Timepatch: A Novel Technique for the Parallel Simulation of Multiprocessor Caches*

Umakishore Ramachandran
Gautam Shah
Ivan Yanasak
Richard Fujimoto

e-mail: {rama, gautam, yanasak, fujimoto}@cc.gatech.edu

Technical Report GIT-CC-96-24

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

Abstract

We present a new technique for the parallel simulation of cache coherent shared memory multiprocessors. Our technique is based on the fact that the functional correctness of the simulation can be decoupled from its timing correctness. Thus in our simulations we can exploit as much parallelism as is available in the application without being constrained by conservative scheduling mechanisms that might limit the available parallelism in order to guarantee the timing correctness of the simulation. Further, application specific details (which can be gleaned from the compiler) such as data layout in the caches of the target architecture can be exploited to reduce the overhead of the simulation. The simulation correctness is guaranteed by patching the performance related timing information at specific points in the program (commensurate with the programming model). There are two principal advantages to this technique: being able to simulate larger parallel systems (both problem size and number of processors) than is feasible to simulate sequentially; and being able to speed up the simulation compared to a sequential simulator. For proof of concept, we have implemented this technique for an execution-driven parallel simulator for a target architecture that uses an invalidation-based protocol. We validate the performance statistics gathered from this simulator (using traces) by comparing it against a sequential simulator. We provide a careful breakdown of where the time is spent in the parallel simulator, and suggest optimizations to reduce the overhead of the simulation. With these optimizations, we show that the timepatch technique is both viable and offers significant speedups with the number of processors.

*This work has been funded in part by NSF PYI award MIPS-9058430, MIPS-94085550, CDA-9501637, and matching equipment grants from DEC, SYSTRAN, and IBM. An abbreviated version of this paper appeared in ACM Sigmetrics 1995 [SRF95].

Key Words:
Parallel simulation, performance evaluation, cache consistency, execution-driven simulation, shared memory multiprocessors, performance debugging.

1 Introduction

Shared memory multiprocessors usually have private caches associated with each processor. There are many parameters to be tuned with respect to the cache design such as the cache size, the line size, associativity, the replacement policy, and the protocol used for cache coherence. Thus cache simulations play a very important role in the design cycle of building shared memory multiprocessors by aiding the choice of appropriate parameter values for a specific cache protocol and estimating the performance of the system. Various simulation techniques including trace-driven [EK88, ASHH88], and execution-driven [Fuj83, CMM+88, DGH91] methods have been used for this purpose. Most of the known approaches to cache simulation are sequential. Such simulations impose a heavy burden on system resources both in terms of space and time. The elapsed time for the simulation is particularly limiting on the size of the system that can be simulated with realistic workloads. Given the availability of commercial multiprocessors, it is attractive to consider their use in reducing the elapsed time for the simulation by parallelizing the simulation itself. The expected benefit is in being able to simulate larger parallel systems (both number of processors and problem size) than can be feasibly simulated sequentially (due to space and time constraints), and the potential for speeding up the simulation compared to a sequential simulator.

Parallel simulation techniques are viable if they result in speedups as more processors are employed in the simulation. Typically these techniques fall into two categories - *conservative* and *optimistic*. In conservative techniques two parallel units of work can be scheduled at the same time if and only if one is guaranteed not to affect the execution of the other [Fuj90, CM79]. From the point of view of simulating cache-coherent multiprocessors, such a restriction invariably inhibits the simulation from being able to exploit the available parallelism in the application. On the other hand, optimistic scheduling techniques such as Time Warp [Jef85] could result in considerable state saving overhead that may dominate the execution.

We develop a new technique, called *timepatch*, that exploits the available parallelism in the application for driving the parallel simulation of caches for shared memory multiprocessors. The method is based on using application specific knowledge to yield a mechanism that is conservative with respect to generating a correct sequence of instruction execution (commensurate with the programming model), but is optimistic with respect to the timing information. Specifically, functional correctness of the simulated execution is ensured by executing the synchronization operations in the application faithfully as would be executed on the target parallel machine. The timing correctness of the simulation is accounted for at well-defined points in the application (such as synchronization

points). Thus the technique widens the window over which units of simulation work can be executed in parallel without having to synchronize with one another for timing correctness compared to conservative techniques. Since the technique never requires having to go back in simulated time for functional correctness (despite the optimism with respect to the timing information) the amount of state saving that is required for TimePatch is quite small compared to optimistic techniques. Further we use application specific knowledge (that can be gleaned from the compiler) such as the data layout in the caches to reduce the overhead of simulation. The result is a potential for significant speedup in simulation time for our technique which tracks the speedup inherent in the original parallel application.

The main contributions of this work are:

- a novel technique for parallel simulation of cache-coherent multiprocessors,
- development of performance enhancement strategies aimed at reducing the overheads of parallel simulation,
- a proof of concept prototype implementation that embodies the technique and the enhancements,
- performance results showing the improvement in performance with increasing number of processors.

In Section 2 we present the background and related work. We next state our assumptions and develop the technique in Section 3. An initial implementation and related issues are outlined in Section 4. Section 5 gives the corresponding validation and preliminary results. Using our initial implementation as a vehicle to analyze the overheads of parallel simulation, we then devise optimizations to the implementation of the timepatch technique in Section 6. Section 7 gives the validation and results with these optimizations incorporated into the simulator. Finally we present some concluding remarks and direction for future research in Section 8.

2 Related Work

Traditionally most cache studies have used either traces or synthetic workloads to drive the simulation. There are drawbacks to both of these approaches. Trace-driven simulation has some validity concerns (as observed by several researchers [FH92, GH93, Bit89]) owing to the distortions that may be introduced due to the instrumentation code that is inserted for collecting the traces. These distortions include non-uniform slowdown of the parallel processes due to varying amount of tracing code in each process, and overall slowdown in the execution speed of all processes owing to tracing.

Since the execution path of a parallel program depends on the ordering of the events in the program, both these distortions have the potential of completely changing the execution path unless timing dependencies are carefully eliminated from the traces [GH93]. Program startup effects may also distort the results, especially if the trace length is not long. Further, the traces obtained from one machine may not represent true interactions in another machine. Lastly, the traces usually do not capture OS related activities (such as interrupts, context switches, and I/O) unless hardware instrumentation is available [SA88]. Many synthetic workload models have been proposed that could avoid the problems associated with traces [AB86, Pop90]. Since application characteristics vary widely, it is difficult to generate a synthetic workload model that is representative of the memory access pattern across a wide range of applications. The drawbacks with both trace-driven and probabilistic simulation can be overcome with an execution-driven simulator. In this approach, real applications are used as the workload on the simulated target architecture. Thus the observed memory access pattern will be the actual one that will be seen on the target architecture. The primary disadvantage of this approach is that it is extremely slow since each instruction has to be simulated for the new architecture in question.

A simple modification to the execution-driven simulation technique has the potential for considerably reducing the simulation time in an execution-driven simulator. In cache simulations, we are interested only in the processor interaction with the cache/memory subsystem. Thus, we simulate only those events that are external to the processor, i.e., those that interact with the memory subsystem and we let the other "compute" instructions execute on the native hardware. In our case, the events of interest include loads, stores and synchronization. This technique of trapping only on "interesting" events saves considerable simulation time and has been used by others [DGH91, BDCW91, SSRV94, CMM+88]. This method, often referred to as "program augmentation", is certainly less expensive than execution-driven simulations in which every instruction is interpreted [Lig92]. An inherent assumption with this method is that instruction fetches do not affect the caching behavior of memory hierarchy. In spite of program augmentation, it may be infeasible (both in terms of space and time) to simulate large system and problem sizes with sequential simulation. Therefore, we explore methods of parallelizing the simulator in this research.

Synchronization of parallel simulators are often characterized as being *conservative* or *optimistic*. For example, in the conservative approach employed in the Wisconsin Wind Tunnel [RHL+92], only events that will not be causally affected by another event are processed in parallel. If t (called the *lookahead*), is the minimum time (for e.g. inter-processor communication time) required for one event to affect another event then we have a range of timestamps (from T to T+t, where T is the current lowest timestamp) that can be processed in parallel. Thus the parallelism is limited to the number of events that fall within this window of size t. Further, deadlocks are a potential problem with some conservative algorithms [CM79]. In the optimistic approach [Jef85], events are processed

as soon as they are generated, as though they are independent of others. Such optimism might result in incorrectness in simulating the actual behavior of the application. This situation (when causal violations in processing events are detected) is rectified by rolling back the prematurely executed event computation to a previous correct state, and re-executing the computation to preserve the causal dependencies. The optimistic approach requires preserving the necessary state information to restart computation in case of a possible rollback. This state saving could be a considerable overhead in simulating the cache behavior of shared memory parallel systems, though incremental state saving may alleviate this problem somewhat.

