A Parallel Trace-driven Simulator: Implementation and Performance*

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

September 8, 1994

Abstract

The simulation of parallel architectures requires an enormous amount of CPU cycles and, in the case of trace-driven simulation, of disk storage. In this paper, we consider the evaluation of the memory hierarchy of multiprocessor systems via parallel trace-driven simulation. We refine Lin et al.[10] original algorithm, whose main characteristic is to insert the shared references from every trace in all other traces, by reducing the amount of communication between simulation processes. We have implemented our algorithm on a KSR-1. Results of our experiments on traces of four applications and three different cache coherence protocols show that parallel trace-driven simulation yields significant speedups over its sequential counter-part. The communication overhead is not substantial compared to the dominant overhead due to the processing of replicated inserted references.

We also investigate filtering techniques for multiprocessor traces. We show how to filter –in parallel– private and shared references. Our technique generates filtered traces for various block sizes in a single pass. As expected, the simulation of filtered traces is much faster but parallel simulation of filtered traces is not as effective since the ratio of unfiltered shared to private references is now much larger.

*This work was supported in part by NSF Grants CCR-91-01541 and CCR-91-23308

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[3] and Tango[8] run on single processor workstations. Recently execution-driven parallel simulators for parallel architectures [12, 4] have been implemented on specific parallel architectures (the TMI CM-5 and BBN Butterfly respectively). The challenge in these parallel simulators is to have 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 with parallel simulation.

In this paper, we consider the evaluation of the memory hierarchy of multiprocessor systems via parallel trace-driven simulation. Trace-driven simulation can be used to simulate the effect of various cache coherence protocols, cache configurations and organizations, and can also take into account the network topology and its parameters [5]. We consider only simulation with real traces as input since the generation of synthetic traces does not appear to offer any speed advantage in a multiprocessor environment [2] with the drawbacks of having to choose the system and application parameters to characterize the resource demands of a certain workload. Our experiments will be based on real traces collected on the Sequent multiprocessor system using MPTrace [6].

The remainder of the paper is organized as follows: In Section 2 we present the basic idea of the parallel trace-driven simulation[10]. In Section 3 we describe how the communication problem was handled in the original algorithm and our techniques to reduce the amount of unnecessary communication. In Section 4 we discuss implementation issues. In Section 5 we present the performance results of the parallel simulation and the speedups that were achieved using a KSR-1 system. Section 6 shows how filtering, a technique used

in uniprocessor trace driven simulation to reduce time and space requirement, can be incorporated successfully in parallel trace-driven simulation. Conclusions are given in Section 7.

2 The Basic Parallel Trace-driven Simulation Algorithm

Our goal is parallel trace-driven simulation of multiprocessor architectures. The target system 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 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.

The basic idea of the parallel simulation [10] is to preprocess the input traces so that the shared references of each input trace are inserted into all the other input trace files. Then the preprocessed input trace files for each simulated processor can be read in and simulated concurrently. If the input traces included a timestamp for each event, there would be no difficulty in the insertion process. We could simply merge the shared references into the other trace files by their timestamps. However, since the timestamp is not recorded in the traces, we compute a pseudo timestamp for each event.

Let t_{i-1} be the timestamp of the last event e_{i-1}. Then the timestamp t_i for the current event e_i is computed as follows:

1. if e_i is a memory reference to a private variable, then $t_i = t_{i-1} + c_{pri}$, where c_{pri} is the average number of cycles that a private reference takes. Since private references have very high cache hit ratio, we assume that $c_{pri} = 1$.

2. if e_i is a memory reference to a shared variable, then $t_i = t_{i-1} + c_{sh}$, where c_{sh} is the average number of cycles that a shared reference takes. In our simulation, we assume that $c_{sh} = 10$.

3. if e_i is an instruction fetch, which is also a private reference, then $t_i = t_{i-1} + c_{pri} + c_{ins}$, where c_{ins} is the number of cycles the instruction takes without the presence of memory latency. c_{ins} is recorded by MPTrace when traces are collected.

It is important that the shared references and the inserted references be kept in the same order in all the preprocessed input traces. This property is a necessary condition to guarantee that the parallel simulation algorithm described below is deadlock free. Note that the traces inserted with shared references from other processors might not be a completely accurate representation of the processes being traced, but they represent traces of "one" possible execution.

