Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

PARALLEL TRACE-DRIVEN CACHE SIMULATION BY TIME PARTITIONING

Philip Heidelberger
Harold S. Stone

IBM Research Division
T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598

ABSTRACT

This paper describes a technique for performing parallel simulation of a trace of address references for the purposes of evaluating different cache structures. An obvious way to achieve fast parallel simulation is to simulate the individual independent sets of a cache concurrently on different computers, but we show this technique is not efficient in a statistical sense because of a high correlation of the activity between different sets. Only a small fraction of the sets should actually be simulated. To put parallelism to effective use, we show that a trace of the sets to be simulated can be partitioned into disjoint time intervals, and each interval can be simulated concurrently. Because the contents of the cache are unknown at the start of the time intervals, this parallel simulation does not produce the correct counts of cache hits and misses. However, after simulating the trace in parallel, a small amount of resimulation can produce the correct counts. The resimulation effort required is proportional to the size of the cache simulated and is not proportional to the length of the trace.

1. INTRODUCTION

Caches are small high-speed buffer memories that play a major role in the performance of high-speed computer systems. Consequently, as part of the design of a new computer, many different cache architectures are evaluated and compared under realistic operating conditions in order to assure that a cache design is both effective in attaining high-performance and is efficient in terms of its cost. Such analysis is generally performed by trace-driven simulation from address-reference traces of realistic workloads. Introductions to cache architectures and analysis techniques can be found in [Stone 1987] and [Smith 1982].

The problem of speeding up the process of cache simulation has been widely studied because the number of design variations to be considered is relatively large, and the length of cache traces required for high accuracy may run to hundreds of millions of references. In spite of research advances in this area, cache analysis is still very costly, and it becomes worthwhile to consider parallel methods for cache simulation in order to reduce the simulation time. An algorithm for parallel simulation is the subject of this paper.

Several authors have contributed techniques for reducing the complexity of cache analysis using conventional serial programs. The stack processing algorithm of Mattson et al. [1970] allows the simultaneous determination of cache hit-ratios for multiple caches of different sizes during a single pass simulation of the trace. The algorithm works for a certain class of cache replacement algorithms including LRU (Least Recently Used), which is the most commonly implemented replacement algorithm. The stack processing algorithm takes advantage of the fact that, under this class of replacement algorithms, the contents of a smaller cache are contained within the contents of a larger cache. This type of structure has also been exploited by Puzak [1985] who reduces the length of the trace by passing it through a simulation of a small cache. Essentially, only the misses to the small cache are recorded and subsequently used as input to simulations of a variety of larger caches. Hill and Smith [1989] introduce extensions of this idea that permit different cache structures to be simulated on a single pass of a trace. Thompson and Smith [1989] describe a simulation means for capturing traffic generated by writes as well as miss ratios, and are able to do both operations simultaneously for multiple cache designs during a single pass of a trace.

Efficient techniques for generating traces have been studied by Stunkel and Fuchs [1989] and Dwarkadas et al. [1989], while an efficient technique for storing traces (i.e., trace compaction) has been described by Samples [1989].

However, there has been little work on parallelizing trace-driven cache simulations. Lin et al. [1989] describe a technique for parallel simulation of a cache-coherent multiprocessor system. This technique uses one processor for each simulated cache. Simulations of all caches can run concurrently, but must synchronize at points where an intercache reference occurs. This involves inserting all intercache references into the reference trace of each individual cache. To obtain correct simulation data, it is sufficient to synchronize at every intercache reference, but it is possible to reduce the number of synchronizations for certain cache-coherence protocols. The parallel simulation algorithm has the property that the maximum speedup is limited to the number of caches being simulated. In particular, no speedup is obtained when simulating the cache of a uniprocessor. In addition, if each simulated cache contains at least n references, then the run length of the algorithm is $\Omega(n)$.

In this paper we investigate the problem of performing the analysis of a cache trace on a parallel processor system. The method we use is to break a long trace into several shorter traces, and to process the shorter traces in parallel on independent processors. No synchronizations are required during the parallel simulation. At the end of the simulation, it may be necessary to have a single synchronization in order to determine when all simulations have completed.

