Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

STACK EVALUATION OF ARBITRARY SET-ASSOCIATIVE MULTIPROCESSOR CACHES

Y. Wu
R. Muntz

September 1993
CSD-930032

Stack Evaluation of Arbitrary Set-Associative Multiprocessor Caches*

Yuguang Wu and *Richard Muntz*
Computer Science Department
University of California, Los Angeles

Abstract

We propose a simple solution to the problem of efficient stack evaluation of LRU multiprocessor cache memories with arbitrary associative set-mapping. It is an extension of the existing stack evaluation techniques for all set-associative LRU uniprocessor caches. Special marker entries are used in the stack to represent data blocks (or lines) deleted by an invalidation-based cache coherence protocol. A method of marker-splitting is employed when a data block below a marker in the stack is accessed. Using this technique, one-pass evaluation of memory access trace yields hit ratios for all cache sizes and set-associative mappings of multiprocessor caches in a single pass over a memory reference trace. Simulation experiments on real multiprocessor trace data show an order-of-magnitude speed-up in simulation time using this one-pass technique.

Key Words—cache memory, coherence by invalidation, set-associative, simulation, stack evaluation.

1 Introduction

Trace-driven simulation is the most widely used evaluation method for performance studies of cache memories. It is more realistic and accurate than analytical modeling, and more efficient than software-driven system emulation[Chaiken 90]. For practical purposes, trace-driven simulation is viewed as a good approximation tool for comparing various cache memory designs. Stack evaluation is an efficient trace-driven simulation method for two-level cache memory hierarchies[Mattson 70]. It produces performance measures such as hit ratios for arbitrary cache sizes in a one-pass processing of a CPU memory access trace, eliminating case-by-case simulations for specific cache sizes.

When the least-recently-used (LRU) algorithm is used by the cache as its block replacement algorithm, a one-pass stack processing can produce hit ratios for all set-associative mapping schemes[Mattson 70]. We are interested in extending this set-associative stack evaluation technique for an LRU cache on an uniprocessor computer to LRU caches on a multiprocessor computer.

There are issues regarding the validity of trace-driven simulation for multiprocessor caches: perturbation to the trace data by the tracing mechanism, differences in trace data across different runs of the same program, and particular system configuration (e.g., cache sizes) under tracing. All these factors can potentially change the global execution order of the program(s) being traced, since they all affect the way processors interact with one another, including their relative order of arriving at some synchronization point, their relative speed in finishing their assigned jobs, and consequently their job scheduling.

It has been found[Koldinger 91] that there is insignificant difference in simulation results due to tracing perturbation, for both process-based coarse-grained parallel programs and thread-based medium-grained parallel programs; miss ratios did not vary much between different runs of a program, especially for coarse-grained programs. For stack evaluation, even though the actual ordering of memory access requests may be altered by the changes in exactly which references are cache misses due to the different cache sizes, the changes would be slight and their effect on simulation results insignificant[Smith 93]. Generally, stack evaluation methods are useful in predicting the general performance trend of cache memories, making a helpful tool in the early design stages. They can be used to narrow down the initially vast design space into a few, more manageable choices, which in turn can be studied using other less efficient but more accurate methods such as case-by-case simulations or

*This work has been supported in part by NSF grant CCR-9215064. A preliminary version of this report appeared in the proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing, Dec. 1992.

software-driven emulation.

On multiprocessors where each CPU has its own local cache, the issue of cache coherence arises. Like a uniprocessor cache, each cache still has to implement some block replacement algorithm, in case a referenced block is not in the cache and no cache space is available for it. Unlike a uniprocessor cache, however, multiprocessor caches have to interact among themselves by some cache coherence protocol to keep all the caches consistent[Dubois 82, Dubois 88, Stenström 90]. There are three categories of cache coherence protocols: immediate-coping, invalidation, and validation. Immediate-coping is the case where the updating cache broadcasts the changed data to all other caches[McCreight 84]. Invalidation, on the other hand, does not broadcast the data; instead the updating cache sends an invalidation message to all other caches holding a copy of the data block, and they discard their copies [Censier 78, Papamarcos 84, Archibald 86, Cheong 88, Li 89, Chaiken 90]. With the validation methods, the updating cache does not broadcast the new data nor sends an invalidation to other caches; before accessing any local data block, a cache must make sure that it has acquired the latest version of that data block. If the cache owns the block, then it is guaranteed to have the latest copy; otherwise, it has to contact the owner of the block (a remote cache or the shared memory) in order to get the latest version. Generally, immediate-coping is good for infrequent writes and high degree of data sharing, and invalidation is suitable for frequent writes and low data sharing. A validation-based protocol incurs high overhead and is deemed impractical.

For an immediate-coping protocol, a write to a block in one cache does not change the presence or absence of that block in another cache. Other CPU's writing will update a block if it is in cache–no effect if it is not. From the view-point of cache evaluation, a CPU's local cache is unchanged by reads or writes in other caches and stack evaluation can be applied independently for each cache. For a validation-based protocol, the situation is similar: the content of a CPU's local cache is independent of those of other caches and independent stack evaluation can be done for each cache.