When the preprocessed traces are simulated concurrently, it is only when shared references and inserted references are encountered that there might be a need to communicate with other processes. These references are called interaction points. Since preprocessing identifies all interaction points, the brute force approach to parallel simulation would be to barrier synchronize the execution of the simulation processes at each interaction point.

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

Table 1: Communication requirements for different protocols. 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

However it may not be necessary for all simulation processes to rendezvous at each interaction point. Depending on the type of memory operations (read/write) and the cache coherence protocol being simulated, it is possible that, at some interaction points, a simulation process may be able to determine its cache status by only using its local information. For such interaction points, there is no need for communication and the corresponding references can be simulated in a fashion similar to the processing of private references. The rest of the interaction points that still need communication and synchronization are called synchronous points. Table 1 shows the synchronous points for different cache coherence protocols.

A sketch of the basic parallel trace-driven simulation process is then:

```
paralleldo i from 1..N
    while ( not  end of trace input i )
       read in a memory reference event R ;
       case (R.type)
          private:   nop;
          shared :   if  ( R is a synchronous point )
                           wait until it  receives a message about
                           the corresponding inserted reference
                           from each other cache simulation process ;
          inserted:  if  ( R is a synchronous point )
                           send a message to the simulation process
                           whose input trace contains the
                           corresponding shared reference;
       endcase
       update the status of cache i ;
    endwhile
endparalleldo
```

3 The Communication Problem

In order to test the viability of parallel trace-driven simulation, we looked at three cache coherence protocols, Berkeley, Illinois and Firely [1]. They are interesting because the number of synchronization points increase with each protocol (see Table 1).

According to Table 1, there is no need for any kind of communication for the Berkeley protocol (see [10] for a proof) since each cache simulation process can always decide its status based on its own local information. The parallel simulation processes of the Berkeley protocol can run independently and therefore the parallel simulation of this protocol represents the best case, communication-wise. In the case of the Illinois protocol, only shared read misses are synchronous points. When a read of a shared reference, say r_t, misses in the cache C_i (in the following we will denote by C_i either the cache itself or the simulation process simulating that cache's behavior), its simulation process needs to communicate with the other cache processes to determine the state of the missing line. This communication will occur when the other cache processes reach the inserted shared reference corresponding to r_t. In the case of the Firefly protocol, this communication is required for both shared read and shared write misses.

As can be seen from Table 1 there is no difficulty for a cache simulation process to figure out whether a *shared reference* is a synchronous point since all the required information is local (type of operation and hit/miss information). However, whether communication is required when an inserted reference is processed cannot be determined locally by the simulation process encountering the inserted reference. Instead Instead it depends on whether its corresponding *shared reference* is a synchronous point or not. The local information which was sufficient for shared references is no longer enough to decide when it is necessary for an inserted reference to communicate.

Lin's original algorithm proposed a conservative approach, i.e., messages would be sent when inserted references corresponding to potential interaction points were encountered. In the case of the Illinois protocol, a cache simulation simply sends out messages for all the *inserted reads*. In the case of the Firely protocol, messages are sent for all the *inserted references*. Obviously this method could introduce a lot of unnecessary communication.

To reduce the communication overhead, we seek other information in order to exclude extraneous sends and receives. We maintain a simulation time for each cache process C_i. The simulation time is computed in the same way as the pseudo timestamp (Section 2).

- Case 1: C_i, with simulation time T_i, is simulating an inserted reference R^i_{ins1} with corresponding shared reference R_{sh1} in C_k whose current simulation time is T_k.
  - (1.1) If C_i is slower than C_k ($T_i < T_k$), then C_k must have passed the shared reference R_{sh1} without waiting for a message from C_i on the result of R^i_{ins1}. This implies that R^i_{ins1} is not a synchronous point. Thus no message need to be sent.
  - (1.2) Otherwise ($T_i \geq T_k$) C_i sends out a message to C_k to inform C_k of its status on simulating R^i_{ins1}.
- Case 2: C_i is simulating a shared reference R_{sh2} which is a synchronous point at time T_i. It needs the status of other processes. Let C_j be another process with current simulation time T_j and let R^j_{ins2} be the inserted reference corresponding to R_{sh2}.