The output counts of cache hits and misses obtained at the end of the simulation are inaccurate because no processor has the correct initial cache state at the beginning of its simulation. We show that by resimulating a small number of references, the correct counts can be obtained. Hence, the parallel simulation achieves a speedup that is nearly proportional to the number of processors with the overhead limited to the initial partitioning of the input data and to the resimulation phase. Thus a large number of processors can be applied to a single long trace.

Practical implementations of caches avoid costly searches of cache memory by partitioning the address space into disjoint regions. Each region of the address space is assigned to a small set of lines of cache memory, thereby reducing the search effort in the cache. An obvious alternative way to parallelize the simulation is to take advantage of the set structure of caches, and have a different processor simulate each set. Since references to different sets never affect each other, simulation of different sets can be done totally independently of one another. We show in this paper that set partitioning is not statistically efficient, and that parallel simulation by time partitioning is a satisfactory way to achieve useful speedup.

Our general approach bears some resemblance to the Space-Time parallel simulation approach of Chandy and Sherman [1989]. However, we describe a concrete, efficient (and important) example of parallel simulation by time partitioning, whereas the discussion in [Chandy and Sherman 1989] provides no guidelines for the efficient application of Space-Time to real problems. The approach of Greenberg et al. [1990], which was developed independently of our work, for solving (i.e., simulating) certain queueing recurrence relations

using parallel prefix and "iterative folding", is also somewhat similar in spirit to our approach in that both techniques involve additional simulation passes to correct earlier tentative simulations.

The remainder of the paper is organized as follows. In Section 2, we review cache structures, and formulate a stochastic model for comparing the efficiency of set and time partitioning. In Section 3, we describe the basic parallel-time partitioning algorithm for uniprocessor caches. In Section 4, we summarize the results and indicate some generalizations.

2. A STOCHASTIC MODEL OF CACHE HIT-RATIOS

In this section we review the structure of a cache memory to remind the reader that it is composed of independent sets, each of which operates on a disjoint part of the address space. The independence leads to a natural parallel method for cache simulation. We complete the section by showing that the natural parallel method is not a statistically efficient technique for performing cache analysis because the data for the independent sets are highly correlated.

2.1 Cache Structure

Each access to cache memory causes the cache to be searched for the presence of a given address. If the address is present, the memory request is honored by the cache. If the address is not present, the memory request is honored by a remote memory, but with a higher delay.

For practical reasons, the search is conducted over a small number of items in cache memory rather than over the full memory. The collection of items actually searched is called a *set* and the cache is said to be *set associative*. The memory address space of a system is partitioned into m disjoint address spaces, typically by selecting $\log_2 m$ bits of an address by letting these bits designate the set of addresses in one of the m disjoint sets of the partition. A cache for this address space is also organized into m sets, and each cache set contains a fixed number of cache lines, k. We say the cache is *k-way set associative* in this case. Typically, k= 1, 2, or 4. Each cache line contains multiple words and an address tag that indicates the corresponding address in main memory of that cache line.

During a cache memory access, $\log_2 m$ bits of the memory address identify the set of the cache to access. When the set is accessed, the cache reports all k lines in the set together with the address tags stored with those lines. All address tags are compared with the access address simultaneously, and if one of them matches the access address, we say that a *hit* has occurred. The data from the corresponding line are the data requested. If no match occurs, the data must be obtained from a remote memory. In this case we say that a *miss* occurred.

On a read miss, a line is selected for replacement and the referenced line from a remote memory replaces the line selected. On a write miss, one possible action is to retrieve the full cache line from a remote memory, replace the LRU line in the set with this line, and update the word or bytes in the line that are altered by the write. Some cache designs treat writes differently, but the differences do not affect the results reported in this paper. The goal of a cache simulation is to determine the hit ratio, the fraction of all references that are hits.

