*Proceedings of the 1989 Winter Simulation Conference*
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

# TECHNIQUES FOR THE TRACE-DRIVEN SIMULATION OF CACHE PERFORMANCE

Susan J. Eggers
Edward D. Lazowska
Yi-Bing Lin
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

## ABSTRACT

In contemporary computers, cache memories are interposed between processors and primary memories in order to decrease access time and bus traffic. Because the design of the cache is critical and the factors affecting its performance are complex, trace-driven simulation is widely used and studied. This paper surveys three interesting techniques for the trace-driven simulation of cache designs: stack analysis methodologies that make it possible to obtain performance measures for a wide variety of cache designs from a single run of the simulator, compression algorithms specifically tailored to memory reference traces, and an approach to parallel trace-driven simulation of multiprocessor caches that dramatically reduces the simulation's synchronization and thus its running time.

## 1. INTRODUCTION

In contemporary computers, cache memories are interposed between processors and primary memories in order to decrease access time and bus traffic.

The design of the cache is critical to system performance. There are a large number of design parameters: size, associativity, speed, coherence policy (for multiprocessors), and others. Cache performance is also dependent on the detailed memory reference patterns of programs. For all of these reasons, trace-driven simulation is widely used to gauge the performance of alternative cache configurations. The technique has been the focus of considerable research activity, the goal of which is to reduce its potentially enormous computational demands. This paper surveys three developments in this area.

The first technique that we consider is a stack analysis simulation methodology that makes it possible to obtain performance measures for a wide variety of cache designs from a single run of the simulator. This technique, originally developed in the context of virtual memory systems (Mattson et al. 1970), was easily adapted to simultaneously simulating multiple cache sizes. It has recently been extended to other cache parameters, such as associativity (Hill 1987) and copyback and multiprocessor caches (Thompson 1987).

The second technique is a set of compression algorithms specifically tailored to memory reference traces (which can be huge). Originally devised for paging behavior (Smith 1979), the technique has been adapted to uniprocessor cache simulations (Puzak 1985) and extended to multiprocessor studies (Eggers 1989).

The third technique is an approach to the parallel trace-driven simulation of multiprocessor caches that dramatically reduces the simulation's synchronization and thus running time (Lin, Baer & Lazowska 1989).

## 2. STACK SIMULATION

The first technique that we discuss in this paper is *stack simulation*. Our discussion follows that of Hill and Smith (1989).

It seems natural to expect to perform a separate simulation for each cache configuration that is to be evaluated (e.g., for each different cache size). Stack simulation allows performance measures for a variety of configurations to be obtained from a single simulation run, dramatically reducing simulation execution time.

Mattson et al.'s development of stack simulation (Mattson et al. 1970) is based on the *inclusion property*: subject to certain restrictions, a larger cache will always contain a superset of the blocks contained by a smaller cache. The restrictions include that the caches must have the same block size and the same set mapping function, and that the replacement algorithm must be a stack discipline (e.g., LRU). If the inclusion property holds, then a single simulation can yield performance measures for caches of different sizes, by utilizing a single stack per set to keep track of the resident blocks belonging to that set (the first $n$ entries correspond to the $n$ most recently used blocks in a set).

The benefits of stack simulation for cache design studies are obvious and dramatic. For this reason, the Mattson et al. algorithms have been extended continuously over the years. Traiger and Slutz, two of Mattson's original co-authors, generalized the algorithms to handle caches with different numbers of sets and different block sizes (1971), in the case of LRU replacement.

More recently, Hill (Hill 1987; Hill & Smith 1989) extended the Traiger and Slutz algorithms to handle caches with different associativities. Their *all associativity simulation* simultaneously simulates alternative direct-mapped and multiple set-associative caches that have the same block size, do no prefetching, and use LRU replacement, for arbitrary set mapping functions.

Thompson (1987) investigated a number of data structure alternatives for managing the simulation stacks, and extended the approach to copyback and multiprocessor caches. Essentially, Thompson showed that elements of cache state, such as dirty/clean state and coherency state, obey the inclusion property, in addition to the validity state demonstrated by Mattson et al.