A parallel cache simulation scheme, based on a time-parallel simulation technique, using program traces has been proposed by Heidelberger and Stone [HS90]. A portion of the program trace is allocated to each processor. In this scheme, each processor assumes an initial state, and simulates its portion of the trace independent of the other processors. The statistics computed by each processor could be wrong due to an incorrect initial state. To address this situation, each processor gets its initial state from its logical predecessor and re-executes the simulation. This step may need to be repeated until the initial state of each processor matches the final state of its logical predecessor. The time-partitioning method is a fairly promising technique since typically there is just one repetition in a cache simulation because of the locality of references. However, it has the usual problem associated with trace-driven methods. In [HS90], it is also shown that it is sufficient to execute the traces for a set of cache lines instead of the entire cache. An implementation of a parallel trace-driven simulation on a MasPar is discussed in [NGLR92], which offers extensions to the above approach.

Dickens et al. [DHN94] suggest a technique for parallel simulation of message-passing programs. Their objective is to simulate the performance of these programs on a larger configuration of a target machine on a smaller host machine. The Wisconsin Wind Tunnel [RHL+92] uses a direct execution approach to simulate a shared memory multiprocessor on a message-passing machine. Using a portion of the memory at each node of CM-5 as a cache, WWT simulates a fine-grained version of shared virtual memory [LH89] through the ECC bits of the CM-5 memory system. The use of ECC bits allows WWT to avoid trapping on each memory operation compared to other execution-driven simulators. Thus only misses and access violations (which manifest as ECC errors in the WWT) are handled through special software trap handlers that simulate the target cache protocol. Using the minimum network latency Q as the *lag*, WWT implements a conservative parallel simulation technique requiring all processors to synchronize every Q cycles for processing the events generated in that window. Another interesting technique that completely eliminates the need for global synchrony is due to Muller et al. [MRSW]. In their approach, each processor times its actions independently. The trick is to artificially slow down the simulated execution of each component of the target system (commensurate with the path length of the simulated component

and its actual timing on the target machine), such that the resulting simulation reflects the balance of speeds of the target system. This approach is inherently parallel since it requires no global synchronization among the simulated entities. We explore a different technique for parallelizing the simulator; a technique that attempts to exploit the parallelism that is inherently present in the application.

3 The Timepatch Technique

The objective is to develop an execution-driven simulation platform that would enable gathering performance statistics (such as cache hit/miss rates, and message counts) of parallel programs executing on a *target* shared memory parallel machine. The target machine is simulated on a *host* which is also a shared memory parallel machine. Consider a shared memory parallel program with n threads to be executed on a n processor target machine, on which each thread is bound to a unique target processor. We use this program to drive our simulator on the host machine. We map each thread of the program to a separate physical processor of the host machine. Each host processor simulates the activity of the thread (mapped on to it) on the target processor. The simulation has to faithfully model the functional behavior of the original program as well as the timing behavior due to interprocessor communication and synchronization on the target machine. We assume a basic load/store type RISC architecture for the processors of the target machine where interprocessor interactions occur only due to memory reference instructions. We further assume that each target processor has a private cache which is maintained consistent using some cache consistency protocol. Thus the "interesting" events that have to be modeled for gathering the performance statistics of the memory hierarchy of the target machine are the load, store and synchronization operations. Only these interesting events trap into the simulator so that their behavior on the target machine can be modeled faithfully. Upon such traps, the simulator updates the state of the accessed cache block commensurate with the cache protocol implemented on the target machine and performs the intended operation (such as load/store of the data item). The "uninteresting" instructions in the thread are executed at the native speed of the host processor (i.e. they are not simulated at the instruction level). Instead, the time it takes to execute these instructions on the target machine is accounted for in the simulation.

As alluded to above, a simulation has to guarantee two kinds of correctness: *behavioral* and *timing*. A conservative simulation approach addresses both these correctness criteria simultaneously by never allowing event processing to get ahead of the permissible lag. Since no knowledge about the interaction between the events is available to the conservative simulator it must necessarily assume that any event can potentially affect another event outside the lag. In an execution-driven simulation of a shared memory parallel system it is possible to decouple the two correctness criteria

as discussed below.

To ensure behavioral correctness it is sufficient if we "simulate" the synchronization events in the parallel program correctly (i.e. as they would happen on the target machine) since these events in turn guarantee the correctness of the shared memory accesses governed by them. The correctness of the shared memory accesses governed by the synchronization events is guaranteed because of the following reason. Our host machine is also a shared memory multiprocessor and thus will reflect modifications to the shared state affected by the simulated parallel program. Simulating a synchronization event correctly implies ensuring that these events are executed in the same time order as in a conservative simulation, and providing a consistent view of the shared state of the parallel program to all the participating processors at such synchronization points. If the program uses implicit synchronization (i.e. it has data races), then behavioral correctness can still be ensured so long as such accesses can be recognized and flagged by the compiler.

The timing correctness criterion is achieved by a technique called *timepatch* which is described below. Each processor maintains its notion of simulated time which is advanced *locally*. The update of simulated time occurs due to one of two reasons. Firstly, upon executing a block of compute instructions on the host processor a call is made to the simulator to advance the time by the amount of time that block would have taken to execute on the target machine. Secondly, due to the traps into the simulator for the interesting instructions. The load/store events can be to either shared or private memory. These memory accesses may interfere with accesses from other processors of the target machine due to either true or false sharing on the target machine. However, each host processor simulates these load/store accesses as though they are non-interfering with other processors on the target machine and accounts only for the hit/miss timing. The potentially incorrect assumption that these accesses do not interfere with other processors may therefore result in the local notions of simulated time being inaccurate. Thus, at a synchronization point, these timing inconsistencies have to be fixed so that synchronization access is granted in our simulation to the processor that would actually be ahead in time on the target machine and therefore preserve behavioral correctness. Such timing inconsistencies can be corrected as follows. We maintain a history of all the memory accesses on a per processor basis. At synchronization points we merge these history logs to determine the ordering relationship between these accesses. Using this global ordering we can determine the inter-processor interactions (such as invalidation messages) that were previously not accounted and appropriately modify the notion of time of the corresponding processors. Details of how this reconciliation is done are presented in the next section.

The simulation technique outlined above is conservative with respect to the behavioral correctness since it faithfully executes the synchronization events in the parallel program in time order. However, it is optimistic with respect to timing correctness since it allows each processor to account for timing between synchronization points independent of other processors. The price for

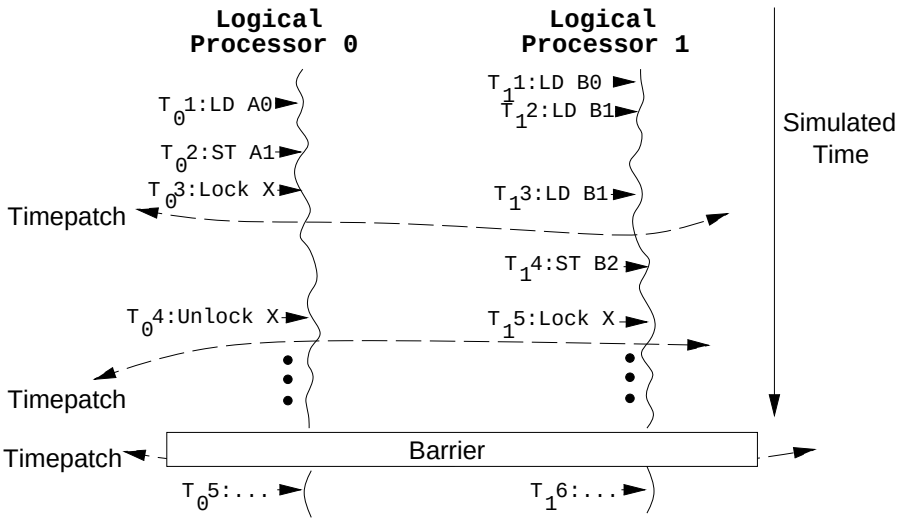

Figure 1: A Timepatch Example

this optimism is the time overhead in performing the timepatch at synchronization points and the space overhead for recording the history of accesses made by each processor.

Figure 1 demonstrates the timepatch technique with an example. Let us say that the application uses 2 logical processors. We consider only load, store and synchronization operations since those are sufficient to illustrate our idea. Consider the events at time T_02 on processor 0 and T_12 on processor 1. As can be seen from Figure 1 there is no order implied between these two events. In our method, each logical processor is bound to a physical processor and execution is not stalled until synchronization points. Thus in our simulator these events can be processed in parallel. However, if we assume that $A1$ and $B1$ reside on the same cache line and that an invalidation-based protocol is in effect on the target processor, the events T_02 and T_12 could indeed affect each other. Assume that T_02 (store to $A1$) occurs after T_12 (load of $B1$) but before T_13 (load of $B1$). In this case, this second load of $B1$ on processor 1 (event T_13) will be a miss due to the invalidation of that cache line by the store of $A1$ on processor 0 (event T_02). Observe that even though our hit/miss and timing information may be incorrect, the program's functional correctness is not violated because the shared memory is kept consistent by the underlying host processor. At synchronization points, the timepatch technique has to be applied and timing errors have to be fixed before granting these synchronization requests. Consider the events T_03 and T_15 that correspond to the lock requests on processors 0 and 1 respectively. Let us say that on the target processor, processor 0 gets the lock first. However in the simulation it is possible that processor 1 reaches the lock request before processor 0 does due to different speeds of the host processors. To ensure that the behavior on the target machine is accurately reproduced, timepatch is applied at T_15 and it is determined that processor 0 is lagging behind processor 1 in simulated time and that processor 0 could possibly

affect the outcome of this access. Based on this outcome, processor 1 is stalled until the simulated time of other processors (in this case only processor 0) advances to processor 1's current notion of time. Observe that while the timepatch operation is in progress only processor 1 is stalled waiting for other processors to cross time $T_1 5$ so that it can be granted synchronization. The other processors can continue processing the events of their respective threads. Similarly if processor 0 arrived at the event $T_0 3$ first it only needs to stall until processor 1 crosses $T_0 3$. A barrier is a special synchronization operation and after this operation the simulated time on all the processors participating in the barrier is the same; i.e., values of $T_0 5$ and $T_1 6$ are exactly the same.