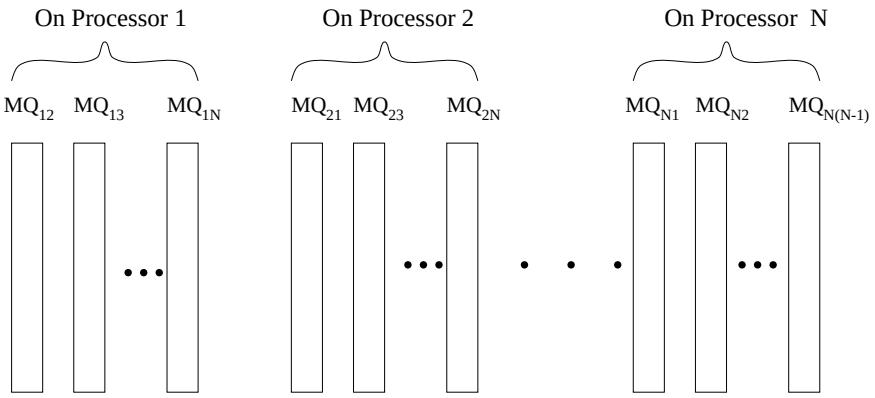

Figure 1: The data structure of message queues

- (2.1) if C_i is slower than C_j ($T_i < T_j$), it must be the case that C_j has simulated R^j_{ins2} since R^j_{ins2} and R_{sh2} have the same timestamp. When C_j was simulating R^j_{ins2}, it was not able to judge whether R^j_{ins2} was a synchronous point or not because at that time C_j's simulation time was greater than C_i's also. Thus, following 1.2, C_j already sent a message to C_i, say at time t_0. In fact, C_j might have sent other messages to C_i prior to t_0 that were not read by C_i since they did not correspond to synchronization points. Since, as described below, messages between C_j and C_i are entered in a queue in a FIFO manner, ordered by timestamp, more efficient techniques than a linear search can be used, e.g. binary search, to retrieve the message corresponding to R^j_{ins2}.
- (2.2) If C_i is faster than C_j ($T_i \geq T_j$) it has to wait for a message from C_j to arrive. When C_j encounters the inserted reference R^j_{ins2} at T'_j while C_i is waiting, it must be that $T_i = T'_j$. According to 1.2, a message *will* be sent from C_j to C_i.

The major data structure to facilitate the interprocess communication is a set of message queues. For each pair of cache simulation processes C_i(sender) and C_j(receiver), there is a dedicated message queue MQ_{ij} (cf. Figure 1). Due to the fact that C_i needs to write to a message queue (message send) more often than it needs to read from it (message receive), we choose to place the queue closer to its writer than to its reader.

We can now present a more detailed version of the parallel trace-driven simulation algorithm.

```
paralleldo i from 1..N
   while ( not  end of trace input i )
      read in a memory reference event R ;
      case (R.type)
         private: nop;
         shared :
           if  ( R is a synchronous point )
               {
                 for  j = 1 .. N && j != i {
                    if ( len (msg_queue[j][i] >0 )
```

```
                        search  the message queue and extract
                        the useful message;

                        update the message queue by deleting
                        the messages that are older than the
                        requested message ;
                       }
                    else {
                        busy waiting until the message arrives
                       }
                  }
               }
          inserted: /* R is inserted from trace j */
            if  ( R is a synchronous point )
               if  ( simu_time[i] < simu_time[j] ) {
                     send a message to the simulation process
                     whose input trace contains the original
                     corresponding shared reference;
                  }
       endcase
       update the status of cache i ;
    endwhile
endparalleldo
```

4 Implementation

4.1 Preprocessing

The preprocessing of input traces calls for the insertion of the shared references into all the other trace files. If this were implemented literally, the total size of the traces would expend significantly. Assume that we are tracing an application on N processors, i.e., we have N input trace files, each of average length L, and with a proportion f_{share} of shared references. Then the total amount of extra memory needed to store the inserted references is $N \times (N-1) \times L \times f_{share}$. For example, with $N = 10$ and $f_{share} = 5\%$, the extra memory required would be about 50% of the original trace. To mitigate the potential pressure on the disk space, we create a temporary file by extracting all the shared references from the trace files and assigning a timestamp to each event (as mentioned in Section 2). In addition to the timestamp, the inserted references carry also the identification of the process that generates the shared reference. Then we sort, in increasing timestamp order, the shared references from this temporary file into a *shared reference file*. The amount of extra space needed for this implementation is $N \times L \times f_{share}$, i.e., an overhead proportional to N rather than to N^2. The tradeoff for the memory (on disk for archival storage and in main memory while processing) saved by this approach is the extra amount of time required for merging the shared reference file with the individual trace streams during simulation. The time to extract and sort the shared reference file is a one-time cost that is roughly equivalent to the

cost of merging inserted references if we were to adopt a straightforward implementation.