2.2 Analyzing Concurrent Simulations of Cache Sets

Because the address space is divided into m disjoint subspaces, we can simulate the activity in each of the spaces independently and in parallel. Hence, we can obtain up to an m-fold speedup rather trivially by a parallel simulation. However, this is somewhat misleading. Although we can do the computation m times faster by using parallelism, the data obtained by the parallel simulation are not much more accurate than the data obtained by simulating only a few of the disjoint subspaces. Hence, the simulation is not efficient in a statistical sense, and we are wasting much of the computational effort.

To see why this is true, let $\mathbf{T} = \{A(k), k \geq 1\}$ be the trace of address references to two sets (or groups of sets). We view the trace as being very long, in effect of infinite length. This trace can be partitioned into two subtraces of references to each of the individual sets. Denote the subtraces by $\mathbf{T}_i = \{A_i(k), k \geq 1\}$, $i = 1, 2$. Let $N_i(n)$ be the number of references to set i in the first n references of $\mathbf{T}$ and let $H_i(n)$ be the number of hits to set i in the first n references of $\mathbf{T}$. Let $H(n) = H_1(n) + H_2(n)$ and define the hit ratios

$$\overline{H}(n) = H(n)/n \ , \quad \overline{H}_i(n) = H_i(n)/N_i(n) \ .$$

Notice that $\overline{H}_i(n)$ is the hit ratio to set i in the first $N_i(n)$ references in subtrace $\mathbf{T}_i$.

We now assume that the behavior of the cache and the address traces can be modeled by stochastic processes sufficiently well behaved so that the cache hit ratios obey strong laws and central limit theorems. Specifically we assume that the sets are stochastically identical in the sense that

$$\lim_{n \to \infty} \overline{H}_i(n) = h \text{ a. s. (almost surely)}$$

for some constant h (the steady-state hit ratio),

$$\lim_{n \to \infty} N_i(n)/n = 0.5 \text{ a. s.}$$

and

$$\left(\sqrt{N_1(n)}\ (\overline{H}_1(n) - h), \sqrt{N_2(n)}\ (\overline{H}_2(n) - h)\right) \Rightarrow \mathbf{N(0,A)} \qquad (1)$$

as $n \to \infty$ where $\Rightarrow$ denotes convergence in distribution, and **N(0,A)** denotes a bivariate normally distributed random vector with means zero and covariance matrix **A**. We further assume that $A_{11} = A_{22} = \sigma^2$ and $A_{12} = A_{21} = \rho\sigma^2$. By multiplying Equation 1 by

$$\left(\sqrt{N_1(n)/n}\ , \sqrt{N_2(n)/n}\ \right),$$

summing the two components, and applying standard weak convergence arguments, we obtain

$$\sqrt{n}\ (\overline{H}(n) - h) \Rightarrow N(0, \sigma^2(1 + \rho)).$$

Thus a simulation of r references of $\mathbf{T}$ results in a variance of approximately $\sigma^2(1 + \rho)/r$, whereas from Equation (1), a simulation of r references of subtrace $\mathbf{T}_i$ results in a variance of approximately σ^2/r. As a first order approximation, it takes equal effort to simulate r references to the full trace $\mathbf{T}$ or r references to the set i subtrace $\mathbf{T}_i$. However, if $\rho > 0$, the simulation of just $\mathbf{T}_i$ produces a smaller variance than the simulation of addresses from both sets. The intuition behind this is that the r references of $\mathbf{T}_i$ run over a longer period of time than do the r references from the full address space. There is more information obtained by running a single set for a longer time than there is in examining a collection of highly correlated references produced over a shorter period of time.

Thus the single set simulation is statistically more efficient provided $\rho > 0$. In general, ρ is greater than 0, and has been observed to be fairly high. In other words, when one set experiences a high rate of misses, say at a context swap, all sets are likely to experience a high rate of misses. This argument also formalizes the notion of set sampling as described in [Puzak 1985]. Although he did not directly measure the correlation, Puzak found that he could obtain highly accurate estimates by simulating only about 10% of the sets.