The above example illustrates how the timepatch technique works. In the next section, we detail the initial implementation of this technique on the KSR-2.

4 Preliminary Implementation

4.1 Description

The target machine to be simulated is a CC-NUMA machine. Each node has a direct-mapped 64 KByte private cache. The shared memory implemented by the target machine is sequentially consistent using a Berkeley style invalidation based cache coherence protocol with a full mapped directory. We assume that hits in the simulated cache cost X cycles, that service from local memory costs Y cycles, and that service from remote memory costs P cycles. We also assume that invalidations cost Z cycles irrespective of the amount of sharing. X, Y, P, and Z thus parameterize the latency attributes of the network that are relevant from the point of view of consistency maintenance.

The above assumption ignores the contention that could result on the network due to remote accesses generated from the processors. As a result the execution time for a parallel application may be worse than what the simulator predicts. Further the performance statistics could also be affected by this assumption. The main motivation for not simulating the network activities in detail is the time overhead for simulating this aspect of the system architecture. The rationale for this assumption is as follows. In a well-balanced design of a parallel architecture one would expect that the network would be able to handle typical loads generated by applications. For example, experimental results on a state-of-the-art machine such as the KSR-2 have shown that the latencies for remote accesses do not vary significantly for a wide variety of network loads [RSRM93]. In [SSRV94], it was reported that the contention overheads observed in several applications were quite small. Recent studies [CKP+93] have also shown that parameterized models of the network may be adequate from the point of view of developing performance-conscious parallel programs. In any event, we do not expect the performance statistics gathered for the memory hierarchy to be affected significantly by ignoring contention since a high hit rate is expected in the private caches

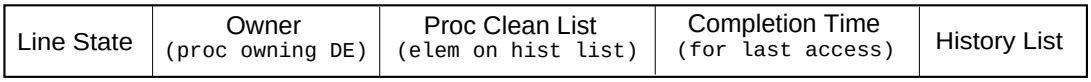

Figure 2: Structure of a Memory Directory Element (DE)

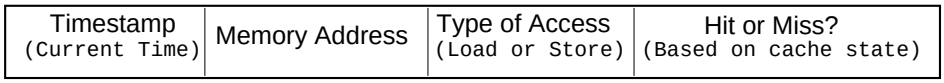

Figure 3: An Entry in the Patch Table

of a shared memory multiprocessor implying substantially small amount of network activity.

For the preliminary implementation, the host machine is KSR-2 [Res92]. KSR-2 is a COMA machine with a sequentially consistent memory model implemented using an invalidation-based cache coherence protocol. We map each node of the target machine on to a unique node on the host machine. The input to the simulator is a parallel program meant to be executed on the target machine. As mentioned earlier in Section 2, we use program augmentation technique to execute the "uninteresting" non-memory reference (compute) instructions at the speed of the native host processor. The input program is augmented (currently by hand at the source level) with traps into the simulator for every load, store and synchronization operation.

Typically, cache simulations maintain the directory state of memory, the state of each of the caches and a notion of (global) simulated time. The timepatch approach uses the following information in addition to the above. We maintain a *local* notion of simulated time associated with each processor of the target machine. Events executed on a processor only modify this local time. The timing information is correct in the absence of interactions with other processors.

To account for such interactions each processor maintains a *patch table* of time-stamped events (of all its memory operations). The simulator also maintains a memory directory data structure (See Figure 2). Each directory element (DE) in this data structure contains the usual information needed for protocol processing. The memory directory also maintains a "completion time" of the last access serviced by the memory directory. (The additional DE fields "Proc Clean List" and "History List" will be discussed later.)

On a trap corresponding to a load/store event, the cache directory state is updated and (hit/miss) time is accounted for based only on the current cache state. Traditional cache simulations also check the state of the memory directory in order to determine if further coherence actions (such as invalidations) are required. We do not perform this step at this point of access. Instead, the processor creates an entry (see Figure 3) in its patch table which gives the current (local simulated) time, the memory address accessed, the type of operation and an indication if the operation was treated as a hit or a miss. Notice that this operation does not require interaction

with any other processor and can proceed independently of other processors. Note also that the possible errors related to timing correctness are that we either treated a memory access as a cache access or that we did not account for the overhead of coherence maintenance. Consequently the local notion of time could be incorrect and possibly less than the "actual" value. For the compute instructions executed on the host processor, the local simulated time is updated based on the time it would have taken on the target processor.

As we mentioned in the previous section, to guarantee behavioral correctness it is necessary to ensure that the synchronization accesses in the parallel program be ordered faithfully in the simulation to correspond to their ordering on the target machine. Thus when a thread reaches a synchronization point in its execution it is necessary to reconcile all the per-processor local notions of simulated times to derive a correct ordering for the synchronization access. This reconciliation involves resolving inter-processor interferences that may have happened due to the load/store memory references that this thread generated up to the synchronization point.

The "timepatch" function (see Figure 4) is called by a processor to perform this reconciliation. A counter $addtime_j$ is associated with each processor j, in which the correction that needs to be applied to the local notion of simulated time is recorded by the timepatch function. Since all memory accesses have to go through the memory directory, it is the central point of conflict resolution for competing memory references from different processors. The timepatch function determines the timing correction that needs to be made to the accesses from each processor based on the interactions between accesses from different processors. This determination is done by applying these accesses to the memory directory when timepatch is called. First the accesses of the processors recorded in the respective timepatch tables have to be ordered before they can be applied to the memory directory. Since each processor timestamps its accesses in order, the local timestamp tables are already sorted in increasing time order. The timepatch routine has to merge all the local timepatch tables, apply them in order to the appropriate memory directory entries, determine the possible interactions among the accesses, and reconcile the ensuing timing inconsistencies. Before we start applying these accesses to the memory directory, we find the timestamp t_j of the last entry in each of the per-processor timestamp tables. We next determine the minimum value Min of the t_j's. Min represents a time up to which all processors have advanced in simulated time. Therefore all the accesses from the processors up to this point of time will be in the timepatch tables, and the potential interaction among these accesses can be resolved. We cannot handle events with timestamps greater than Min at this point because it is possible that the processor associated with the current minimum Min (the one lagging behind in simulated time) may make accesses that could cause interactions with other accesses that happen in its "future". The first unprocessed entry is picked from each of the local timestamp tables and inserted into a sorted tree if its timestamp is less than Min. Note that there is *at most* one entry per processor at a time in

this tree. The record x_j with the minimum timestamp is deleted from the tree and the memory directory element corresponding to this operation is updated. This update will also determine the increment (if any) that has to be applied to the counter $addtime_j$ of processor j. It should be clear that this increment can never be negative since we process the events in time order. There will be a non-zero increment when the interval between consecutive accesses to the same memory block is less than the servicing time for that access, and represents the queueing delay at the memory directory as well as the cost of consistency maintenance which were overlooked when the access was made. The timestamp of any entry accessed henceforth from the timestamp table corresponding to processor j is incremented by the value in $addtime_j$. If the increment has to be applied to the processor that determined Min originally, then Min will have to be recomputed. The next entry (if any remain) from this processor's timepatch table is inserted into the tree if it is less than the new Min. The above processing of entries from the tree continues until the tree is empty.

Upon return from the timepatch routine, all inter-processor interferences have been correctly accounted for up to time Min. If the processor that invoked this routine (upon reaching a synchronization point in its execution) still finds itself to be the least in simulated time then it can perform the synchronization access. If not, it would have to block until all the other processors have caught up with it in simulated time to make this determination. It should be noted that while a processor is performing timepatch, other processors can continue with their execution.

We will use the same example used in the previous section (Figure 1) to illustrate in detail how timepatch works. As before, we assume that $A1$ and $B1$ are located on the same cache line. The time order of the events shown in Figure 1 captures the order in which they should occur on the target machine. Again, we assume that host processor 1 executes faster than host processor 0, and gets to T_15. Suppose that the only entry in the table associated with processor 0 at this point is T_01. In the timepatch routine if we process T_13 then we would not be able to consider the effect of T_02 on T_13 since T_02 has not yet occurred. Thus in this case timepatch should only consider events up to time T_01. In the meanwhile, processor 1 has to stall until processor 0's time is at least T_15. Eventually processor reaches T_03 and timepatch is invoked again because of the synchronization operation. Now both processors 0 and 1 are waiting for the synchronization access. The timepatch routine determines that T_03 is lesser than T_15 and grants synchronization access to processor 0. Observe that this is the desired behavior on the target machine.