The interesting case is an invalidation-based protocol, where a write in one cache results in blocks being invalidated (deleted) in other caches. In other words, an invalidation by a write in one cache produces an empty block frame in all other caches that have a copy of the changed data block. From the standpoint of stack evaluation, this effectively leaves an empty block in the stacks of the effected caches.

Mattson, et al. [Mattson 70] used a special marker entry "#" in the stack to represent an empty block frame caused by the invalidation of an I/O operation. All marker entries contribute to stack evaluation, and the invalidated block frames in the cache have the highest priority of being selected in replacement decisions. For fully-associative LRU caches, a marker will remain at the same position in the stack until another data block below it is accessed, at which time the marker is moved down to the stack position of the newly accessed data block, and the referenced data block is moved up to the top of the stack. For set-associative caches, whether a marker should move, when a data block below it is accessed, is dependent upon whether the empty frame (more precisely, the invalidated data block from which the empty frame was obtained) and the newly requested data block are in the same set. As two blocks can be in the same set for one associativity and in different sets for another associativity, the movement of a marker is not obvious for arbitrary set-associative mapping. This problem was proposed and its difficulty discussed by Wang and Baer[Wang 91].

In this paper we propose a solution to this problem. In section 2 previous stack evaluation techniques are reviewed, especially those that are related to LRU arbitrary set-associative stack evaluation on uniprocessor caches. In section 3 we present a new method for one-pass evaluation of multiprocessor caches with invalidation-based cache coherence protocols, yielding performance measures for all set-associative mappings. In section 4 we give simulation results of the method on some multiprocessing application trace data. Section 5 concludes with a summary.

2 Previous works

In this section we review some of the previous results in efficient cache memory evaluation techniques. The seminal work on this subject was by Mattson et al. [Mattson 70], who showed how certain block replacement algorithms, called *stack algorithms*, can have their hit ratios be calculated for arbitrary cache capacities in a single pass over a memory access trace. There have been several significant extensions to the original stack evaluation techniques. Gecsei extended it to multiple level memory hierarchies with different block sizes for LRU replacement and a special class of stack algorithms[Gecsei 74]. Thompson extended stack evaluation of general stack algorithms to write-back caches and sector caches[Thompson 87, Thompson 89]. Wang extended write-back techniques to LRU caches for all set-associative mappings[Wang 89, Wang 91].

2.1 Stack evaluation

A cache block replacement algorithm is called a *stack algorithm* if, while being used, the cache contents in a demand-fetched two-level hierarchy always satisfies an *inclusion* property; namely, the contents of a smaller cache is always a subset of that of a larger cache, for any access sequence[Mattson 70]. Several popular replacement algorithms including the LRU, least-frequently-used (LFU), and minimal (MIN, replacing the block that won't be accessed till the farthest future) are stack algorithms. A necessary and sufficient condition for a replacement algorithm to be a stack algorithm is, at any time t, there is a priority list P_t of all previously accessed data blocks, and if replacement is necessary, the block with the lowest priority is chosen[Mattson 70]. Let $X = x_1, x_2, \ldots, x_L$ be a memory access trace from the CPU, in which x_t is the block being accessed at time t. For stack replacement algorithms, there is an efficient procedure, called stack evaluation, that produces hit ratios for the entire range of cache sizes with one pass over the access trace X. As explained below, during the one-pass processing of X, some data on block usage is collected, and at the end hit ratios are derived from this data.

Due to the inclusion property, at any time t, the cache contents, for all cache capacities, can be represented succinctly by a list $S_t = [s_t(1), s_t(2), ..., s_t(\gamma_t)]$, where each $s_t(i)$ is a distinct block and γ_t is the number of distinct blocks accessed up to time t ($\gamma_t \leq t$). The contents of a cache with a capacity of C blocks at time t is just the first C elements of S_t. S_t is called the *stack* of the replacement algorithm at time t (Block $s_t(1)$ is the top of the stack). Under LRU, the stack S_t is the list of all blocks referenced up to time t, ordered according to most recent reference.

Let x_t be the block accessed at time t. The *stack distance* Δ_t for x_t is the position of block x_t in stack S_{t-1},

$$x_t = s_{t-1}(\Delta_t).$$

Δ_t is set to ∞ if x_t is not in S_{t-1}, which means that it has not been referenced before. x_t is resident in a cache of size C if and only if $\Delta_t \leq C$. The percentage of stack distances that are less than or equal to C during the stack evaluation of trace X is the cache's hit ratio for X.

To reflect the situation that x_t is just accessed, stack S_{t-1} has to be updated. Stack updating is achieved by pulling x_t to the top of stack, and switching subsequent stack entries according to the priority list P_t, as is shown in Figure 1.

