, these decisions are made statically since the insertion of shared references mandates when messages will be sent. Consequently the communication and synchronization costs during the actual simulation become very cheap. Moreover, some of the cache coherence activities that involve the interconnection network in the target system are now converted to local simulation actions. This is most striking in the simulation of the Berkeley protocol where no communication between logical processes is required.

However, there are drawbacks to the preprocessing. Since every shared reference is inserted in every trace, there is a large amount of processing of irrelevant references and, in the case of the simulation of the Illinois and Firefly protocols, of sending either extraneous or redundant messages.

This preprocessing and additional reference processing is not necessary in the optimistic approach. Processes forge ahead, communicating or synchronizing only when necessary. However, since the simulation is speculative, state must be saved, tests for rollback are necessary, and rollback themselves do occur. The overhead of state saving, rollback detection, and rollback processing can be tuned by balancing the frequency of state saving versus the rollback distance, and the complexity of rollback testing versus the number of rollbacks. These optimizations are not possible in the straightforward conservative simulation.

The benefit obtained by paying the price of the overhead of state saving and rollback processing is that the logical processes can exploit speculation and lookahead [7]. This is particularly applicable to cache simulations where most events are only partially ordered (cache line states are independent of each other). Often events are processed out of (time) order but still lead to correct states; for example coherence messages for two different lines are completely independent and can be treated in any order.

In terms of implementation, it is obvious that choosing the most efficient data structures, managing the memory, and optimizing the code is much more difficult in the optimistic approach. Preprocessing in the conservative approach is trivial. The simulation itself (recall Section 2) presents no real challenge. On the contrary, the dynamic data structures for the optimistic approach are not as simple, their management is quite elaborate (specific messages in the simulation are devoted to the reclaiming of memory and synchronization of logical processes that might be running "too fast" [12]) and, for example, procedures to restore state, test for rollback and cancel messages only when necessary require care. Furthermore, as in any speculative paradigm, the optimistic simulation is hard to debug.

A definite advantage of the optimistic approach over the conservative one is that it is more flexible and more general. The Time Warp-like simulation can be used to simulate detailed memory behavior including the semantics of atomic reference instructions such as locking. Also it can be more readily adapted to comprehensive instruction-level execution driven simulation.

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. Cache miss numbers are based on the optimistic simulation [1].

4 Performance Results

4.1 Applications and Traces

Four applications were chosen to measure the performance of the parallel simulators. They are Water, Locus, Mp3D and Maxflow. The first three are in the Splash benchmark suite [14]. 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 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 [6].

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.

[1]Each trace-driven simulation correspond to a (not “the”) possible execution. Thus there might be slight difference in misses between the conservative and optimistic simulations.

4.2 Experiment Results of the Conservative Simulation

4.2.1 A simple performance Model

The overhead of the parallel simulation consists of the overhead of processing the inserted references and the overhead of communication and synchronization. For the Berkeley protocol, there is no communication and synchronization overhead at all. Define N, L and f_{share} as the number of input trace files (i.e., the number of processors involved in an application), average length of each trace, and the portion of shared references respectively. Then in the conservative parallel simulation, the (average) number of references to be processed by a single process will be $L + L \times f_{share} \times (N-1)$. Assume that it takes a time unit to simulate one memory reference. The sequential simulation time is:

$$T_{seq} = N \times L$$

The parallel simulation time, without communication, is:

$$T_{para} = L + L \times f_{share} \times (N-1)$$

The speedup upper bound is therefore:

$$\begin{aligned} MAXSpeedup &= \frac{T_{seq}}{T_{para}} \\ &= \frac{N \times L}{L + L \times f_{share} \times (N-1)} \\ &= \frac{N}{1 + f_{share} \times (N-1)} \end{aligned} \tag{1}$$

4.2.2 Experiment results

Application	Speedup (12 processors)			
	Berkeley	Illinois	Firefly	Upper bound
Water	9.2	8.2	7.7	10.2
Locus	7.2	6.3	6.2	8.3
Mp3d	5.9	5.1	5.0	6.4
Maxflow	5.0	4.1	3.8	5.5

Table 4: Speedups of the conservative parallel simulation.

Table 4 displays the performance of the conservative simulation and its upper bound. For any given application, the speedups decrease from the Berkeley simulation to the Firefly simulation because the communication and the synchronization required in the simulation (cf. Table 1) increases monotonically from Berkeley to Firefly while each simulation executes the same amount of references.