## 3. TRACE COMPACTION FOR PARALLEL PROGRAMS USING A CACHE FILTER

Stack simulation is one approach to reducing the simulation execution time for cache design studies. Another is *trace compaction*, which attempts to reduce the number of trace records that must be processed, without losing information that affects the results of the simulation.

The volume of trace data required for cache simulations can be excessive even in studies of uniprocessor systems, because a fairly large snapshot (in numbers of memory references) is needed to obtain statistically significant results. As cache sizes increase, this number mushrooms. The problem is exacerbated in multiprocessor simulations because the total trace size is proportional to the number of processors. For example, the 6 million reference (per processor) traces used in (Eggers 1989), after postprocessing, comprised approximately 1.8 to 2 gigabytes for a 12 processor simulation. Practically speaking, traces must be compacted to be usable.

Traditionally, encoding schemes such as Ziv-Lempel compression (Ziv & Lempel 1977) have been used. Memory reference traces that are used for cache studies can be further reduced by special techniques, such as cache filtering, that take advantage of the particular "semantics" that apply.

### 3.1. Uniprocessor Approaches

Uniprocessor cache filters reduce the size of memory reference traces by removing all cache hits from the trace. The filter is a cache simulator, the input to which is the original trace, and the output from which is trace records of cache misses and summary information for cache hits, whose trace records are removed. In other words, only those references that cause bus operations are explicitly recorded in the filtered trace. There are two restrictions on the configurations of caches that are analyzed with the filtered traces. Both restrictions guarantee that caches of varying sizes see correct hit/miss behavior. First, the analyzed caches must contain no fewer sets than the cache simulated in the filter. Second, they must use the same block size as the filtered cache.

The first example of this approach, the stack deletion filtering technique described in (Smith 1979), was developed to analyze program paging behavior but is easily adapted for cache studies. In the most general version of the scheme, references to the first $D-1$ positions ($D$ being the deletion parameter) in an LRU stack of memory references are removed from the trace, and a counter, reflecting the amount of processing time required for the number of references that were eliminated, is output with the next recorded reference. The technique achieves a reduction in trace length (where length is defined in numbers of entries) of a factor of 14 to 36, when $D$ is, for example, 6. However, since the algorithm is not directly tied to a cache simulation, some error in the hit/miss classification is introduced when using the compressed data when $D$ is greater than 2.

An alternative that addresses this problem, first described in (Puzak 1985), is to base the filtering directly on a cache simulation rather than on an LRU stack approximation. The Puzak approach represents the summary cache hit information using runlengths of consecutive hits. For caches ranging from 4K to 16K bytes, with 64 and 128 byte blocks, this method produces a filtered trace approximately one tenth the size of the original.

Another alternative (Samples 1989) is to record the difference between the address of the reference that hits in the cache and the one that most recently missed in the same block. Because this exposes the patterns of locality in memory references, it produces traces that are good candidates for further compression by schemes that rely on pattern matching techniques. When used as a preprocessor to Ziv-Lempel compression, this technique produces compressed files at least as small as Puzak's approach. Although it does not compact as well as some other schemes, it retains all the information from the original trace; the original trace can be reconstructed, e.g., to regenerate traces for a different block size or smaller cache size.

### 3.2. Multiprocessor Approaches

In generalizing Puzak's approach to the multiprocessor case, Eggers (1989) broadened the criterion for reference elimination by cache filters from Puzak's simple hit/miss model to a filtering technique that is based on any change of state, in order to represent the additional bus operations caused by sharing. In this more general scheme, state is defined as the superset of cache state, dirty/clean state, coherency state and synchronization state. (Thompson (1987) uses a similar state definition as the basis for a technique for simulating multiple sized caches in a multiprocessor, alluded to in Section 2 above.) Cache state is the criterion used in (Puzak 1985); it differentiates between valid and invalid blocks, and is required to support the hit/miss criteria. Dirty/clean state distinguishes between the first write to a block and all others. It is needed for two reasons. First, only the first write in a sequence of per processor writes generates a bus operation (the invalidation signal) in the write-invalidate protocols; and, second, dirty, private data must be copied to memory on block replacement.