4.2 Merging the Shared Reference File with the Individual Trace File on the Fly

From the point of view of a specific simulation process C_i, the shared reference file contains shared references, the ones with C_i's own id, and inserted references, those with the id's of other processes. The input trace files containing the original private and shared references are not modified.

When performing the parallel simulation, each simulation process C_i has two input streams, its original trace file F_i, and the shared reference file F_{sh}. C_i computes the simulation time for F_i on the fly in the same way as the timestamps in the shared reference file were generated. Thus the simulation time and the timestamps of the shared and inserted references have the same meaning. Let t_i be the current simulation time for C_i and t_{sh} be the timestamp of the next unprocessed (by C_i) reference in F_{sh}. We compare t_i with t_{sh} to decide which of the two references e_i in F_i or e_{sh} in F_{sh} to process:

1. if $t_i < t_{sh}$, simulate e_i, the next reference in F_i, and e_i is a private reference.
2. If $t_i > t_{sh}$, simulate e_{sh} and e_{sh} is an inserted reference.
3. If $t_i = t_{sh}$, then simulate "both" with
   (a) if the id of e_{sh} is C_i's id, then simulate as a shared reference.
   (b) otherwise, simulate as an inserted reference.

The simulation process can then proceed as in the algorithm given in Section 3.

4.3 The Communication Cost on Shared Memory Cache Coherent Architectures

Our parallel trace-driven simulator has been implemented on KSR-1 – a shared memory architecture with coherent caches [9]. Each simulation process is on a different KSR-1 node (processor + cache). Communication between the simulation processes is through sender/receiver message queues (see Figure 1) located in the sender's memory. Each of the $N-1$ initially empty queues on each processor is a circular buffer whose empty elements will automatically be recycled. Send and Receive are implemented as follows.

Message Send: When a process C_i needs to send a message M_1 to C_j, it writes M_1 into the message queue MQ_{ij}. Suppose $ADDR_1$ is the address where M_1 is to be written and it maps to cache block B. There are two possible situations: (1) No other process has cached $ADDR_1$ and (2) there is one, most likely C_j, or more processes which happen to have copied the content of B due to reading some old message in $ADDR_1$. In the former case, since C_i is the exclusive owner of the cache block B, the write can proceed locally. In the latter case, C_i has to invalidate the block B on other processor(s), which is as expensive as a cache read miss.

Message Receive: When C_i is to receive a message M_2 from C_k, it needs to search the message queue MQ_{ki} to find M_2. At this time, all the cache blocks containing messages in MQ_{ki} that have not been read by C_i are in exclusive state in C_k. Therefore in the process

of searching M_2, C_i will incur a series of cache read misses and copy all the messages it touches into its own cache.

On the KSR-1, the memory latency ratio between a cache miss and a cache hit is 8:1, if the item is in the (local) second level cache, and about 70:1 if the item is in the first level cache. From the above description of send and receive operations, we can see that a send is much cheaper than a receive in our implementation. In our parallel simulation algorithm, we mainly rely on the binary search of message queues MQ_{ij} to reduce the number of messages to retrieve in the receive case. In addition, when a simulation process C_i encounters a cache read miss of the target system, it does not necessarily check on *all* the other processes. As soon as C_i receives a message informing it that another process contains the missed cache block, it can decide its cache status and stop inquiring about other processes. For the Firefly protocol, we do the same for the write misses as well. As we shall see later in 5.3, about 70%-90% of message retrieves are saved by the above two optimizations. The reduction in the number of messages retrieved does not only speed up the receiving process but also helps the sending process because less invalidations need to be issued.