When reading Table 4 column-wise, we see that the speedups decrease from Water to Maxflow for each of the three cache protocols. This is due to the increase in the portion of

Application/ Protocol	References Executed	Messages Received	Synchro Points
Water/Berkeley	3510945	0	0
Water/Illinois	3510945	413062	1554
Water/Firefly	3510945	471893	6951
Locus/Berkeley	4335038	0	0
Locus/Illinois	4335038	1027604	3864
Locus/Firefly	4335038	1274616	12735
Mp3d/Berkeley	5575601	0	0
Mp3d/Illinois	5575601	1241451	5584
Mp3d/Firefly	5575601	2546271	84618
Maxflow/Berkeley	9257321	0	0
Maxflow/Illinois	9257321	3946269	26147
Maxflow/Firefly	9257321	4420917	86992

Table 5: Statistics of the conservative parallel simulation: Columns give the application/the cache coherence protocol, the number of references processed, the number of messages received and the number of synchronization points.

the shared references, from Water with only 1.56% to Maxflow with almost 10.90%. The portion of the shared references (read/write) determines the amount of inserted references to be executed as well as the amount of messages to be received. The more shared references there are, the more extra references to be simulated, and the more message passing there will be. As shown in [11], with careful design in data placement and communication optimization, message passing can be conducted efficiently. As for the synchronization overhead, we measured the time that each process waited in execution. It turns out that the synchronization overhead is a small fraction of total execution time (at most 7%).

If we look at the maximum speed in the conservative framework, it is quite clear that when the level of sharing becomes significant the processing of the inserted references is the major factor that limits the performance of the parallel simulation (a 2:1 factor from Water to Maxflow).

4.3 Experiment Results of the Optimistic Simulation

The overall performance (cf. Table 6) of the optimistic simulation depends on many factors. In contrast with the conservative approach, we cannot find metrics that yield either an upper bound on the speedup or a ranking among protocols. The speedup of the optimistic simulation is mainly governed by three countering forces, emanating from a combination of miss ratio (application) and protocol:

- the communication overhead (processing of messages), which is determined by the total amount of messages.

Application	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 6: Speedups of the optimistic parallel simulation.

Application/ Protocol	References Executed	Messages Received	Synchro Points
Water/Berkeley	3495289	37176	0
Water/Illinois	3177643	46786	1572
Water/Firefly	3033988	126592	5762
Locus/Berkeley	3476559	83042	0
Locus/Illinois	3676484	115218	3914
Locus/Firefly	3188808	282306	12814
Mp3d/Berkeley	4274450	260076	0
Mp3d/Illinois	4100421	294916	6513
Mp3d/Firefly	2954986	1890176	85911
Maxflow/Berkeley	6179143	533940	0
Maxflow/Illinois	6090350	803875	26537
Maxflow/Firefly	4213948	1914363	87015

Table 7: Statistics of the optimistic trace-driven simulation: Columns give the number of references that are actually executed, the number of messages received and the number of synchronization points, e.g., places where *Request* messages are issued.

- 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.

The simulation of the Berkeley and of the Illinois protocols often have comparable rollback overhead which can be estimated by looking at the number of extra references simulated (compare the first columns of Tables 3 and 7). In general, the Illinois simulation generates more messages and bears a higher synchronization overhead; its speedup will therefore tend to be lower than Berkeley's. (For detailed analysis on the amount and the types of the messages generated by a particular cache coherence protocol and their impact on the performance, see [12]). The speedup when simulating the Firefly protocol will be even lower in cases where the communication and synchronization overhead outweighs the rollback overhead. The simulations of 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's is about 6 times that needed in Berkeley's (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 and thus the optimistic approach performs somewhat like the conservative scheme. The situation with 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, i.e., generating a large rollback distance. It can be computed (from Tables 3 and 7) 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 simulation 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 Water.

4.4 Quantitative Comparison of the Two Simulation Methods

Both approaches yield significant speedups (cf. Tables 4 and 6). There is no definite trend on which approach performs better than the other either by application or by protocol. Data in Tables 5 and 7 allow us to elaborate more on the various trade-offs but we cannot reach a definite conclusion.

By comparing the first column of Table 3 that gives the number of references in the original traces with the first columns of Tables 5 and 7, we can see the amount of extra reference processed in each method. When the level of shared references is high in applications such as Maxflow, the conservative simulation processed substantially more references than the optimistic one. However, the extra references processed in the optimistic simulation are most costly since they correspond to rollbacks and hence a cost for rollback testing and state recovery had to be incurred. Thus basing our comparison only on the number of references processed would be erroneous.