There are certain subtle cases arising due to the semantics of synchronization events which the implementation has to handle. Barrier is one example. In this case, the table entries of all the processors participating in the barrier have to be cleared when the last processor arrives at the barrier irrespective of Min. The synchronization traps into the simulator correctly handles such cases.

```
Begin Mutex /* only one processor can perform this action at a given instant */
    (1) for each processor j initialize a counter addtime_j to zero
        /* this keeps track of the correction that needs to be applied
        * to the local notion of that processor's time
        */
    (2) determine t_j, the timestamp of the last entry in each of the
        per-processor tables, and find the minimum Min among these.
        /* timepatch can be performed only up to this time Min */
    (3) pick the first unprocessed event x_j from each per-processor table
    (4) for each x_j if ( x_j.timestamp < Min )
        (4:1) insert x_j into tree sorted by the timestamp x_j.timestamp
        end for /* end of (4) */
    (5) while (tree not empty)
        (5:1) delete a node lt with lowest timestamp from the tree
            (say it belongs to processor j - then lt = x_j )
        (5.2) apply the access of lt to the corresponding memory directory
            and set flag if timing inconsistency detected
            /* this will update the state of the memory directory entry
            * and the timestamp associated with this memory directory
            */
        (5:3) if ( flag is set ) /* implies timing inconsistency */
            (5:3:1) increment the counter addtime_j appropriately
            (5:3:2) Recompute Min
                end if /* end of (5:3) */
        (5:4) pick the next unprocessed entry x_j for processor j
        (5:5) increment x_j.timestamp by addtime_j
        (5:6) if ( x_j.timestamp < Min )
            (5:6:1) insert x_j into the tree
            end if /* end of (5:6) */
            /* continue processing until tree is empty */
        end while /* end of (5) */
End Mutex
```

Figure 4: Pseudo-code invoked to patch time

4.2 Practical Considerations

While timepatching is strictly required only at synchronization points for the correctness of the proposed technique, we are forced to do it more often due to space constraints since it is infeasible to maintain very large tables for the timepatch entries. The main idea behind timepatch is to increase the window of time over which it is possible to have parallel execution of the simulated threads of the target architecture. This window is reduced somewhat due to performing timepatch more frequently, but it has the beneficial side effect of advancing the global simulated time, deleting some of the entries from the local timestamp tables, and thus clearing up resources that can be reused.

5 Preliminary Results

The are two aspects to be evaluated to appreciate the merit of our technique. The first is validation of the technique itself to ensure that the performance statistics of the target architecture obtained using our technique are correct. The second is the speedup that is obtained from the parallel simulator. Ideally, one would obtain the speedup curve of the parallel simulator to track the speedup achievable in the original application. However, this depends on how the overheads in the simulation itself get apportioned among the participating processors. We first address the validation question.

5.1 Validation

We developed a sequential simulator that models the same target machine using CSIM [Sch90], which runs on a SPARC workstation. In order to validate our parallel simulator we used randomly generated traces to drive the sequential and parallel simulators. The performance statistics used to verify the parallel simulator are the simulated cycles (for performing all the memory references in the traces) and the message counts (number of messages generated due to write invalidation). The validation results are shown in Table 1. The traces consist of only load and store references and different proportions of read to write ratios. One of the traces shown has 5000 references and the other has 10000 references. The message counts for the two cases are off by at most 2% for the traces considered. The agreement in simulated times is within 2% for one trace and varies between 7% and 12% for the other trace. CSIM is a process oriented simulation package, which schedules the CSIM-processes in a non-preemptive fashion within a single Unix process. We do not have any control over the internal scheduling policy that CSIM uses for scheduling the runnable CSIM-processes. We believe that the differences observed for the second trace are due to the effects of this scheduling policy. Overall the validation results indicate that our parallel simulator simulates

Application	Processors	Simulated Cycles			Message Counts		
		Sequential	Parallel	Difference	Sequential	Parallel	Difference
Trace-5K	4	671490	670395	0.16%	6045	6040	0.08%
Trace-5K	8	790010	788600	0.18%	13055	13039	0.12%
Trace-5K	16	868520	883835	-1.76%	27095	27087	0.02%
Trace-10K	4	1519895	1703035	-12.04%	11151	11377	-2.02%
Trace-10K	8	1660345	1781825	-7.32%	27765	27881	-0.41%
Trace-10K	16	1757455	1895990	-7.88%	59980	60010	-0.05%

Table 1: Validation of the Parallel Simulator

Processors	Speedup
1	1
2	1.26
4	1.39
8	1.41

Table 2: Speedup for 64x64 Matrix Multiplication

the target machine with reasonable accuracy.

5.2 Speedup Result

Next we address the speedup question. In order to illustrate and understand the performance of our preliminary implementation, we use the matrix multiplication program.

The matrix multiplication problem can be easily parallelized without any false sharing (except for boundary conditions). True sharing is only for read-shared data and thus there is no synchronization in the code. As expected, the raw code we developed for the matrix multiplication program shows linear speedup on KSR-2. Thus we expect that the parallel simulator which uses the same code with augmentation will track the speedup of the application program and give similar speedups. Consider the performance of our simulator when we use 64x64 matrix multiplication to drive our simulation. These results are shown in Table 2.

As seen from the Table the performance of the simulator falls considerably short of the expectations. This is because of the overheads in the simulation. In the next section we analyze what these overheads are and suggest implementation techniques that can be used to reduce these overheads

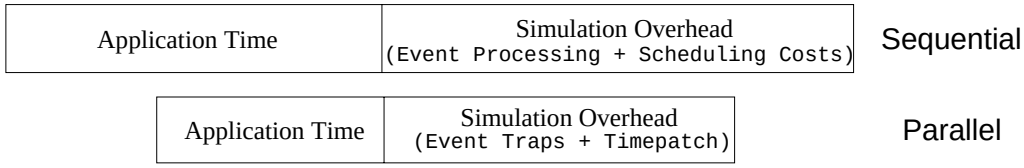

Figure 5: Expected gains using parallel simulation

and thus achieve the speedup potential of the timepatch technique.

6 The Anatomy of the Parallel Simulator

While performing an execution-driven simulation there are two costs involved - that of the actual application and the simulation overhead (see Figure 5). Let us consider how these costs vary as we simulate different numbers of target processors. For a given problem size, the amount of processor cycles used by the application (assuming a deterministic computation) in a sequential setting is almost a constant since the problem that has to be executed has not changed, irrespective of the number of processors simulated. However, simulating more processors adds to the simulation overhead in the sequential setting because of additional process management and other effects such as decreased cache locality. Now let us consider our parallel simulator. The application time is unaffected by our simulation methodology and hence the available parallelism in the application should be observed in our simulation method as well. The real question is how the simulation overhead is apportioned among the processors when we simulate larger numbers of target processors. The simulation overhead can be broken down into two components: (i) time-stamping and local state maintenance on each memory access, and (ii) timepatching at synchronization points. The first component includes the traps incurred for loads and stores. Since these traps and ensuing state maintenance activities are local to each processor, they are done in parallel without interfering with other processors. Therefore assuming that the total number of loads and stores remains a constant for a given problem size the overhead due to the first component also should exhibit the same speedup properties as the application itself, and thus should not limit the speedup of the simulation. The second component of the simulation overhead includes fixing timing errors as well as waits incurred due to local timepatch tables filling up. In the preliminary implementation the timepatch function is performed only by one processor at a given instant and is the source of the *sequential* bottleneck in the simulation.

To investigate the cause for the relatively poor performance of the simulation we profiled our simulator in order to understand where the time is spent. Figure 6 shows the per-processor breakdown of the various costs involved for different numbers of processors. In the preliminary implementation,