4.4 The I/O Issue

Another implementation issue involved in parallel trace-driven simulation is the I/O problem. It is well-known that trace-driven simulation is I/O intensive[2]. For parallel trace-driven simulation, it would be most efficient if concurrent simulation processes could read their own input traces in parallel. Unfortunately, for most parallel systems, parallel I/O, i.e., I/O at each node of the system, is not available. In the system on which we ran our experiments the processes have to go through a single disk read/write header to fetch the data, which means that all the I/O is sequential. This fact would hide any benefit that could possibly be obtained by doing parallel simulation if I/O time were to dominate simulation time. In order to look at the impact of the parallel simulation algorithm, we discount the I/O artifact. We read the traces into buffers and then simulate. This approach would be realistic if we were to perform several simulations on the same set of data so that the I/O overhead could be amortized over multiple simulations; Or if I/O could be performed in parallel, I/O buffering could effectively overlap the I/O activities with the simulation computation. In our simulator, we create a special process for I/O buffering. Before the simulation starts, it reads in a big part of the trace for each process. Then the simulation is started in parallel. During the simulation, whenever one or more of the trace buffers are empty, all the simulation processes are stopped and wait for I/O buffering. The simulation is resumed after the I/O process fills trace data into the empty buffer(s) and the time to do the I/O is not counted in the execution time.

APPL	Num of Refs	Sh Refs(%)	Sh Reads(%)	SR Misses	Sh Writes(%)	SW Misses
Water	2997322	46693(1.56)	39711(1.32)	1554	6982(0.23)	1517
Locus	2997196	121622(4.06)	101242(3.38)	3864	20380(0.68)	2917
Mp3d	2949901	238700(8.09)	131305(4.45)	5584	107394(3.64)	14091
Maxflow	4209327	458954(10.90)	374545(8.90)	26147	84409(2.01)	18148

Table 2: The sharing characteristics of the applications: Columns give name of the application, total number of references (including instruction fetch), number of shared references, number of shared read references, number of shared read cache misses, number of shared write references, and number of shared write cache misses. Cache miss numbers are based on the Berkeley protocol.

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. These applications were selected because they are "real applications", the proportion of shared references varies from application to application so that we can examine the parallel simulation performance as a function of the amount of shared references, i.e., communication, and the traces were already collected.

Among the four applications, the first three are in the Splash benchmark suite [13]. 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.

Table 2 shows the memory access characteristics of the four applications. All of the above applications have 12 input trace files. The data given in Table 2 are average numbers for the multiple trace streams of one application. The caches that were simulated were 512KB, 2-way set associative with a block size of 32 byte.

5.2 A Simple Performance Model

Before we present the performance data of the parallel simulation, we introduce a simple model to estimate the best results that we can expect from the parallel simulation algorithm. The overhead of the parallel simulation consists of the overhead of processing the inserted references and the overhead of communication. In the case of the Berkeley protocol, there is no communication overhead at all. With N, L and f_{share} defined as in Section 4.1, the (average) number of references to be processed by a single simulation process will be $L + L \times f_{share} \times (N-1)$. Assume that it takes a unit time to simulate one memory reference. Then the sequential simulation time is:

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

APPL	maxspeedup	speedup	MSG waiting(%)	save in snd(%)	save in rcv(%)
Water B	10.2	9.2	0	0	0
Water I	10.2	8.2	3.12	5.44	89.42
Water F	10.2	7.7	5.02	12.56	80.54
Locus B	8.3	7.2	0	0	0
Locus I	8.3	6.3	6.09	5.98	88.86
Locus F	8.3	6.2	6.11	7.09	75.81
MP3D B	6.35	5.92	0	0	0
MP3D I	6.35	5.13	4.45	8.64	86.22
MP3D F	6.35	5.01	5.08	14.05	73.16
Maxflow B	5.46	5.01	0	0	0
Maxflow I	5.46	4.14	4.02	4.22	82.97
Maxflow F	5.46	3.82	4.99	34.48	86.64

Table 3: The performance statistics of the parallel simulation: Columns give application and cache coherence protocol (B – Berkeley, I – Illinois, F – Firefly), speedup upper bound of the parallel simulation, real speedup, accumulated time waiting on messages (in percentage of the total execution time), savings in the messages sent, and savings in the messages retrieved. Savings are computed based on the potential communication volume.

The parallel simulation time, without communication, is:

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

The best performance (speedup) that we can expect is therefore:

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

The first column of Table 3 gives the upper bounds of the speedups for the four applications.