Now consider the amount of messages transmitted in parallel simulations. In the conservative framework, as mentioned earlier, communication due to the cache coherence activities in target systems is partially (or totally in the Berkeley simulation) taken care of by the simulation of the inserted references. In the cases where communication is needed in both methods, i.e., the Illinois and the Firefly simulations, the conservative method always transmits more messages than the optimistic one. However the processing of the messages in the conservative simulation is much cheaper for two reasons. First, messages in the conservative simulation must be arriving in ascending timestamp order. They can enter message queues and be processed in a FIFO manner, while new messages in the optimistic simulation need to be sorted and inserted into the message queue at appropriate positions. Second, most of the messages sent in the conservative simulation are not needed. The recipients can quickly skip over such messages by using an efficient search scheme such as binary search

[11]. In fact the amount of messages that are actually *processed* after the communication optimization in the conservative simulation is nearly the same as the amount of messages in the optimistic simulation. Thus, despite the larger number of messages *transmitted*, the overall message passing overhead incurred in the conservative simulation is not necessarily higher than it is in the optimistic simulation.

Finally, the last columns of Tables 5 and 7 show that the two simulation paradigms encounter approximately the same amount of synchronization points. But, again, synchronization in the optimistic framework has a larger negative effect performance-wise because in addition to waiting for others to supply cache state information, processes are limited, by synchronization, in their capability to exploit the weak ordering constraints.

In summary, when the two methods execute roughly the same amount of references, e.g. in simulating Water under the Berkeley protocol, the conservative simulation will perform better. In most cases, the conservative simulation executes far more references than the optimistic simulation, but each extra reference, message, and synchronization cost less. The essential reason for the optimistic simulation to perform well is that the trace-driven simulation does not require a total order on the events to be simulated. With a fair amount of sharing but not too many misses and not too much synchronization (e.g., Water and Locus under Illinois), the optimistic simulation is advantageous. However as the synchronization becomes more and more frequent (e.g. Mp3d under Firefly), the capability of exploiting the partial event order is weakened. Accordingly, the performance of the optimistic parallel simulation in this case is worse than that of the conservative simulation.

5 Conclusion

In this paper, we have studied and compared two parallel simulation schemes. Performance-wise, both methods yield significant speedups. The conservative approach relies on preprocessing to identify all the interaction points statically by inserting shared references in every trace. The overhead of processing the inserted references is traded-off against a low cost in communication operations. The performance is bounded by the portion of shared references in the applications. By contrast, the optimistic approach allows processes to exploit weak ordering rather than total ordering constraints, communicating and synchronizing with each other only when necessary. Despite the fact that each extra reference, message and synchronization cost more in the optimistic simulation, in most cases, the overall performance of the optimistic simulation is slightly better since far fewer references are processed than in the conservative simulation. However as synchronization becomes more and more frequent, the optimistic simulation eventually loses its performance advantage. The major sources of overhead in the optimistic simulation – rollback and communication and – are inherently determined by the shared reference miss ratio. For the four applications used in our experiments, the shared reference miss ratio rises as the level of sharing increases. Thus the two simulation schemes present similar patterns of performance degradation with increased sharing.

In terms of implementation, the optimistic simulation requires considerably more effort in refining the algorithm – choosing appropriate optimizations, designing efficient data structures and managing memory. On the other hand, the optimistic approach has the definite advantage over the conservative one that it can be adapted to comprehensive instruction-level simulation as well as detailed simulation of memory behavior including the semantics of atomic reference instructions such as locking.

References

[1] James Archibald and Jean-Loup Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model". ACM Transactions on Computer Systems, Vol.4, No.4, November 1986, 273-298.

[2] Eric A. Brewer, Chrysanthos N.Dellarocas, Adrian Colbrook and William E.Weihl. "PROTEUS: A High-Performance Parallel-Architecture Simulator". Technical Report MIT/LCS/TR-516, Laboratory for Computer Science, MIT.

[3] Eugene D. Brooks III, Timothy S. Axelrod, Gregory A. Darmohray. " The Cerberus Multiprocessor Simulator". Parallel Processing for Scientific Computing, 384-390, SIAM, 1989.

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

[5] K.M. Chandy and J. Misra. "A Case Study in Design and Verification of Distributed Programs". IEEE Trans. on Software and Engineering, Sept 1979, 440-452.

[6] 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, pp37-47, 1990.

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

[8] H.Davis, S. Goldschmidt and J.L. Hennessy. "Multiprocessor simulation and tracing using Tango". Proceedings of the 1991 International Conference on Parallel Processing, Vol. I, pp.99-107, 1991

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

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

[11] X. Qin and J.-L. Baer. "A Parallel Trace-driven Simulator: Implementation and Performance". Proceedings of International Conference on Parallel Processing, August, 1994 II314-II318.

[12] X. Qin and J.-L. Baer. "Optimistic Trace-driven Simulation". Submitted for publication.

[13] Steven Reinhardt, Mark Hill and James Larus. " The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers". 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, pp48-60, 1993.

[14] Jaswinder Pal Singh, Wolf Dietrich Weber and Anoop Gupta. "SPLASH: Stanford Parallel Applications for Shared-Memory". Computer Architecture News, Vol.20, No.1, pp5-44, March 1992.