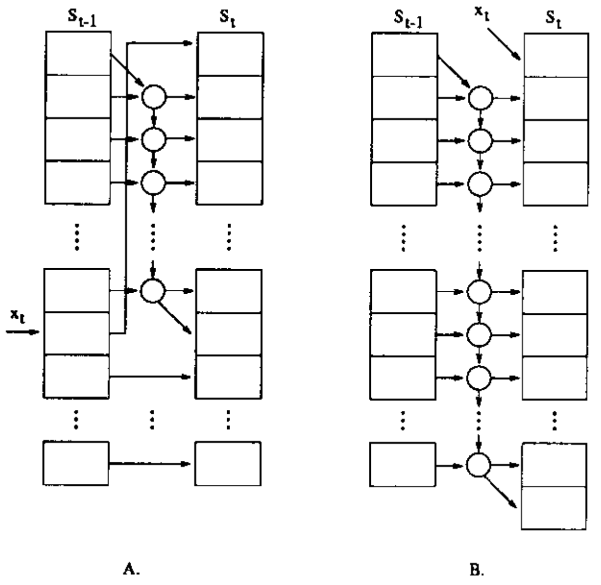


Figure 1: General stack updating. A. x_t is in S_{t-1}, B. x_t is not in S_{t-1}.

Stack updating of LRU at time t is simple: delete x_t from S_{t-1} if it is in S_{t-1}, and push x_t to the top of the new stack. This is shown in Figure 2.

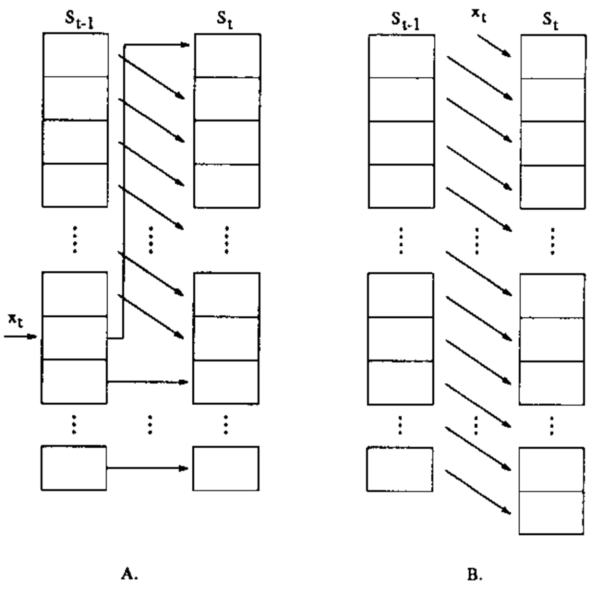


Figure 2: LRU stack updating. A. x_t is in S_{t-1}, B. x_t is not in S_{t-1}.

Let $n(\Delta)$ be a counter for the number of times that

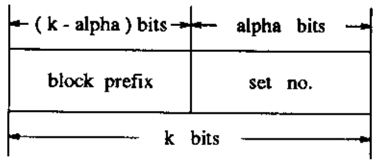


Figure 3: Block number

the referenced block was at stack distance Δ during trace evaluation. At each time point t, $n(\Delta_t)$ is incremented, so that at the end of trace processing, all counters will contain the proper counts.

The number of times an accessed block is found in a cache of capacity C is

$$N(C) = \sum_{\Delta=1}^{C} n(\Delta)$$

and the hit ratio is given by

$$P_{hr}(C) = N(C)/L$$

where L is the total length of the trace.

2.2 Set-associative mapping

So far we have assumed that a block can occupy any block frame in the cache memory, i.e., the mapping of data blocks in the cache is unconstrained. This is usually called *fully-associative* mapping. Unconstrained mapping has the disadvantage of having to search the entire cache each time a block is to be located, causing slow response to CPU's access request. A constrained mapping, called *set-associative* mapping, is used in practice to reduce the search time, whereby each block is restricted to a subset of the cache block frames[Mattson 70].

Assume that there are 2^k distinct blocks in the address space. One commonly used set-associative mapping, two's power congruence mapping, is to partition the 2^k blocks into 2^α disjoint sets of equal sizes, where $0 \leq \alpha \leq k$. We call α the *set length*, which indicates set-associative mapping. Each set has $2^{k-\alpha}$ blocks. The sets are numbered from 0 to $2^\alpha - 1$, and a block's set is determined from the α low-order bits of its block address, i.e. by $\bmod 2^\alpha$, as shown in Figure 3.

Usually the cache has an equal number D of block frames for all sets, with the cache's total capacity being $C = 2^\alpha \cdot D$. Such a cache is called D-way set associative cache; when $D = 1$, it is called direct-mapped, where each block has only one possible frame to reside in the cache. When a block x is accessed, a search is made of the D block frames of its associated set. If it is not there, and if all D block frames are occupied, a block replacement is made to remove one of these D blocks from the cache.

Since the sets are disjoint, the cache can be treated as a collection of 2^α independent caches, one for each set. The two-level cache-main memory hierarchy can also be viewed as a collection of 2^α independent hierarchies, each with a cache size of D. They can be handled separately using stack evaluation techniques.

For a general stack algorithm, stack evaluation technique must be applied individually to each value of the set length α. A total of $k + 1$ passes of trace evaluation are needed for all values of α between 0 and k. For LRU replacement, however, only a single pass trace evaluation can determine the hit ratios for all values of α.

2.3 LRU set-associative evaluation

Under LRU replacement, the stack S_{t-1} is the list of all the blocks referenced from time 0 to $t-1$ ordered according to most recent reference. Stack $S_{t-1}(i, \alpha)$ for set i and set length α can be determined by listing in order all the stack entries of S_{t-1} that belong to set i when the set length is α[Mattson 70].