Coherency state includes the five MOESI values (invalid, private clean, shared clean, private dirty, shared dirty) (Sweazey & Smith 1986). Generating a memory reference for any potential change of coherency state essentially means that all shared operands are output. All write-shared references must be recorded, because it cannot be determined a priori, which will result in a bus broadcast

in the write-broadcast protocols. Read-shared operands must also be included, because in the write-invalidate protocols, write hits produce a different bus operation (an invalidation) than write misses (a full data transfer). For the hit to be detected, the block must already reside in the cache when the write occurs.

Synchronization state comprises a processor's first attempt to obtain a lock, its acquiring the lock, its unlocking it, and its reaching a barrier or flushing the cache. Memory references that correspond to these coherency-related instructions (i.e., instructions that implement locking and unlocking, reaching a barrier, and executing software coherency mechanisms) must also be output. If not explicitly recorded in the filtered trace, the multiprocessor simulator would be unable to synchronize processes and ignore and/or replace them with other sequences of code.

Explicitly specifying these output records leaves runlengths only to hits of instructions and private data (excepting the first write to the blocks). (The runlengths may be further subdivided into separate runs of contiguous reads and writes. The subdivision is required to ensure correct simulation, when the simulator's cache controller implements one-cycle cache reads and two-cycle cache writes.) The amount of trace reduction achieved by substituting runlengths for trace records is called the compression ratio. The compression ratio is defined as the number of items in all runlengths (number of references eliminated), divided by the total number of memory references. Multiprocessor traces filtered in this way are roughly 15% of the size of the original unfiltered traces (Eggers 1989). When further compressed with Ziv-Lempel encoding, the final traces are approximately 4.5% of the size of the originals.

## 4. PARALLEL TRACE-DRIVE SIMULATION OF MULTIPROCESSOR CACHES

Since simulations are time-consuming, it is natural to attempt to use parallel computing to accelerate them. The final topic that we consider in this paper is a parallel trace-driven simulation technique for multiprocessor cache coherence protocols, originally described in (Lin, Baer & Lazowska 1989). We first present a general technique, and then optimize this technique to simulate various specific protocols. The surprising result is that using this technique, the processes simulating the caches often need to do little or no communication, even when simulating shared references. Thus, linear or near-linear speedup is possible.

In the parallel cache simulation, we have one *cache process* to simulate each processor-cache pair. There is a separate reference trace for each cache process. Each trace includes two types of memory references: *private* and *shared*. A private reference originating at processor/cache $j$ does not have any effect on caches other than $j$, whereas a shared reference may update the status of other caches. All caches can be simulated asynchronously in parallel as long as there are no shared references, but concurrency control of some sort is required when shared references are encountered.

Two mechanisms can be used to synchronize the cache processes. The *optimistic* mechanism assumes that no communication is required between two processes. When a communication request from cache process $i$ to cache process $j$ occurs, and $j$'s simulation time is ahead of $i$'s, $j$'s computation is rolled back to the synchronization point. The optimistic mechanism needs to save process state, so that it can be restored when rollbacks occur. Because of the complexity of this state when simulating caches, the optimistic mechanism does not appear practical in this case.

An alternative is the *conservative* mechanism, which ensures that every event (either a reference request from the input trace or a synchronization request from other cache process) is handled according to timestamp order. In other words, before cache process $i$ handles the next memory reference from its input trace, it checks other cache processes to see if any inter-process communication may occur. This implies that all processes are synchronized at every event, and the synchronization overhead is too high to be acceptable. In the next subsection, we describe a simple technique which effectively reduces this overhead.