The communication overhead in the Illinois and Firely protocols is more difficult to estimate. An upper bound on the number of messages, e.g., if we were to follow Lin's original algorithm, is the number of inserted reads, $(N-1) \times Sh\_Reads$, for the Illinois protocol, and the number of inserted references, $(N-1) \times Sh\_Refs$ for the Firefly protocol. A lower bound would be SR_Misses for Berkeley and $(SR\_Misses + SW\_Misses)$ for Firefly if the processor incurring the miss had an oracle telling it which processor to poll to get the status of the missing line. If this oracle did not exist, like in the simulation, the above numbers would have to be multiplied by some factor between 1 and $N-1$.

5.3 Experiment Results

Table 3 shows the results of our experiments. The data given in Table 3 are average numbers of several simulation runs and represent averages of the multiple streams for a given application. We can make the following observations.

Observation 1: The real speedup of the parallel simulation of the Berkeley protocol is close to its performance upper bound. The source of the difference is in our implementation. As mentioned in 4.1, in order to save disk space, we actually do not insert shared references from other processes into the original traces. Instead we create a shared reference file. When executing the parallel simulation, we experience the overhead of merging the shared reference file with the original trace files. This requires extra timestamp comparisons.

Observation 2: The performance difference between the simulation of the Berkeley protocol and the simulation of the other two protocols is small. This implies that the communication cost introduced in simulating the Illinois and the Firefly protocols does not cause a severe performance degradation. Table 3 shows that the message waiting time, the synchronization overhead incurred in communication, is a small fraction of the total execution time (at most 7%). We also display the savings of the messages sent and received over the initial algorithm. As can be seen, the saving in the amount of the messages sent is not great. However the amount of messages retrieved (in the receive process) is dramatically reduced. Since the "send" are almost always local and much less costly than the "retrieve", the search scheme and the data structure we have implemented are the right choices for reducing the communication overhead.

Observation 3: Usually the computation-to-communication ratio gives a good idea of the performance of a parallel program, with the higher ratio yielding the better performance. In our case, we estimated this ratio as the amount of references processed in the traces divided by the amount of costly communication (messages retrieved). By using this metric, MP3D has a lower ratio than Maxflow when simulating the Firefly protocol. However, the speedup in Maxflow is significantly lower (by about 20%). This is because the most significant part of the overhead, namely the processing of inserted references, is not reflected in the computation-to-communication ratio. As shown in Table 2, Maxflow's percentage of inserted references is higher than MP3D's. This clearly dominates the communication costs. Essentially the insertion of the shared references from other processors is a tradeoff for low cost communication because all the interaction points are identified before the simulation begins and synchronization is very efficient.

The performance data of the Firefly protocol is of particular importance because the communication incurred in the simulation reflects the real amount of bus or interconnect activities that would be involved in the simulated target systems. If more detailed memory system simulations were desired, we would expect that all the shared read and write misses i.e., the references that supposedly involved some bus transaction in the target system, would require some sort of communication among the simulation processes.

6 Filtering

6.1 Filtering Multiprocessor Traces

Very long traces are required to study the behavior of large caches. Reducing the length of the traces will reduce both simulation compute and I/O times and storage space on disk. A number of filtering techniques have been proposed to compact single processor traces [11, 14]. Wang and Baer [15] extended trace reduction to multiprocessor traces. The basic idea behind trace reduction is to make use of the cache inclusion property. If a reference causes a hit in a small direct-mapped cache, it will also be a hit in a larger cache under the condition that the larger cache has the same block size as that of the smaller cache. Therefore, if we are interested in metrics such as hit ratios, we can remove from the traces those references that hit in the small cache.

Note that in multiprocessor systems the filtering will be slightly different. Not only do we need to keep all references that miss in the filter cache but we must also keep the (shared) references that will potentially modify the status of corresponding lines in other caches. For example, shared write references that hit the filter cache in a non-exclusive state must be kept in the reduced trace.

The currently existing filtering techniques can be easily adapted to parallel trace-driven simulation. First, we can filter multiprocessor traces in parallel and, second, we do not need to modify conceptually the parallel simulation algorithm presented previously. There is however one implementation question that requires changes, namely the generation of the simulation clock times while processing the traces. Our solution is to write timestamps into the filtered traces during the filtering process.