Turning now to the parallel processing implications of this result, suppose it takes one unit of time for a processor to simulate a reference. Then it takes r units of time for Processor 1 to simulate r references of $\mathbf{T}_1$ and for Processor 2 to simulate r references of $\mathbf{T}_2$ concurrently, since the simulations can proceed independently of one another. The resulting variance from this simulation is $\sigma^2(1 + \rho)/(2r)$. Next suppose that in r time units, two processors can simulate $2\alpha r$ references of subtrace $\mathbf{T}_i$ using the time partitioning algorithm. We expect $\alpha < 1$ since there is a certain amount of overhead. The variance from this simulation

is $\sigma^2/(2\alpha r)$, so time partitioning is more efficient than set partitioning provided $(1/\alpha) \leq (1 + \rho)$. Since the time partitioning algorithm is very efficient for long traces, $\alpha \approx 1$, and since $\rho > 0$, time partitioning will usually be more efficient than set partitioning.

To achieve the benefits of both time and set partitioning, we recommend that the simulation be limited to a small number of sets, as per the findings of Puzak [1985], and that the trace of these addresses be partitioned in time and simulated in parallel as described in the next section.

3. THE ALGORITHM

We assume that the input trace contains references to just the collection of sets that are to be simulated. We also assume that the cache replacement policy is LRU within a set.

Assume the input trace consists of N entries and that there are P processors. The input trace is partitioned into P sequential subtraces, the first consisting of the first N/P references of the original trace, the next consisting of the next N/P references of the original trace, etc. If N is not a multiple of P, the subtraces should be as equal as possible in length. Each trace is analyzed on a separate processor. Each processor initializes its cache contents to an illegal tag value. There are two types of illegal tags: $+\infty$ and $-\infty$. The $+\infty$ tag is used to initialize the contents of the cache at the beginning of the first subtrace. This tag is used because the cache contents at the beginning of the trace are unknown (and can never be known). All other processors initialize their caches to the $-\infty$ tag. This tag is used because the cache contents at the beginning of the first pass through the subtrace are unknown. However, in subsequent passes, the $-\infty$'s may be replaced by valid addresses. Each processor simulates its input subtrace with the initial cache contents and produces a count of the number of misses. This involves maintaining a list of most recently used (MRU) to LRU lines within each set. On a miss, the LRU line is discarded, and the address causing the miss is put in the MRU spot in the list. If a reference is not in the cache, then it is counted as a miss. In addition to counts, whenever the simulation removes a value of $-\infty$ from the cache on a miss, the processor also records the address that experienced the miss. We call this sequence of addresses the resimulation subtrace. Notice that if there are k lines in the set, then the resimulation subtrace for the set contains at most k entries.

At the end of the simulation, each processor has a count of the number of misses. The sum of these misses is actually an upper bound on the true number of misses, but is too high. Each of the P simulations produces misses at the beginning of its subtrace that may, in fact, be hits on a continuous sequential simulation of the trace. The reason that they are recorded erroneously as misses is that the true contents of the cache are unknown at the beginning of the subtrace, and thus, the initial references to the cache produce misses. Note that a lower bound on the misses can also be computed by assuming that a reference that replaces a $-\infty$ was actually a hit. In order to obtain the correct counts, we need to determine which of these references should have produced hits.