### 4.1. The Basic Algorithm

The idea of the basic simulation technique is to pre-process the input traces so that all shared references are inserted into each input trace. These references added to each trace are called *inserted references*. With these merged traces, all potential interactions among caches are identified before the simulation commences; we refer to these as *interaction points*. During the simulation, all private references between two interaction points can be handled without interruption, and the synchronization overhead for the conservative mechanism is considerably reduced. In (Lin, Baer & Lazowska 1989) we show that if (1) there are $k$ cache processes, each assigned to a dedicated processor, and (2) the arrivals of interaction points to each cache form a Poisson process, with the same mean for each cache, then the speedup is $O(\sqrt{k})$.

### 4.2. The Modified Algorithm

The basic algorithm requires that all cache processes synchronize at each interaction point. The modified algorithm relaxes this restriction. At some interaction points, a cache process can determine its cache status using only local information (including that provided by its merged trace), so there is no need to consult the status of other caches. Thus we can process these interaction points in the same way as private references. Whether an interaction point can be processed locally is dependent on the type of interaction (a shared reference, which is a *processor-induced transaction*, or an inserted reference, which is a "possible" *bus-induced transaction*) and the cache coherence protocol under investigation.

In Table 1, the first column lists various information that is required by different cache coherence protocols, while the second and third columns indicate the sources of this information for processor-induced and bus-induced transactions, respectively.

| information required | source of information | |
|---|---|---|
| | processor-induced transaction | bus-induced transaction |
| current state of the block | status of the cache process | status of the cache process |
| read/write | arrival event | arrival event |
| hit/miss | status of the cache process | status of the cache process or other cache procs. |
| shared/ non-shared | status of other cache processes | status of the cache process |

**Table 1: Sources of Information**

From Table 1 we see that synchronization is required during the simulation if (1) the cache coherence protocol requires information about actual sharing for a processor-induced transaction, or (2) the protocol cannot provide information about hit/miss for a bus-induced transaction. A transaction (or an interaction point) is *synchronous* iff it requires synchronization in the simulation. An interaction point that requires synchronization is a *synchronous point*. Otherwise, it is a *non-synchronous point*.

As just noted, the specific cache coherence protocol under investigation has a bearing on whether an interaction point is synchronous (requires synchronization) or not. Table 2 gives "synchronization information" about five protocols: Berkeley ownership (Katz et al. 1985), Illinois (Papamarcos & Patel 1984), write-once (WO) (Goodman 1986), Firefly (Thacker & Stewart 1987), and Dragon (McCreight 1984). The first three of these protocols are based on "invalidation", while the last two are of the "distributed-write" type.

| protocol | shared | | | | inserted | |
|---|---|---|---|---|---|---|
| | read | | write | | read | write |
| | hit | miss | hit | miss | | |
| Berkeley | | | | | | |
| Illinois | | X | | | | |
| WO | | X | | | | |
| Firefly | | X | | X | | |
| Dragon | | X | | X | | |

("X" means synchronization is required)

**Table 2: Synchronization Information for Protocols**

Table 2 shows that a surprisingly large number of event types can be simulated without the need for synchronization. For example, an ownership-based invalidation protocol such as Berkeley ownership does not require *any* synchronization. The other two invalidation-based protocols require synchronization only at read-misses. For the distributed-write protocols, synchronization on write-misses is also required. The modified algorithm takes advantage of this observation.

The timing analysis of the modified algorithm is complex and is not shown in detail here. (See (Lin, Baer & Lazowska 1989).) The key result is that, under the assumptions stated earlier ($k$ cache processes, each assigned to a dedicated processor, and Poisson arrivals of interaction points to each cache with the same mean for all caches), the time spent in synchronization is always less than the computation time. In other words, the simulation speedup will be at least 50% of the number of processors.

## 5. SUMMARY

We have described three approaches for reducing the computational demands of cache design studies employing trace-driven simulation. Stack simulation allows performance results for a variety of cache configurations to be obtained from a single simulation run. Trace compaction reduces the number of specific events that must be simulated, without reducing the information content of the simulation. Parallel simulation allows the power of multiple processors to be applied to a single simulation run.