6.2 Releasing the Condition of Single Block Size

Straightforward filtering techniques require that the caches to be simulated have the same block size as that of the filter cache. This means that we need to produce and save reduced traces for every possible block size under study. It is not unlikely that the space savings gained from the trace reduction would be erased by the need to save every filtered trace. Wang and Baer [15] advocate generating universal reduced traces by collecting the superset of misses that occur on every cache filter with different block sizes. We describe now a methodology for obtaining such universal reduced traces.

It is easy to oberve that the inclusion property necessary for filtering does not hold when we use caches with different block sizes. Consider the example shown in Figure 2. Cache C_1 is a small direct-map cache with block size of 1 word. Cache C_2 is a larger direct-map cache with block size of 4 words. Let R_1, R_2 and R_3 be 3 read references to words X, Y ($Y \neq X + 1$) and X respectively. Assume that X and Y map to two consecutive blocks in the small cache. Figure 2 shows the contents of the two caches after the execution of R_1 (part A) and R_2 (part B). Now when R_3 is executed, we have a hit in the filter cache C_1 and a miss in the large cache C_2. The same situation can occur in multiprocessor traces upon invalidations. Assume the status depicted in Figure 3 part A. Then, another processor writes X and an invalidation based protocol is used. The resulting status is shown in Figure 3 part B. Now a read reference to $X + 1$ hits in the filter cache and misses in the larger cache.

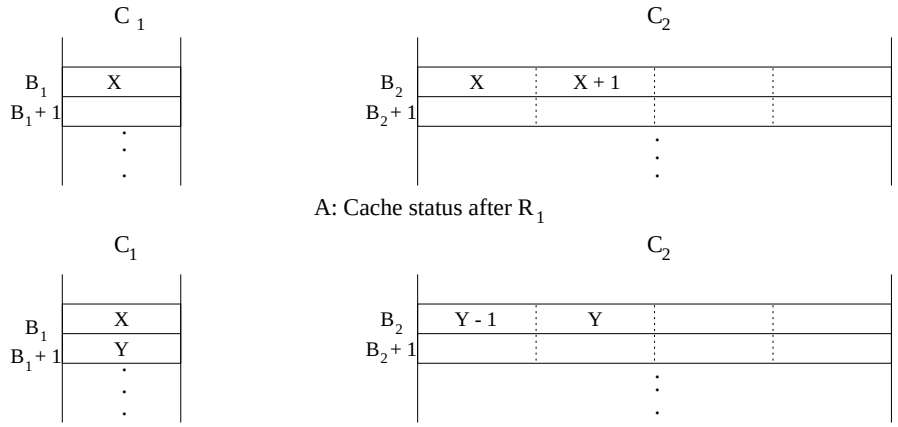

Figure 2: The replacement that violates the cache inclusion property in single processor traces.

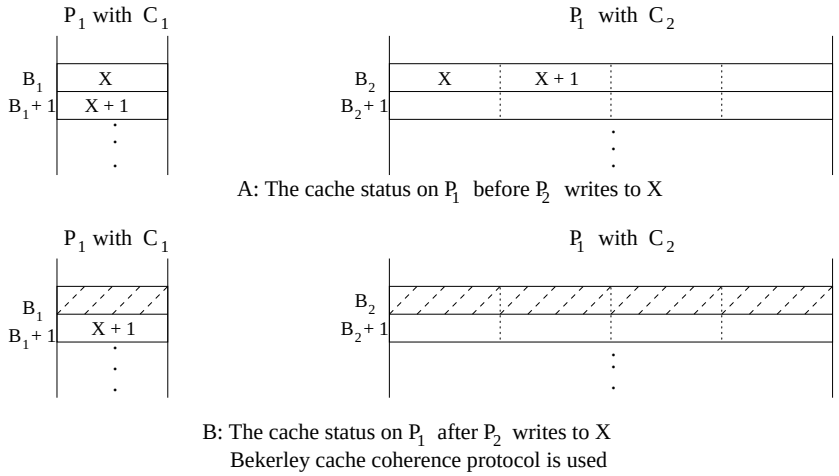

Figure 3: The invalidation that inviolates the cache inclusion property in multiprocessor traces.