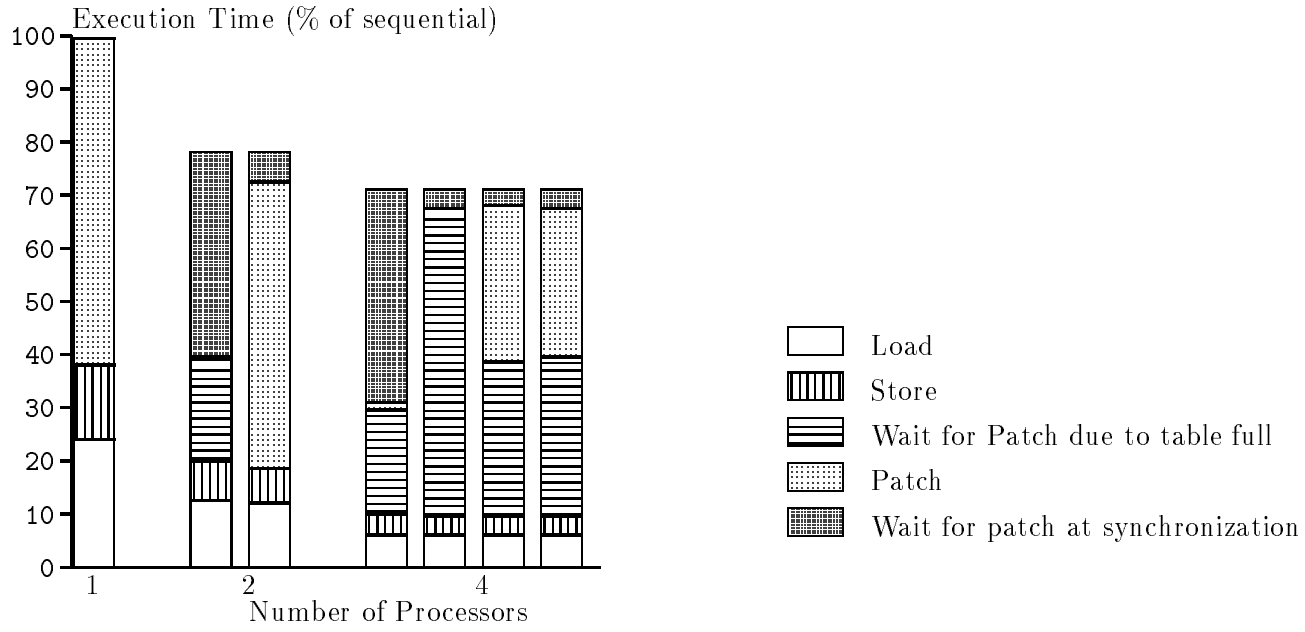

Figure 6: Comparative costs in the 64x64 Matrix Multiplication

the overheads dominate to the extent that the compute time in the application is negligible. For this reason, the compute time is not shown in the figure. The breakdown gives the time to perform the load traps (labeled *Load* in Figure 6), the store traps (labeled *Store*), the time to perform the timepatch operation (labeled *Patch*), the wait time experienced because the timestamp table is full (*Wait for patch at table full*), and the wait time for the timepatch operation to complete at synchronization (*Wait for patch at synchronization*). The load and store traps are application dependent and are inevitable. The costs associated with performing timepatch and the waits are the simulation overheads of our technique and our aim is to make them as small as possible. From Figure 6, we see that the costs for load/store traps decrease proportionally as we increase the number of processors since this work gets divided among the processors. In the 4 processor case we observe that the wait times due to the timestamp tables becoming full is a significant portion. This preliminary implementation uses a table size of 5K entries per processor. Thus the tables get filled up quite quickly. This forces a processor to attempt to perform a timepatch operation. But if some other processor is already in timepatch then this processor has to wait until some space is freed up in its local table. So reducing the wait time at timepatch is an important step for speeding up the simulation. Another observation from the chart is that the cost to perform the timepatch operation is a significant portion of the total time. Thus another avenue for speeding up the simulation is to speed up the execution of the timepatch operation. The wait time for patch at synchronization is a subtle issue. If the application has little serial overhead and no work-imbalance, then there should not be any significant waiting at synchronization. The waiting that we do see in Figure 6 is brought

about by a processor having to wait at a barrier for some other processor to finish timepatching. This waiting time will shrink if we reduce the patch overhead. However, as we will see shortly, for a fixed problem size this waiting time becomes the limiting factor to performance. We overcome this obstacle by parallelizing the timepatch technique (Section 6.3).

In the following subsections, we explore methods to cut down these overheads.

6.1 Reducing the Wait Time at Patch

The processor that is ahead in simulated time will eventually be blocked at a synchronization access. Therefore, one possibility to reduce the wait times at timepatch is to make this processor perform the timepatch operation. Clearly, the total execution time for the simulation cannot be less than the cumulative patch time since timepatch is essentially a sequential process in the first-cut implementation. However, it can be seen from Figure 6 that the total sum of the patch times observed on all the processors is close to the execution time. Therefore, throttling the fastest processor is not going to help. On the other hand, if the size of the timepatch tables are increased then that would reduce the number of trips to timepatch that each processor would have to make and therefore reduce the amount of waiting that each processor would have to do at timepatch. We observed that with a table size of 500K entries, the waiting time at patch approached zero. This reduction occurred without increasing the overall work to be done by the timepatch function.

6.2 Optimizations to Reduce the Patch Time

Given that timepatch is the dominant source of overhead we turn our attention to seeing if that work itself can be reduced. It is instructive to see the number of timepatch entries that are created and the actual number of them that impact the simulated time. For example, in matrix multiply with 16 processors, 798,916 timepatch entries are created but only 61,742 entries (less than 7%) actually result in change to the simulated time due to the inter-processor interferences. It turns out that it is possible for the compiler to predict which references will result in interferences from the data layout the compiler employs for the shared data structures in the caches. Therefore, if this knowledge is available to the simulator then it can eliminate a sizable number of the entries from the timepatch table. We did this optimization for the applications that are considered here. There are three kinds of data: private, true-shared, and false-shared. It is possible for the compiler to distinguish these categories of data, and this information can be made available to the simulator. The simulator should create patch entries for all false-shared data since they could cause interference depending on program dynamics. The simulator does not have to create patch entries for private data since the data layout of the compiler ensures no interference with other processors. The true shared data gets shared across synchronization regions in the program. Thus within a synchronization region

Processors	Improvement over Unoptimized	Speedup (for optimized)
1	11%	1
2	49%	2.2
4	56%	2.8
8	66%	3.7

Table 3: Comparing the optimized and unoptimized Matrix Multiplication Simulation

it is sufficient if there is at least one patch entry for the first access for each such variable in that region by a processor.

We modified our implementation to use these optimizations. Associated with each cache block is a timestamp that gives the last access to that cache block. We associate a *sync-time* with each processor. This is the timestamp of the last synchronization operation carried out by this processor. The augmenter which generates the load/store traps gives this additional information whether the access is to private, false-shared, or true-share data. Upon every true shared access the timestamp in the referenced cache block is checked against the sync-time for that processor. If the sync-time is greater then an entry is created for this reference in the patch table. An entry is created for every false-shared access and none for private.

With these optimizations the number of timepatch entries that are actually processed during the entire simulation is down from 798,916 for 16-processor matrix multiply to 267,636 (by a factor of 3). This is still around a factor 4 more than the number of entries that actually cause timing errors. By a more careful analysis and labeling of the accesses we may be able to bring down this number even further.

The cache statistics (such as message counts and hit rate) observed for the optimized and unoptimized versions were the same indicating that these optimizations are indeed correct. Since the objective here is to show the viability of our simulation technique these performance statistics are not germane to the rest of the discussion.

Table 3 shows the percentage reduction execution times and speedups for matrix multiply with these optimizations. As can be seen, we do get a reasonable speedup up to 8 processors for matrix multiply (3.7 for 8 processors). We also see that the optimizations give a significant payoff (ranging from 11% to 66%). Since both versions use the same table sizes and periodicity for timepatch, the reduction in time is entirely due to these optimizations.

6.3 Parallelizing the Patch Work

However, despite all of these optimizations, the simulator did not show speedup beyond 8 processors for matrix multiply, and beyond 4 processors for another application, integer sort (IS) from the NAS benchmark suite. The reason is simply Amdahl's Law. In the preliminary implementation, the patch operation is performed sequentially. While the computation overhead of patch does not grow linearly with the problem size (due to the optimizations discussed in Section 6.2), it does become the limiting factor to speedup for a fixed problem size per Amdahl's Law. As a result, as the number of processors increases most of them are waiting at synchronization points for one processor to complete patch. Primarily to eliminate these simulation-induced synchronization stalls, we turn our attention next to parallelizing the patch work. A secondary benefit is that the overhead of patching gets distributed more evenly across all processors as compared to the preliminary implementation.

The basic idea is that the processors on the host machine alternate as a group between two parallel phases: the *computation* phase and the *patch* phase. The computation phase represents the original parallel program execution on the target architecture, together with the creation of the patch table entries towards time reconciliation (as we discussed previously). The patch phase represents the overhead work to be carried out by the host machine to reinforce timing correctness for the parallel simulation. The patch phase can be initiated by any processor due to (a) its patch table becoming full, or (b) arrival at a sync operation in the parallel program. At this point, all of the processors switch to the patch phase and reconcile their patch table entries up to *Min*, a time up to which all processors have advanced in simulated time (see Section 4.1).

There are two primary approaches to parallelizing the patch function. The first method treats the available processors as a pool of computing resources used to collectively work on the time-ordered global list of patch table entries. One of the attractive features of this option is that the patch work is inherently evenly balanced across all processors. Another benefit is that the ordered list ensures that all patch table entries are applied in the proper global order. Unfortunately this method has three main drawbacks: first, the ordered list becomes a major bottleneck[1]. Second, one must still ensure that, once the patch table entries are out of the list and in the custody of the processors, that the processors still process them in a global time-ordered manner (without degenerating to simple sequential processing). Finally, the list order must be maintained throughout the patch operation. To the extent that processing each patch table entry does not interfere with the processing of the others, this order maintenance represents overhead (based on the number of entries, rather than on the number of processors) that is unnecessary.