Let $[x_t]_\alpha = x_t \bmod 2^\alpha$ denote the set that block x_t belongs to under set length α, and Δ_t^α the stack distance of x_t in the stack $S_{t-1}([x_t]_\alpha, \alpha)$. All stack distances $\{\Delta_t^\alpha\}$ can be determined in a single scan of the global LRU stack S_{t-1}[Mattson 70]. Suppose the current referenced block is x_t, and the jth entry y on S_{t-1} is being scanned: $y = s_{t-1}(j)$. Define the *right match function* $\mathrm{RM}(x_t, y)$ as the number of consecutive low-order matching bits in block addresses x_t, y. For example, RM(1010, 0110) = 2, RM(1000, 1001) = 0. The block y will affect Δ_t^α if and only if y is in the same set as x_t, that is, $\mathrm{RM}(x_t, y) \geq \alpha$.

Let $\{\mu(r)\}$ be a set of counters for $0 \leq r \leq k$. To determine $\{\Delta_t^\alpha\}$ for all α, one can scan down the stack S_{t-1} and increment counter $\mu(\mathrm{RM}(x_t, y))$ at each stack entry y. The scanning stops when x_t is found, and the stack distance Δ_t^α is given as

$$\Delta_t^\alpha = \sum_{r=\alpha}^{k} \mu(r) \qquad (1)$$

where $0 \leq \alpha \leq k$. If x_t is not found in S_{t-1}, Δ_t^α is set to ∞ for all set lengths. After each access x_t, the stack distance counter $n_\alpha(\Delta_t^\alpha)$ is incremented for each set length α.

3 Multiprocessor LRU set-associative evaluation

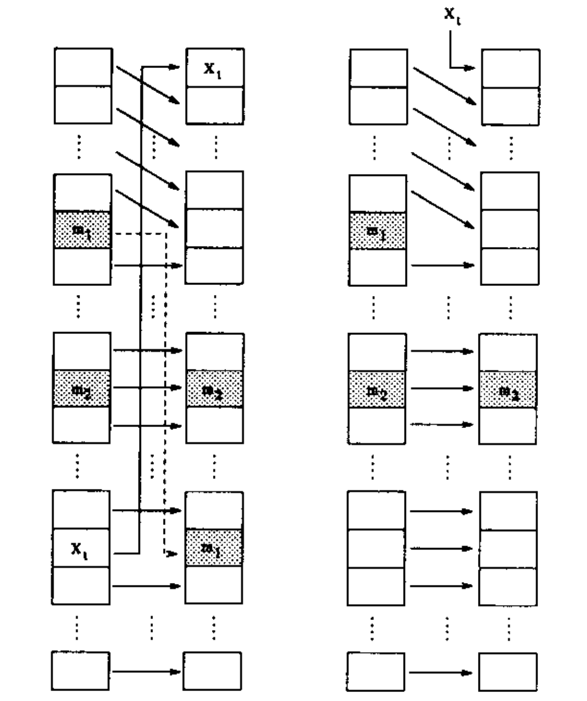
As with uniprocessor caches, we want to make a one-pass scan of the fully associative LRU stack S_{t-1} and get stack distance Δ_t^α of every set-associative mapping (or set length) α for multiprocessor caches. As mentioned earlier, an invalidated block in the stack is represented by a marker entry. If a data block below a marker in the stack is accessed, the marker may need to be moved down to a new stack location, depending on whether the accessed data block and the marker are in the same set. The newly accessed data block is always moved up to the top of the stack.

With fully-associative caches, the movement of a marker is simple. If there is one marker above the accessed data block in the stack, the marker is moved down to the stack location of the data block. If there are two or more markers above the accessed data block, then only the top-most one is moved down to the data block's stack location, while the other markers remain in their locations[Mattson 70, Thompson 87]. To update LRU stack S_{t-1}, we scan down the stack until x_t is found or the stack is exhausted, remembering the location of the first marker along the way. This is shown in Figure 4A. In case the accessed block is not in the stack, the top-most marker is removed from the stack, as is shown in Figure 4B. One can easily see it is the correct way to update a stack with markers, by considering each case where the cache size is at least as big as the stack position of the i-th marker in the stack but less than that of the $(i+1)$-th marker in the stack, before stack updating.

To update an arbitrary set-associative LRU stack with markers, We will employ a method called marker splitting.

3.1 Marker splitting

When a data block has just been invalidated, it becomes a marker that has presence in $(k+1)$ sets, for set length α ranging from 0 to k. These are the sets that the invalidated data block would be in for each set length. We represent each marker m with a 2-tuple (b, v), where b is the block address of the original data block that was invalidated, and v is a $(k+1)$ element vector whose elements are either 0 or 1: $v[i] = 0, 1$ for $0 \le i \le k$. We shall call b the *address*, and v the *covering vector*, respectively, of the marker. m represents a marker presence in set $(b \bmod 2^\alpha)$ of set length α if its $v[\alpha] = 1$. When a data block b is invalidated, it changes into a marker $m = (b, v)$



A. x_t is in stack m_1 moves down and m_2 stays.

B. x_t is not in stack m_1 is deleted and m_2 stays.

Figure 4: Constructing fully associative LRU stack with marker(s)

with v set to 1's: $v[i] = 1$ for $0 \le i \le k$, representing $(k+1)$ singular, indivisible markers in each set length. Let $|m|$ be the number of singular markers m is composed of: $|m| = \sum_{i=0}^{k} v[i]$. A marker is *composite* if it denotes multiple singular markers for different set lengths. Two markers are *disjoint* if the dot product of their covering vectors is zero, i.e. $\sum_{i=0}^{k} v_1[i] \times v_2[i] = 0$. Disjoint markers do not have presence in the same set for any set length.