To determine the correct counts, a resimulation phase is required. In practice, if the upper and lower bounds are tight enough then resimulation can be omitted. Processor i ($i \geq 2$) initializes its cache to be the final cache contents from processor $(i - 1)$'s simulation. The processor then simulates the references in its resimulation subtrace in the order they were recorded. These references are precisely the set of addresses that gave misses that might otherwise be hits. The cache state for recognizing whether these addresses should be hits is exactly the cache state of Processor $(i - 1)$ at the end of its simulation. When these addresses are presented in sequence to the cache of Processor i, for each hit observed, Processor i reduces its miss count by 1. In addition, because the replacement algorithm is LRU, we know exactly which line to replace on a miss: one simply uses LRU replacement again. For example, consider simulation of a single set and suppose the set size $k = 4$, the resimulation subtrace is $< B_1, B_2, B_3, B_4 >$ and the contents of the cache at the beginning of resimulation are (in MRU to LRU order) $< A_1, A_2, A_3, A_4 >$. (For the purpose of this discussion we assume that each line contains exactly one word.) Assume that $B_b \neq A_a$ for any a and b so that none of the references in Processor i's resimulation subtrace are in the final cache contents from Processor $(i - 1)$'s simulation. In the resimulation, after the first reference the cache contents are $< B_1, A_1, A_2, A_3 >$ while after the second reference the cache contents are $< B_2, B_1, A_1, A_2 >$. After the third and fourth references the contents are $< B_3, B_2, B_1, A_1 >$ and $< B_4, B_3, B_2, B_1 >$, respectively, although the order of the B_j's is not necessarily MRU to LRU. However, the fact that we lose the MRU to LRU order in the resimulation is now irrelevant. To see the effect of hits in the resimulation, suppose now that $B_2 = A_3$, but that the rest of the B_b's are misses. The cache contents on subsequent references in the resimulation are now $< B_1, A_1, A_2, A_3 >$, $< B_2, B_1, A_1, A_2 >$, $< B_3, B_2, B_1, A_1 >$, and $< B_4, B_3, B_2, B_1 >$. The Appendix contains a pseudo code description of the algorithm.

After resimulation, the sum of the misses observed is the correct number of misses provided the cache is filled by each subtrace. This can be determined by checking to see whether there are any $-\infty$ tags in a cache at the beginning of the resimulation. In this case additional resimulation phases are required to produce the correct counts (although, again, this may not be necessary if upper and lower bounds are tight enough).

The correctness of the algorithm depends on the following observation concerning k-way set associative caches using LRU replacement. Once a set experiences k distinct references, its future cache behavior is completely independent of the cache contents of that set, and of the reference pattern prior to the first of these k distinct references. This observation has been used by Laha et al. [1988] to develop an approximate cache analysis technique based on sampling and simulating nonadjacent time intervals from a reference trace.

If an additional resimulation pass is required, the treatment of $-\infty$ tags in a resimulation phase is exactly as before. The processor needs to save the address of the reference causing a $-\infty$ to be replaced. In the next resimulation pass, only the saved references from the previous pass need to be processed. Eventually, the algorithm will terminate, since in pass i ($i > 1$) (i.e., resimulation pass number $(i - 1)$), processor i is guaranteed not to have any $-\infty$ tags in its initial cache contents.

For very long traces, each subtrace will fill its cache and the total number of references simulated by each processor is (at most) $N/P + L$ where $L = m \times k$ is the total number of lines in the cache, assuming we simulate m sets of a k-way set associative cache. Assume now that a first pass reference takes 1 unit of time to process, while a resimulation reference takes c units of time to process (we might expect $c > 1$ to reflect processor synchronization cost and other overheads). Then the speedup is given by (at least)

$$S(P,N) = N/(N/P + cL) = P/(1 + PcL/N).$$

For modest values of P and c, and for $L/N \approx 0$, $S(P,N) \approx P$ and

$$\lim_{N \to \infty} S(P,N) = P.$$

Even in the worst case that P passes are required, the total processing time is at most $N/P + (P - 1)cL$ in which case we again have

$$\lim_{N \to \infty} S(P,N) = P.$$

Thus for very long traces we expect excellent speedup.

4. SUMMARY

In this paper we have described an approach to parallelizing a trace-driven simulation of a uniprocessor's cache memory. In this approach, the trace is subdivided into nonoverlapping time intervals. Each processor is assigned a different time interval for simulation. However, since the contents of the cache are not known at the beginning of a time interval, a short resimulation

phase is required to produce the correct counts of hits and misses. We also argued why we expect this time partitioning approach to typically be more statistically efficient than a set partitioning approach.

The algorithm can be generalized in a number of directions. First, it can be easily combined with the stack processing algorithm to generate hit ratios simultaneously for multiple sized caches. More specifically, it can generate hit ratios for k-way set associative caches (with m sets) for all values of $k = 1, \ldots, K$. Similarly, it can be combined with Puzak's trace reduction technique. Finally, with some increase in complexity, the time partitioning approach can be extended to the simulation of cache-coherent multiprocessor systems. This extension will be described in a subsequent paper.