The second approach to parallelizing the patch function is more optimistic: each processor

[1] A distributed list scheme can be used, but this would involve much more additional complexity.

processes its own patch table entries and relies on the other processors to inform it if any of its past processing was in incorrect time order with respect to the associated memory directory element (DE) for that memory block. If the processor receives such notifications, it "rolls-back" these DE actions and retries them. One may notice that this scheme is similar to other roll-back methods (e.g. Time Warp [Jef85]). It has, however, the advantage in that far less state history is required. Since Timepatch separates behavioral from timing correctness, only time-related history information must be maintained between patch operations. Unlike the previous ordered-list option, the optimistic scheme can result in an unbalanced load across processors, due to one or more processors having a greater number of patch table entries or roll-backs than the remaining processors. The distribution of patch table entries to processors is dictated by the work distribution in the application program, and thus reflects the load imbalance inherent in the application program. Since a large class of parallel programs belong to the SPMD variety, we expect the work distribution to be fairly uniform. The overhead due to roll-backs is not as controllable, however, and could possibly dominate the patch time with large numbers of processors[2]. Due to its relative simplicity and to the benefits associated with the optimistic approach, we chose to implement this second approach.

6.3.1 Parallel Patch Implementation Details

Figure 8 shows the high-level algorithm driving our parallel patch. Aside from updating the directory element state and detecting the patch termination condition, there are four major mechanisms at work: 1) patch initiation, 2) patch operation, 3) out-of-order DE action detection, and 4) roll-back.

Patch Initiation

As we mentioned earlier, a processor needs to reconcile timing differences with its peers when it hits a synchronization point in the application program. This is accomplished by initiating parallel patch. One method that could be used to summon the other processors to patch is for the initiating processor to send them inter-processor interrupts. However, since such interrupts invariably require kernel intervention in most systems we decided to use a polling approach that works as follows. The initiating processor sets the "patch request" flag. This flag is polled by all the processors at every shared load, shared store, and synchronization operation in the program; basically, the check is made every time a processor slips into the simulator from the application program to record events that can affect the logical times of the processors.

While being very efficient, this strategy in which each processor polls to see if it needs to enter patch phase requires careful handling of potential race conditions and deadlocks. One such deadlock

[2]In the current implementation, however, we have not noticed this trend.

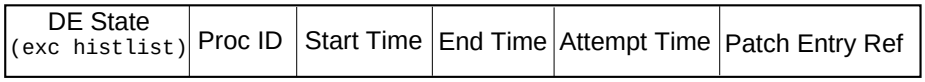

Figure 7: Structure of a History Element

situation arises when a processor is blocked at an application level lock or barrier. In this case, this processor cannot respond to a patch request from a peer processor since it will not poll the request flag until it gets unblocked from the synchronization operation. Therefore, immediately before attempting a potential blocking call a processor should "remove" itself, from the set of processors that must participate in the patch phase. Similarly, immediately after the processor succeeds (or gets unblocked) in performing the synchronization operation it should "add" itself to the list of processors participating in the patch phase. This procedure is accomplished as follows: At the end of the patch phase, if a processor determines that its patch table is empty and that it is about to attempt a synchronization operation then it atomically decrements the *participation counter*. This counter determines the number of processors that will participate in the next patch phase. Upon successful completion of the synchronization operation, when the processor is ready to rejoin in a future patch phase, it atomically increments this counter. To avoid potential data races in which one processor increments this counter while another initiates a patch operation, the patch initiation is controlled by the same mutex lock as this counter. Another consideration is in determining Min – the logical time up to which the participating processors reconcile their patch table entries. The non-participating processors are guaranteed not to generate any events that can affect the logical time of the other processors. Therefore, a processor stores its logical time when it decides to remove itself from the participating set. This stored value is then used in Min computation by other processors until the processor rejoins the set.

Patch Operation

Most of the time used in patching is for patch operation that does not involve roll-back (detailed below). This mechanism applies the memory accesses reflected in the patch table entries to their respective DEs. First the processor acquires a lock on the DE, then checks to ensure that the patch table entry can be applied at that time (i.e. if the current patch table entry's timestamp is greater than the completion time of the past application on the DE). If the entry can be applied, a "history element" (see Figure 7) is placed in time-order on the "history list" of the DE (Figure 2). This history element records the following: the start time, the end time, and the "attempt time" of the patch processing for each patch table entry; a reference to the ID of the processor applying this patch table entry to the DE; a reference to the patch table entry itself; and the state of the DE prior to the change. The "attempt time" is the processor's local notion of the simulated time at

which it attempts to apply this patch table entry to the DE. Note that in contrast to this, the start time is the earliest time that the processor can apply the patch table entry to the DE. Therefore the attempt time and the start time may not always be equal. The attempt time is useful in out-of-order detection, as will become evident shortly.

This parallel application of the patch table entries to the DEs is expected to result in good performance, since we don't expect much contention for the DE locks for most scalable parallel programs.

Out-of-order Action Detection

To allow each processor to detect out-of-order actions on a directory element, the processing of each patch table element results in a history element being placed on the history list of the associated DE (regardless of whether the DE's state is then modified or not). In addition to storing the state of the DE prior to the application of the required changes, the history element also contains the ID of the processor making the change, the start and end times of the DE action, a reference to the patch table element, and the first attempt time for that DE action.

Two situations are used to determine if previous actions on a DE need to be rolled-back:

1. If a history element corresponding to the current patch table entry is encountered, then all DE actions up to and including that one must be rolled-back. This situation occurs when the roll-back of an action on one DE implicitly rolls-back subsequent actions on other DEs.

2. If:

   - the attempt time of the last DE action is greater than the timestamp of the current patch table entry, or
   - if the last DE action's attempt time is equal to the timestamp of the current patch table entry, but this processor had ownership of the DE while the processor that performed the last DE action did not,

   then the previous DE action must be rolled-back.

Roll-Back

When a processor wishes to roll-back a past DE action, it first removes the corresponding history element and restores the original DE state. If the offending DE action was performed by the same processor, then the history element is simply freed. (This corresponds to roll-backs under case 1 above.) If the DE action was performed by another processor, then this other processor must be made to redo the action at a later time. We considered two implementation options at this

```
StartPatch:  /* called by processor wishing to cause a patch phase*/
Begin Mutex /* only one processor may perform init and cleanup for patch */
    (1) set ``patch request flag'', and perform initialization
        /* Processors not blocked on an application program lock */
        /* or barrier will detect the patch request at load, store */
        /* or synch points, and will enter patch phase at PatchMain*/
    (2) enter PatchMain to begin patching with other processors
    (3) perform various patch-related cleanup operations
        /* Does not include cleanup of history elements */
    (4) ***barrier*** /* matches (5) for other processors in PatchMain*/
End Mutex
/* End StartPatch */

PatchMain:  /* performed by participating processors in parallel */
    (1) ***barrier***
    (2) while ( all participating processors not done with patch )
        (2:1) if ( mailbox contains roll-back messages ) AND
                 ( the earliest of the associated patch table entries
                 is earlier than current )
              (2:1:1) reset ``current patch table pointer'' to earliest
                      rolled-back entry
              end if /* end of (2:1) */
        (2:2) if ( patch table entries present in table that do not
                 exceed minimum global time )
              (2:2:1) indicate ``processor not done with patch''
              (2:2:2) lock DE
              (2:2:3) with ( earliest patch entry yet to be patched )
                      (2:2:2:1) for associated directory element,
                                roll-back all actions (removing
                                associated history elements from de's
                                list) that happen later than patch entry
                                /* if roll-back is for other processors,*/
                                /* put message in the processors' mail */
                                /* boxes */
                      (2:2:2:2) perform DE action (creating history
                                element) if it can be done at this time,
                                and set addtime appropriately
                      end with /* end of (2:2:2) */
              (2:2:4) unlock DE else
              (2:2:5) indicate ``processor done with patch''
              end if-else /* end of (2:2) */
        end while /* end of (2) */
    (3) if ( processor is at application lock or barrier) AND
             ( all of its patch table entries are patched)
        (3:1) lock patch participation counter lock
        (3:2) decrement patch participation counter
        (3:3) store processor's current logical time
        (3:4) unlock patch participation counter lock
        end if /* end of (3) */
    (4) clean up history elements from directory element lists
    (5) ***barrier*** (for all processors except the one that initiated patch)
/* End PatchMain */
```


Figure 8: High-Level Parallel Patch Algorithm

point. The first option is to have a processor effect a roll-back by directly modifying the offending processor's patch table pointer and local time increment variables. This method would require either extensive locking and unlocking of these two variables, or a sacrifice of parallelism, and in general is more difficult to control. The second option is to put the history element in the offending processor's roll-back "mailbox". After each processor processes a patch table entry, it checks its mailbox for roll-back messages. If they exist, then the processor handles them accordingly (also freeing the history elements). Note that this option permits the processor sending the roll-back message to continue without waiting for action from the message's recipient.

Note that at the end of an entire patch phase each DE involved will have a history list containing history elements for all actions performed on that DE during the patch phase. Since these history elements are only relevant within that single patch phase, the history lists must be "cleaned" (all history elements removed and freed). Each processor does this by scanning back through its patch table entries, and removing all of its history elements from the associated DEs. Adding an additional per-processor "Proc Clean List" mask (Figure 2) to each DE structure, indicating if history elements are present for that processor on that DE's history list, speeds up cleanup considerably. Although we originally were concerned that these lists might grow so long as to cause substantial overhead at the end of the patch phase (and thus require some sort of garbage collection step within the patch process), this has not been the case.