When a data block in the stack is accessed, and there is one marker m above it in stack, with $m = (b, v)$, then m is split into two disjoint markers $m' = (b, v')$ and $m'' = (b, v'')$, with $|m| = |m'| + |m''|$. m' represents the original marker's presence in the $|m'|$ sets that do not contain the data block, while m'' denotes the original marker's presence in the $|m''|$ sets that contain the currently accessed data block. m' replaces m, and m'' is moved down the stack to where the data block resides; this is illustrated by Figure 5A. If either m' or m'' is empty ($|m'| \cdot |m''| = 0$), there is no splitting: m either moves down or stays where it is, depending on which one of m' and m'' is empty.

If, when a data block in the stack is accessed, there

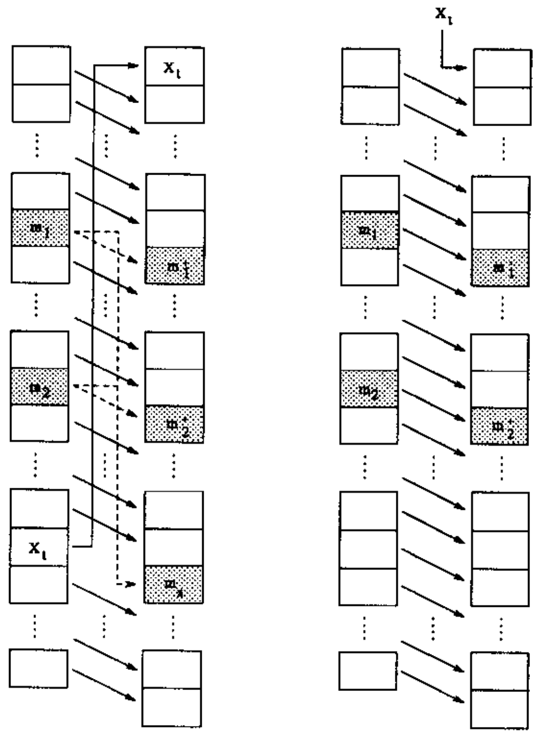


Figure 5: Constructing set-associative LRU stack with marker splitting

are multiple markers above it in the stack, the situation becomes a bit more complex. Unlike the fully associative case, where we only need to remember one top-most marker, here we have to remember the top-most *singular* marker for each set length. Since each marker above the accessed block in stack might contain some singular marker that is in a same set as the data block for some set length, we must examine each one of them and remember the top-most singular marker for each set length along the way. Remember that for each set length, we only want to move down the top-most singular marker, while keeping other singular markers in place.

Details of the stack updating procedure is given in section 3.2. Here we summarize the three possible ways of adjustment of a marker which is above the accessed data block in the stack:

- marker remains unchanged and in the same location; i.e., the marker and the referenced block are not in a same set for any set length, or for any set length that they do share the same set, there was a singular marker above in the stack.
- marker remains unchanged but is moved down to the data block's stack location; i.e., the marker and the referenced block are in the same set for all set lengths which the marker has a valid singular marker, this singular marker is the top-most one in the stack for its set length.
- marker gets split into two disjoint markers; one of them remains in the same location, the other is moved down to the data block's stack location.

The outcome depends on the addresses of the data block and the marker, the covering vector of the marker, and the existence of other markers that are above the current marker in the stack.

3.2 Stack updating

Suppose there is only one marker $m = (b, v)$ above the accessed block x_t in the stack S_{t-1}. Let $r = \mathrm{RM}(x_t, b)$. If $v[i] = 0$ for all $0 \le i \le r$, m is a non-existent marker to block x_t (m'' is empty), and hence can be treated as a normal data block. If $\exists\ j$ with $0 \le j \le r$ and $v[j] = 1$, but $v[i] = 0$ for all $r < i \le k$, then m is in the same set as x_t for every relevant set length (m' is empty); in this case m is moved down to the stack location of x_t. If $\exists\ i, j$ such that $0 \le i \le r < j \le k$ and $v[i] = 1 = v[j]$, then m is replaced by two disjoint markers $m' = (b, v')$ and $m'' = (b, v'')$, whereby

$$v'[i] = \begin{cases} 0 & \text{if } 0 \le i \le r \\ v[i] & \text{if } r < i \le k \end{cases}$$

$$v''[i] = \begin{cases} v[i] & \text{if } 0 \le i \le r \\ 0 & \text{if } r < i \le k \end{cases}$$

m' replaces m in the stack, and m'' is moved down to that of x_t, as shown in Figure 5A.