## ACKNOWLEDGEMENTS

Our research has been supported in part by the National Science Foundation (Grants No. MIP-8352227, CCR-8619663 and CCR-8703049), the Naval Ocean Systems Center, the Washington Technology Center, U S WEST Advanced Technologies, Digital Equipment Corporation (the Systems Research Center and the External Research Program), IBM (through a Predoctoral Fellowship), SPUR/DARPA contract No. N00039-85-C-0269, and California MICRO (in conjunction with Texas Instruments, Xerox, Honeywell, and Philips/Signetics).

## REFERENCES

Eggers, S.J. (1989). Simulation Analysis of Data Sharing in Shared Memory Multiprocessors. Technical Report 89/50, Computer Science Division, University of California, Berkeley. (Ph.D. Thesis).

Goodman, J.R. (1987). Cache Memory Optimization to Reduce Processor/Memory Traffic *Journal of VLSI and Computer Systems* 2,1, pp. 61-86.

Hill, M.D. (1987). Aspects of Cache Memory and Instruction Buffer Performance. Technical Report 87/381, Computer Science Division, University of California, Berkeley. (Ph.D. Thesis).

Hill, M.D., and Smith, A.J. (1989). Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, to appear.

Katz, R., Eggers, S., Wood, D.A., Perkins, C., and Sheldon, R.G. (1985). Implementing a Cache Consistency Protocol. *Proc. 12th Annual International Symposium on Computer Architecture*, pp. 276-283.

Lin, Y.-B., Baer, J.-L., and Lazowska, E.D. (1989). Tailoring a Parallel Trace-Driven Simulation Technique to Specific Multiprocessor Cache Coherence Protocols. *Proc. 1989 SCS Multiconference on Distributed Simulation.*

Mattson, R.L., Gecsei, J., Slutz, D.R., and Traiger, I.L. (1970). Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal 9*,2, pp. 78-117.

McCreight, E. (1984). The Dragon Computer System: An Early Overview. Technical Report, Xerox Palo Alto Research Center.

Papamarcos, M., and Patel, J. (1984). A Low-Overhead Coherence Solution for Multiprocessors With Private Cache Memories. *Proc. 11th Annual International Symposium on Computer Architecture*, pp. 348-354.

Puzak, T.R. (1985). Analysis of Cache Replacement Algorithms. Ph.D. Thesis, University of Massachusetts.

Samples, A.D. (1989). Mache: No-Loss Trace Compaction. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 89-97.

Smith, A.J. (1979). Two Methods for the Efficient Analysis of Memory Address Trace Data. *IEEE Transactions on Software Engineering SE-3*,1, pp. 94-101.

Sweazey, P., and Smith, A.J. (1986). A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE. *Proc. 13th International Symposium on Computer Architecture*, pp. 414-423.

Thacker, C.P., and Stewart, L.C. (1987). Firefly: A Multiprocessor Workstation. *Proc. 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 164-172.

Thompson, J.G. (1987). Efficient Analysis of Caching Systems. Technical Report 87/374, Computer Science Division, University of California, Berkeley.

Traiger, I.L., and Slutz, d.R. (1971). One-Pass Techniques for the Evaluation of Memory Hierarchies. Technical Report RJ892, IBM.

Ziv, J., and Lempel, A. (1977). A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory 243*,3, pp. 337-343.

## AUTHORS' BIOGRAPHIES

SUSAN J. EGGERS is on the faculty of the Department of Computer Science and Engineering at the University of Washington. She received her Ph.D. from the University of California at Berkeley in 1989. Her research interests are in computer architecture.

EDWARD D. LAZOWSKA is on the faculty of the Department of Computer Science and Engineering at the University of Washington. He received his Ph.D. from the University of Toronto in 1977. His research interests are in computer systems.

YI-BING LIN is a graduate student in the Department of Computer Science and Engineering at the University of Washington. His research interests are in parallel simulation.

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195