7 Parallel Patch Results and Observations

In this section we give the results of the Timepatch technique with all of the optimizations discussed in Section 6. We consider two application programs: Matrix Multiply (Matmul) and Integer Sort (IS) from the NAS benchmark. During the course of this research work, our KSR-2 met its demise. Therefore, we have ported our code to a 12-processor (R8000) SGI PowerChallenge shared memory multiprocessor. The results presented are for runs on this machine. The simulated CC-NUMA machine has a 64K byte cache with a 16-byte blocksize. Note that we are not comparing absolute times of the implementations on the two different machines, only the speedup results.

As shown in Figures 9 and 11, the simulation speedup curves for both Matmul and IS closely parallel the respective speedup curves of the applications. This is in agreement with our hypothesis that the timepatch technique should track the parallelism in the application. Note that the speedup for IS is not as good beyond 4 processors purely because we chose a fairly small data size (64K elements, and 2K keys) to keep the run times manageable. The algorithm we used for IS has been shown to scale quite well for larger problem sizes on parallel machines such as the KSR-1 [RSRM93]. Figures 10 and 12 show the breakdown of the simulation overheads for Matmul and IS respectively. Each bar in these Figures gives the following breakdown:

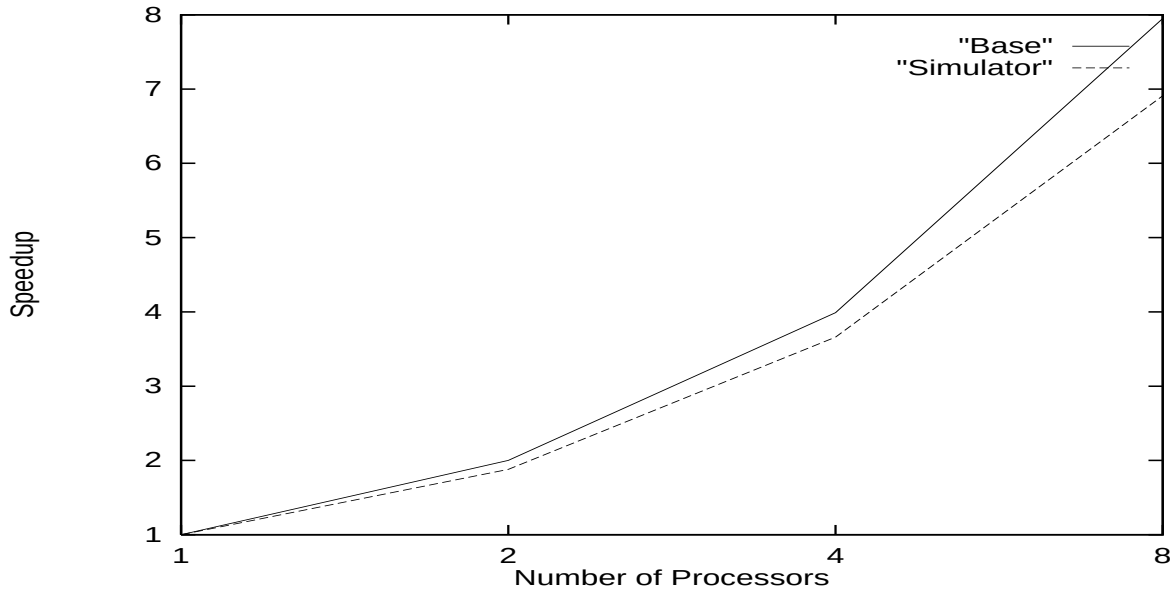

Figure 9: 64x64 Matrix Multiply: Speedup Comparison

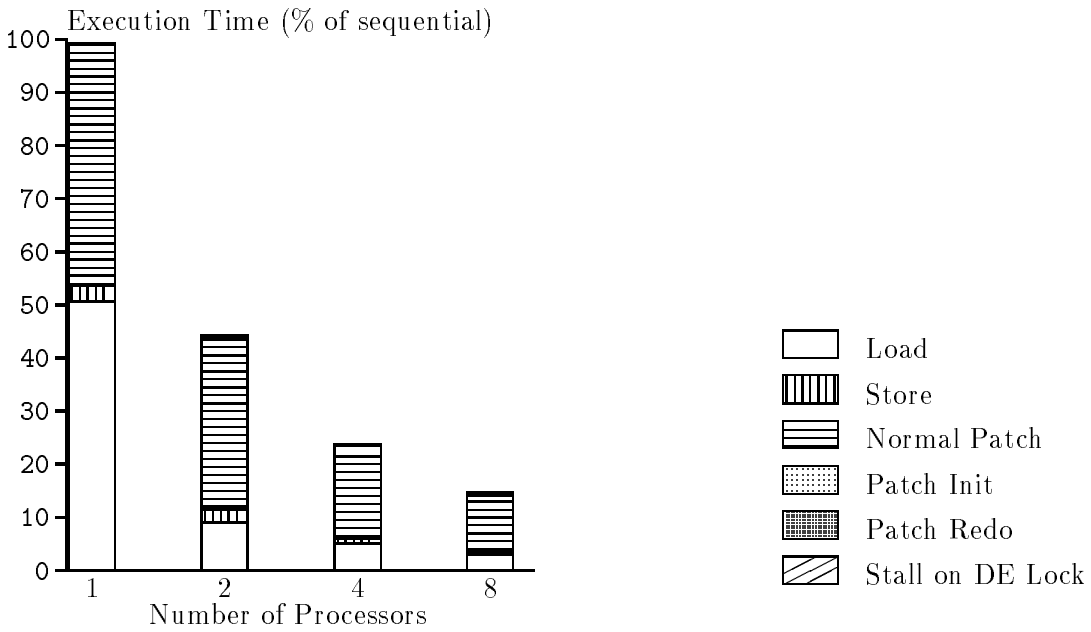

Figure 10: Comparative costs in the 64x64 Matrix Multiplication

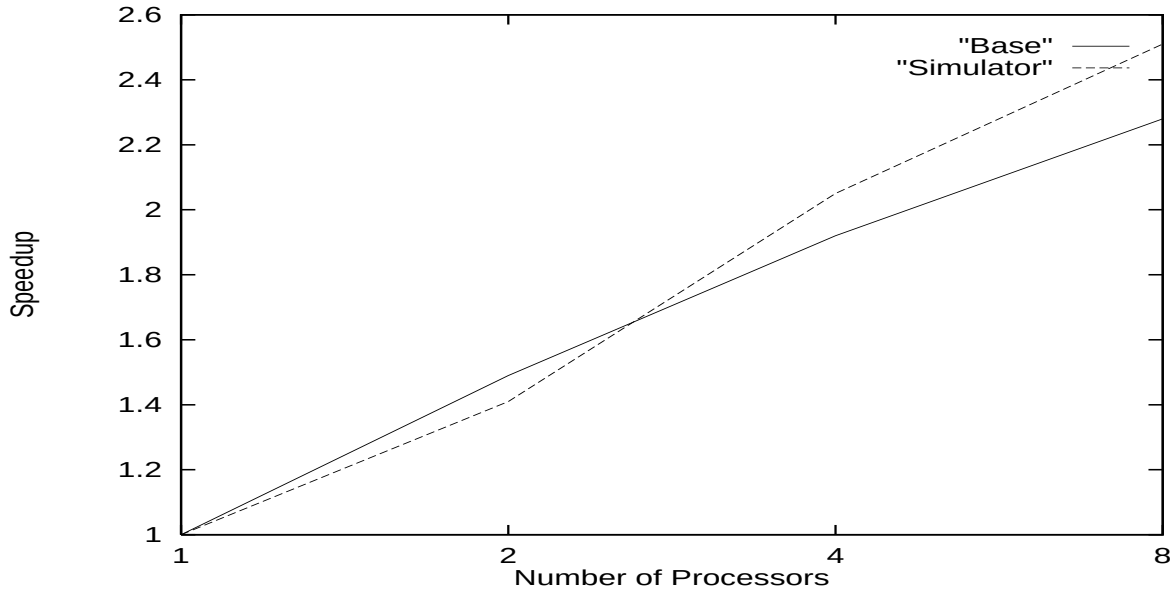

Figure 11: 64K, 2K-key Integer Sort: Speedup Comparison

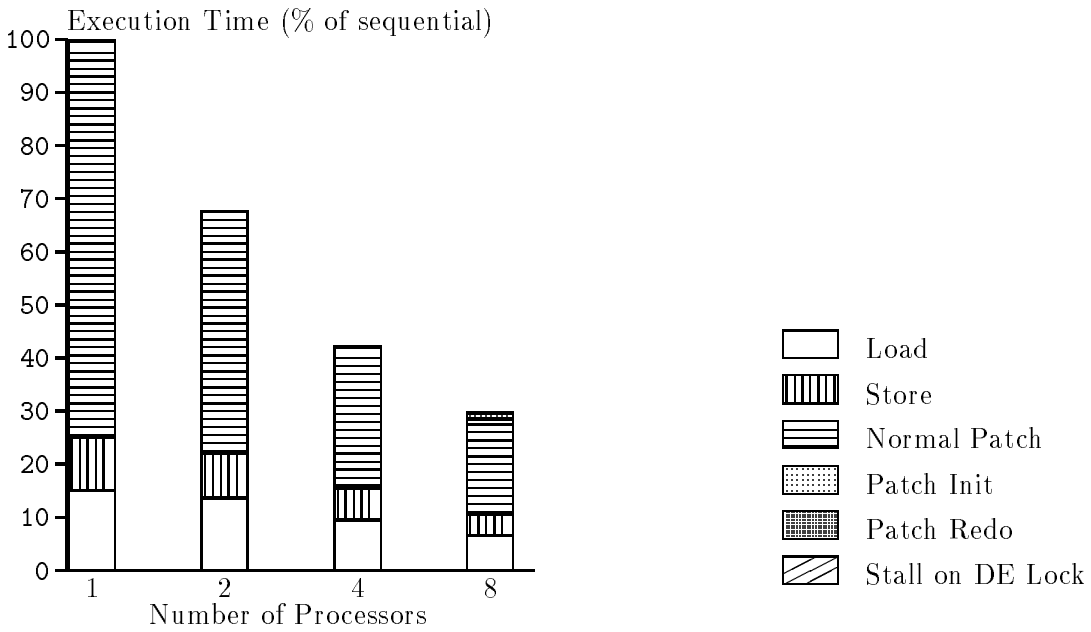

Figure 12: Comparative costs in the 64K, 2K-key Integer Sort

- *Load* and *Store* are the traps into the simulator for loads and stores respectively;
- *Patch init* accounts for time to get the participating processors into the patch phase from the point of initiating the patch;
- *Normal Patch* is the time spent by each processor in the absence of any rollback induced redo's;
- *Stall on DE lock* accounts for any contention encountered for accessing the directory elements during the patch operation; and
- *Redo* accounts for redoing any patch table entry due to rollbacks.

As can be seen in Figures 10 and 12, the stall-on-DE-lock is negligible for both the applications showing that in these two programs there is relatively little contention for accessing shared data. Similarly, the redo times are also negligible in both cases for the same reason since there are not many interprocessor interferences in these two applications. These two factors are important and bode well for our technique since as we pointed out earlier, the possibility of rollbacks and the associated redo could be the limiting factor for performance in such optimistic techniques. There is relatively little time spent in patch-init as well for both applications. This shows that our polling mechanism works well for switching the participating processors to the patch phase. As expected the Load and Store times scale with the number of processors for both applications. We would expect the normal-patch-time to scale as well with increasing number of processors. This is true for the Matmul algorithm. For IS the scaling is not as good beyond 4 processors. This is again due to the effects of Amdahl's law. In the normal-patch-time there is a certain amount of sequential processing when one processor has to set up certain structures prior to patch phase. Further, the cleanup at the end while being done in parallel does involve acquiring locks on DE to remove history elements. With the increased number of processors and a fixed problem size this sequential component starts to be the limiting factor. However, it should be clear that with more realistic problem sizes this would not be a problem. The important thing to note is that the overheads of the simulation are either small or scale as a function of the problem size which gives the assurance that this technique will scale for doing performance studies of larger systems (both number of processors and problem sizes).

A quick observation regarding polling versus interrupts. As we mentioned earlier, we use a polling mechanism to switch to the patch phase. Thus the waiting time is sensitive to the polling frequency and the number of processors. Since we poll on events that trap into the simulator for effecting this switch this waiting time is not deterministic if the application is busy in a computation phase. While this was not a problem for the applications we simulated it is clear that switching

to the patch phase can be effected much more efficiently if the architecture supports low overhead user-level interrupts.

8 Concluding Remarks

We have developed and implemented a new method for parallel simulation of multiprocessor caches. The primary advantages of this scheme are that we can obtain reasonable speedups limited primarily by the application speedups and that this method can be used to simulate larger parallel systems (both number of target processors and problem size) than is possible with a sequential simulator.

In the process of implementing the technique we learned several lessons which are of interest from the point of view of performance tuning parallel applications in general, parallel simulators in particular. The first has to do with the potential for considerable speedup for parallel simulation of multiprocessor caches by gleaning compile time information on the data layout in the caches and passing it on to the simulator. The second is the importance of distributing the overhead of the simulation among the processors. The third is the effectiveness of polling, as opposed to interrupts, for interprocessor communication.