If x_t is not in stack S_{t-1}, m' still replaces m, but m'' is thrown from the stack. This is shown in Figure 5B.

If there are multiple markers above an accessed block in the stack, then for each set length, the top-most singular marker that shares the same set with the accessed block needs to be moved to the location of the data block. All the top-most singular markers in stack S_{t-1} are collected during the search for x_t and, upon finding x_t, they are placed at x_t's location. The general procedure is: search stack S_{t-1} for the accessed data block x_t, and whenever a marker is found, be it composite or singular, check if it contains any singular marker that shares the same set with x_t for a set length, for which a singular marker has not been seen yet. If there are no such singular markers, leave the current marker intact. Otherwise, collect all eligible singular markers, and remove them from the current marker; if the current marker subsequently becomes empty, delete this marker entry from the stack.

Finally, when the data block x_t is found, all the collected singular markers are grouped into one composite marker and it replaces x_t in the stack, while x_t is moved to the top of the new stack S_t. Since these singular markers share some set with x_t, and these sets are for different set lengths, each of them can be represented by a marker (x_t, v); the covering vector v has only one non-zero element (of value 1), whose index is the set length for which the represented singular marker shares the same set with block x_t. Therefore they can be put into one composite marker (x_t, v), whose covering vector is simply the sum of those covering vectors of the collected singular markers.

If the data block x_t is not in stack S_{t-1}, all collected singular markers are discarded.

The LRU stack updating procedure for stack S_{t-1} while scanning stack for $x = x_t$ is as follows:

Stack updating procedure

Let w be a $k+1$ element integer vector initialized to all zero.

Scan down the stack S_{t-1} and do the following at each entry $E = s_{t-1}(j)$ until $x = x_t$ is found or S_{t-1} is exhausted:

If $E = (b, v)$ is a marker, let $r = \mathrm{RM}(x, b)$, $U = \{\ i \mid (0 \le i \le r)\ \wedge\ (v[i] = 1)\ \}$, $W = \{\ i \mid (0 \le i \le r)\ \wedge\ (w[i] = 0)\ \}$. Change E's covering vector v into

$$v[i] = \begin{cases} 0 & \text{if } i \in U \bigcap W \\ v[i] & \text{otherwise} \end{cases}$$

Change vector w into

$$w[i] = \begin{cases} 1 & \text{if } i \in U \bigcap W \\ w[i] & \text{otherwise} \end{cases}$$

Leave marker E where it is if v is not an all-zero vector; remove E from the stack if v is all-zero.

If E is a data block but $E \neq x$, continue.

If E is a data block and $E = x$, then replace $s_{t-1}(j)$ by a new marker $m = (x, w)$ if w is not all zero, and put block x on top of the stack. Break out of the loop.

If S_{t-1} is exhausted but x is not in the stack, throw away the collected vector w, and just pull x to the top of the stack.

The updated stack S_{t-1} is S_t.

Example

There are two markers above a referenced data block $x = 0101$ in the stack. The first marker is $(b_1, v_1) = (1101, 11001)$, and the second marker is $(b_2, v_2) = (1001, 10011)$. All vectors are listed with their highest element first. For example, $v_1[4] = 1, v_1[3] = 1, v_1[2] = 0, v_1[1] = 0, v_1[0] = 1$. $\mathrm{RM}(x, b_1) = 3$, $\mathrm{RM}(x, b_2) = 2$. Initially $w = 00000$. After scanning b_1, $b_1 = (1101, 10000)$, $w = 01001$. After scanning b_2, $b_2 = (1001, 10001)$, $w = 01011$. When x is reached, x is moved to the top of the stack, and a new marker $(0101, 01011)$ is put in x's previous slot.

Theorem 1 *The stack updating procedure correctly places markers for all set lengths.*

Proof. It is clear, from the stack updating procedure, that the vector w records which set lengths have already had a marker in the same set as x. That is, for each set length α with $0 \le \alpha \le k$, $w[\alpha] = 1$ if and only if we have already seen at least one marker in the stack belonging to its set $[x]_\alpha$. The changing of the covering vector v of each marker $E = (b, v)$ encountered by x in the stack is to take away all the *first* markers that are in the same set with block x for some set length. The change to the vector w is to add those newly found *first* markers. If x is found in the stack, then these first markers should all be moved to the location of x, which is what the replacement of x by $m = (x, w)$ does in the procedure. If x is not found in the stack, then all these first markers should be removed from the stack. □

3.3 Stack distance counting

We have just shown how to update the global LRU stack. In this section we discuss how to calculate stack distances for all set lengths. These two operations, stack updating and stack distance counting, are in fact carried out simultaneously during stack scanning.

First let us define a couple of simple and useful functions. The *trailing one function* $\mathrm{TO}(v, k)$ on vector v is the set of indices of the *last* element in each string of consecutive 1's in v that are less than or equal to k. That is, $\mathrm{TO}(v, k) = \{\ i \mid (v[i] = 1) \wedge (((i < k) \wedge (v[i+1] = 0)) \vee (i = k))\}$. For example, $\mathrm{TO}(\{10010110\}, 7) = \{2, 4, 7\}$, and $\mathrm{TO}(\{01010111\}, 7) = \{2, 4, 6\}$.