APPENDIX. Parallel Simulation Pseudo Code

```
{ Processors execute the following operations in parallel. Initialize caches and counters, then simulate. If records remain for resimulation, then copy cache state of predecessor as initial state for this processor, and resimulate. The variable I is a processor index }
begin
  if I = 1 then initialize cache[I] to +∞
    else initialize cache[I] to −∞;
  initialize all counts to 0;
  initialize Input Trace[I] to a segment of input data;
  initialize Output Trace[I] to empty;
  Barrier Synch;
  while any Input Trace[I] is nonempty do
    begin

    { Processor I simulates cache[I] with Input trace[I], and produces Output Trace [I] }
    Simulate;
    Barrier Synch;
    if I > 1 then
      begin
      copy cache[I − 1] to cache[I];
      copy Output Trace[I] to Input Trace[I];
      end;
    Barrier Synch;
    end;
  Combine counts;
  Output results;
end { program }


{ Simulate a segment of the input data. Count all hits. Count a miss only if cache set has no entries of -∞, otherwise save the record for resimulation. This code counts misses during the initialization transient as real misses. It can be modified to ignore misses and hits until the +∞ values are flushed from the cache. }
Procedure Simulate;
begin
  for each record in Input Trace[I] do
    begin
    if hit(cache[I],record) then
      begin
      reorder cache[I];
      hits[I] := hits[I] + 1;
      end
    else
      begin
      if Cache[I] contains -∞ in same set then
        Place this record in Output Trace[I]
        else misses[I] := misses +1;
      Put record into Cache[I] and remove the LRU line;
      end;
    end;
end; {Simulate}
```

REFERENCES

Chandy, K. M. and R. Sherman, (1989). Space-Time and Simulation. *Distributed Simulation, 1989*. B. Unger and R. Fujimoto (eds.). The Society for Computer Simulation, 53-57.

Dwarkadas, S., J. R. Jump, and J. B. Sinclair, (1989). Efficient Simulation of Cache Memories. *1989 Winter Simulation Conference Proceedings*. E. A. MacNair, K. J. Musselman, P. Heidelberger (eds.). IEEE Press, 1032-1041.

Greenberg, A.G., Lubachevsky, B.D., and Mitrani, I. (1990). Unboundedly Parallel Simulations via Recurrence Relations. *1990 ACM SIGMETRICS Proceedings*. ACM Press, 1-12.

Hill, M. D. and A. J. Smith, (1989). Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers* **38**, 1612-1630.

Laha, S., Patel, J.H., and R.K. Iyer (1988). Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Transactions on Computers* **37**, 1325-1336.

Lin, Y. B., J. L. Baer, and E. D. Lazowska, (1989). Tailoring a Parallel Trace-Driven Simulation Technique to Specific Multiprocessor Cache-Coherence Protocols. *Distributed Simulation, 1989*. B. Unger and R. Fujimoto (eds.). The Society for Computer Simulation, 185-190.

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

Puzak, T. R. (1985). Cache Memory Design. Ph. D. Dissertation, University of Massachusetts, Amherst, MA.

Samples, A. D. (1989). Mache: No-Loss Trace Compaction. *1989 ACM SIGMETRICS and Performance '89 Proceedings*. ACM Press, 89-97.

Smith, A. J. (1982). Cache Memories. *Computing Surveys* **14**, 473-530.

Stone, H. S. (1987). *High-Performance Computer Architecture*. Addison-Wesley, Reading, MA.

Stunkel, C. B. and W. K. Fuchs, (1989). TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation. *1989 ACM SIGMETRICS and Performance '89 Proceedings*. ACM Press, 70-78.

Thompson, J. G. and A. J. Smith (1989). Efficient (Stack) Algorithms for Analysis of Write-Back and Sector Memories. *ACM Transactions on Computer Systems*, **7**, 78-116.