There are several directions to extend our work. One direction is to figure out a way to incorporate network contention of the target architecture into the simulator. This direction would allow extending our technique to simulate both memory and I/O intensive applications (which may stress the network) in addition to the compute intensive ones we have studied so far. A second direction is to use this technique to compare different memory systems employing different models of consistency and cache coherence strategies. A third direction is to simulate larger configuration of target architectures on smaller host machines. In our implementation, we used one host processor to simulate one target processor. The timepatch technique does not place any such constraints and we could just as easily have mapped multiple nodes of the target machine to a single processor of the host machine. We also assumed that each target processor has only one thread of the application mapped to it. To relax this restriction we have to take into account the scheduling strategy on the target machine so that the cache effect produced by multiple threads on a processor can be accurately modeled.

Acknowledgments

We would like to thank Jim Winget of SGI, who suggested the idea of using compiler layout of data structures to prune the patch table entries. We would also like to thank the members of our weekly "arch beer" meetings for constructive feedback.

References

[AB86] J. Archibald and J. L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–98, November 1986.

[ASHH88] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *15th Annual International Symposium on Computer Architecture*, pages 280–9, May 1988.

[BDCW91] 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, Massachusetts Institute of Technology, September 1991.

[Bit89] Philip Bitar. A critique of trace-driven simulation for shared-memory multiprocessors. In *16th ISCA Workshop Presentation*, May 1989.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. v. Eicken. Logp: towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, July 1993.

[CM79] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[CMM+88] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. *Performance Evaluation Review*, 16(1):4–11, May 1988.

[DGH91] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using TANGO. In *International Conference on Parallel Processing*, pages II–99–107, 1991.

[DHN94] P. M. Dickens, P. Heidelberger, and D. Nicol. A distributed memory LAPSE: Parallel simulation of message passing programs. In *8th Workshop on Parallel and Distributed Simulation*, pages 32–38, July 1994.

[EK88] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *15th Annual International Symposium on Computer Architecture*, pages 373–82, June 1988.

[FH92] Richard M. Fujimoto and William C. Hare. On the accuracy of multiprocessor tracing techniques. Technical Report GIT-CC-92-53, Georgia Institute of Technology, November 1992.

[Fuj83] R. M. Fujimoto. Simon: A simulator of multicomputer networks. Technical Report UCB/CSD 83/137, ERL, University of California, Berkeley, 1983.

[Fuj90] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[GH93] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-drive simulations of multiprocessors. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 146–57, June 1993.

[HS90] Philip Heidelberger and Harold S. Stone. Parallel trace-driven cache simulation by time partitioning. In *Winter Simulation Conference*, pages 734–737, December 1990.

[Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lig92] Walt B. Ligon. *An empirical analysis of Reconfigurable Architectures.* PhD thesis, Georgia Institute of Technology, August 1992.

[MRSW] Henk L. Muller, Sanjay Raina, Paul W. A. Stallard, and David H. D. Warren. Parallel calibrated emulation as a technique for evaluating parallel architectures. *International Journal of Computer Simulation.* To Appear in a Special issue on simulation in parallel and distributed computing environments.

[NGLR92] David M. Nicol, Albert G. Greenberg, Boris D. Lubachevsky, and Subhas Roy. Massively parallel algorithms for trace-driven cache simulation. In *6th Workshop on Parallel and Distributed Simulation*, pages 3–11, January 1992.

[Pop90] D. A. Poplawski. Synthetic models of distributed memory parallel programs. Technical Report ORNL/TM-11634, Michigan Technological University, September 1990.

[Res92] Kendall Square Research. Technical summary, 1992.

[RHL+92] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. Technical report, University of Wisconsin-Madison, November 1992.

[RSRM93] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability study of the ksr-1. In *International Conference on Parallel Processing*, pages I–237–240, August 1993.

[SA88] R. L. Sites and A. Agarwal. Multiprocessor cache analysis using ATUM. In *15th Annual International Symposium on Computer Architecture*, pages 186–95, June 1988.

[Sch90] Herb D. Schwetman. CSIM Reference Manual (Revision 14). Technical Report ACA-ST-252-87, Microelectronics and Computer Technology Corp., Austin, TX, 1990.

[SRF95] G. Shah, U. Ramachandran, and R. Fujimoto. Timepatch: A novel technique for the parallel simulation of multiprocessor caches. In *Proceedings of the ACM SIGMETRICS 1995 Conference on Measurement and Modeling of Computer Systems*, pages 315–316, May 1995.

[SSRV94] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. An approach to scalability study of shared memory parallel systems. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 171–180, May 1994.