Define the *trailing zero function* $\mathrm{TZ}(v, k)$ on vector v to be the set of indices of the *last* element in each string of consecutive 0's in v that are less than k. That is, $\mathrm{TZ}(v, k) = \{\ i \mid (i < k) \wedge (v[i] = 0) \wedge (v[i+1] = 1)\}$. For example, $\mathrm{TZ}(\{10010110\}, 7) = \{0, 3, 5\}$, and $\mathrm{TZ}(\{01010111\}, 7) = \{3, 5\}$.

Let $\{\mu(r)\}$ and $\{\nu(r)\}$ be two groups of counters for $0 \le r \le k$. To determine $\{\Delta_t^\alpha\}$ for all α, scan down the stack S_{t-1} until $x = x_t$ is found or the stack is exhausted. Suppose the current stack entry being examined is the jth entry in stack S_{t-1}: $E = s_{t-1}(j)$. If E is a data block, increment counter $\mu(\mathrm{RM}(x, E))$. If E is a marker (b, v), consider values of $r = \mathrm{RM}(x, b)$ and $v[i]$'s. If $v[i] = 0$ for all $0 \le i \le r$, marker E is non-existent to block x, do nothing. Otherwise, increment $\mu(i)$ for all $i \in \mathrm{TO}(v, r)$ and $\nu(j)$ for all $j \in \mathrm{TZ}(v, r)$. If x is found in the stack S_{t-1}, then each stack distance Δ_t^α is given by

$$\Delta_t^\alpha = \sum_{r=\alpha}^{k} (\mu(r) - \nu(r)) \qquad (2)$$

where $0 \le \alpha \le k$. If S_{t-1} is exhausted and x is not found, all stack distances Δ_t^α are set to ∞. As in the uniprocessor case, the stack distance counter $n_\alpha(\Delta_t^\alpha)$ is incremented for each set length α.

Numerical example

There are three data blocks and two markers above a referenced data block $x = 0101$ in the stack. The three data blocks are $x_1 = 0111, x_2 = 0000, x_3 = 0001$. The markers are $(b_1, v_1) = (1101, 11001)$, $(b_2, v_2) = (1001, 10011)$. The order of their appearance on the stack is, from top down, $x_1, b_1, x_2, b_2, x_3, x$. As before, all vectors are listed with their highest element first. Initially, $\mu = 00000, \nu = 00000$.

scanning x_1: $\mathrm{RM}(x, x_1) = 1$, $\mu = 00010$, $\nu = 00000$.
scanning b_1 : $\mathrm{RM}(x, b_1) = 3$, $\mu = 01011$, $\nu = 00100$.
scanning x_2: $\mathrm{RM}(x, x_2) = 0$, $\mu = 01012$, $\nu = 00100$.
scanning b_2 : $\mathrm{RM}(x, b_2) = 2$, $\mu = 01022$, $\nu = 00100$.
scanning x_3: $\mathrm{RM}(x, x_3) = 2$, $\mu = 01122$, $\nu = 00100$.

When x is reached, the stack distances are, according to Equation (2),

$$\Delta^0 = 5, \Delta^1 = 3, \Delta^2 = 1, \Delta^3 = 1, \Delta^4 = 0.$$

Theorem 2 *Equation 2 correctly computes the stack distances* Δ_t^α.

Proof. When the stack entry is a data block, Equation (1) applies. So we only need to consider the case where the current stack entry is a marker. We depict in Figure 6 the covering vector of a marker $E = (b, v)$ encountered during stack scanning for block x, with the left-most rectangle representing the kth vector element and the right-most one representing the 0th vector element. A shaded rectangle denotes an element of value 1, and an unshaded one denotes an element of value 0. Here $r = \mathrm{RM}(x, b)$. Notice that all rectangles indexed from r up to k are all unshaded (zero elements), since they are irrelevant to x.

It is clear that marker E represents an empty entry in the same set as x for set length α (i.e., $[x]_\alpha$) if and only if the αth rectangle in the figure is shaded. Thus exactly these set lengths, whose corresponding element in the figure is shaded, should increment their respective stack distance counters by one. For the specific depiction in Figure 6, these set lengths are $a, i, p, p-1$, and 1.

Suppose at first $\mu(r) = \nu(r) = 0$ for each r. Then after evaluating the first marker,

$$\sum_{r=\alpha}^{k} (\mu(r) - \nu(r))$$

is equal to 0 if the αth rectangle is not shaded, and is equal to 1 if it is shaded. Generally, suppose a new referenced block x has been searched in the stack, and $\mu^+(r), \nu^+(r)$ are the new counter values. By the same reasoning, it follows that

$$\sum_{r=\alpha}^{k} (\mu^+(r) - \nu^+(r)) - \sum_{r=\alpha}^{k} (\mu(r) - \nu(r)) = 0$$

if the current αth rectangle is not shaded, and $= 1$ if it is shaded. This means Equation (2) correctly counts stack distance when the stack entry is a marker. □