APPL		Tot Refs (Rduced %)	Pri Refs (Rduced %)	Sh Refs(Rduced %)
Water	before	2997321	2950628	46693
	after	22186 (99.26)	11729 (99.60)	10457 (77.60)
Locus	before	2997195	2875573	121622
	after	37353 (98.76)	6738 (99.77)	30615 (74.83)
MP3D	before	2949858	2711157	238701
	after	134016 (96.46)	5706 (99.79)	128310 (46.25)
Maxflow	before	4209327	3750372	458955
	after	132274 (96.86)	4125 (99.89)	128149 (72.08)

Table 4: The references preserved during the filtering process. Columns give application, total number of references (including instruction fetch), number of private references and number of shared references before and after filtering.

In order to gather a correct universal reduced trace, we perform extra invalidations in the filter cache whenever cache replacement or cache invalidation is in order. The invalidation size is set to be as large as the largest cache block size one would like to study. More specifically, when a cache block needs to be replaced or invalidated, we invalidate all the cache blocks in the filter cache that would be in conflict with the new cache block if the biggest block size were used. It can be proven (by contradiction) that the cache simulation based on the filtered trace generated by the method described above will produce the same cache miss results as if the full length trace had been used.

Table 4 shows that filtering works very well for the private references and reasonably well for the shared references. We used a rather large filter cache (eight times smaller than the caches used in the experiments). On average over 96% of the private references and about 70% - 80% of the shared references are filtered except for MP3D. The reason that the filtering rate of the shared references of MP3D is low is that about 44% of the shared references are write operations.

Table 5 shows the speedup results based on the filtered traces. Because filtering makes the shared references become the majority of the memory references in the traces, the speedups of the parallel simulation based on the filtered traces are significantly lower than those based on the full trace. In the best case, Water, the speedup is about 2.8. As mentioned before the main cause of the low speedups is the overhead of processing the inserted references.

7 Conclusion

In this paper we have shown that parallel trace driven simulation of multiprocessor traces is viable and can lead to significant speedups. Starting from Lin's conservative algorithm, we have described how the amount of necessary communication could be reduced. Our implementation on a KSR-1 system produces speedup results close to the upper bounds predicted by a simple model. Simulation of the protocols that require more synchronization points do not, as expected, enjoy the same level of speedup. However, the main overhead

APPL	spdup
Water B	2.85
Locus B	1.81
MP3D B	1.60
Maxflow B	1.92

Table 5: The performance of the parallel simulation based on the filtered traces. Columns give application and cache coherence protocol (B – Berkeley) and speedup.

in the method is the processing of the shared references that need to be inserted in all the traces. The simulation speedup decreases when the level of sharing increases in the real application being traced.

In addition we have studied filtering techniques for multiprocessor traces. We generate universal filtered traces that allow subsequent simulations with different block sizes. Our results show that the technique is efficient in filtering both private and shared references. The speedups obtained on filtered traces are not as impressive since a large portion of the filtered traces consist of unfiltered shared references.

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, pp273-298

[2] Luis Barriga and Rassul Ayani. "Parallel Cache Simulation on Multiprocessor Workstations". 1993 International Conference on Parallel Processing, I171-I174

[3] 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

[4] Eugene D. Brooks III, Timothy S. Axelrod, Gregory A. Darmohray. " The Cerberus Multiprocessor Simulator"

[5] David Chaiken, Craig Fields, Kiyoshi Kurihara and Anant Agarwal. "Directory-Based Cache Coherence in Large-Scale Multiprocessors". Computer, June 1990

[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] S.J. Eggers, E.J. Koldinger and H.M. Levy. "On the Validity of Trace-Driven Simulation for Multiprocessors". The Proceedings of International Symposium on Computer Architecture, 1991, pp244-253

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

[10] Yi-Bing Lin, Edward D. Lazowska, Jean-Loup Baer. "Parallel Trace-Driven Simulation of Multiprocessor Cache Performance: Algorithms and Analysis". Progress in Simulation, Vol.1 No.1, Ablex Publishing, pp44-80, 1989

[11] T.R. Puzak. "Analysis of Cache Replacement Algorithms". Ph.D Thesis, University of Massachusetts.

[12] 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

[13] 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

[14] A. J. Smith. "Two Methods for the Efficient Analysis of Memory Address Trace Data". IEEE Transactions on Software Engineering, Vol.3, No.1, pp94-101, 1977

[15] Wen-Hann Wang and Jean-Loup Baer. "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis". ACM Transactions on Computer System, Vol.9, No. 3, pp 222-241, August 1991