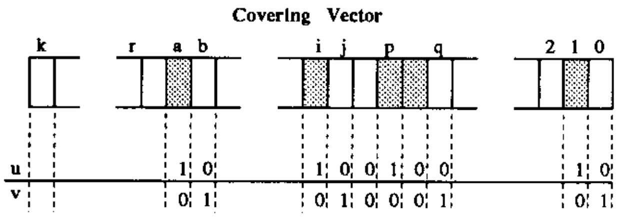


Figure 6: Counting of $\mu(r), \nu(r)$

The stack distance contributed by markers for each set length α can be counted directly during stack scanning. Let $\lambda(\alpha)$ be the stack distance counter for markers of set length α. When E is a data block, increment counter $\mu(\mathrm{RM}(x, E))$ as before. When E is a marker (b, v), increment $\lambda(\alpha)$ for all $\alpha \in \{\, i \mid (0 \le i \le \mathrm{RM}(x, b)) \wedge (v[i] = 1)\}$. For Figure 6, counters $\lambda(a), \lambda(i), \lambda(p), \lambda(p-1), \lambda(1)$ are incremented. We have another way to calculate the stack distances:

Theorem 3 *The stack distance* Δ_t^α *is also given by*

$$\Delta_t^\alpha = \sum_{r=\alpha}^{k} \mu(r) + \lambda(\alpha) \qquad (3)$$

Proof. Incrementing of counter $\lambda(\alpha)$ remembers the number of times the αth rectangle is shaded during the stack scanning process. This is exactly the number of times an empty entry has appeared in set $[x]_\alpha$ before x is found. Hence, counting both data blocks and empty entries, the sum of the right-hand side of equation (1) and $\lambda(\alpha)$ gives the total stack distance of x for set length α. $\Box$

Numerical example

We use the same example. Initially, $\mu = 00000, \lambda = 00000$.

scanning x_1: $\mathrm{RM}(x, x_1) = 1$, $\mu = 00010$, $\lambda = 00000$.
scanning b_1 : $\mathrm{RM}(x, b_1) = 3$, $\mu = 00010$, $\lambda = 01001$.
scanning x_2: $\mathrm{RM}(x, x_2) = 0$, $\mu = 00011$, $\lambda = 01001$.
scanning b_2 : $\mathrm{RM}(x, b_2) = 2$, $\mu = 00011$, $\lambda = 01012$.
scanning x_3: $\mathrm{RM}(x, x_3) = 2$, $\mu = 00111$, $\lambda = 01012$.

When x is reached, the stack distances are given by Equation (3) as

$$\Delta^0 = 5, \Delta^1 = 3, \Delta^2 = 1, \Delta^3 = 1, \Delta^4 = 0.$$

This is the same result as before.

All the auxiliary functions $\mathrm{TZ}(v, k)$, $\mathrm{TO}(v, k)$ and $\mathrm{RM}(x, b)$ used in stack distance counting and stack updating are simple operations. So the time spent in processing a marker stack entry is not much more than that spent in processing a data block entry.

3.4 Time complexity

Our one-pass evaluation saves time in two aspects: the trace data needs to be read only once, and the stack processing is only done once for each access. What we do with arbitrary set-associative evaluation is use one composite marker to represent all $k+1$ possible singular markers, and split it when we have to. The number M, of markers in the stack, is bounded by $M \leq (k+1)I$, where I is the number of effective block invalidations, the invalidations that actually find their target blocks in the stack. Under the assumption that the rate of actual invalidations is not high in real applications, the number of markers produced in the stack will not be very large. Once a marker becomes singular, it will not split further. Therefore the stack does not grow indefinitely because of marker splitting. In addition, markers never ascend in the stack; they tend to descend in the stack as data blocks below them in the stack are accessed. Whenever a marker becomes the last entry of the stack, it can be dropped. The extra work needed in arbitrary set-associative evaluation is simple; the vector operations in singular marker collection and stack updating can be done with efficient bit operations on the simulating machine.

As the method requires sequential scanning of all stack entries above the accessed block in the stack, it defies efficient search data structures such as balanced trees for the representation of the stack. However, as we will see later in simulation experiments, with the exception of general hash tables which can be used by almost any stack evaluation method, sophisticated data structures such as search trees do not noticeably reduce the overall simulation time of a stack evaluation method[1]. This is because the locality property of CPU access produces on average short stack distances, making linear search of the stack quite inexpensive. Moreover, if most references are near the top of the stack, stack searching does not "see" many of the markers on most references.

4 Simulation

We have implemented both the arbitrary set associative evaluation algorithm and the conventional single set associative evaluation algorithm, in order to compare their performances. In this section we report simulation experiments on some real multiprocessor trace data. We are mainly interested in the comparison of simulation times in getting stack distance distributions for all set lengths on a given block size. The characteristics of the trace data are given, followed by a detailed description of algorithm implementations. The simulation results are given in the end.

4.1 Trace data

Three traces of parallel applications are used: Weather, Simple, and FFT. They were obtained using the IBM postmortem scheduling method and represent a possible execution on a 64-CPU multiprocessor[Cherian 89, Chaiken 90]. The Weather application partitions the earth atmosphere into a three dimensional grid and uses finite-difference methods to solve a set of partial differential equations describing the system state. The Simple application models the behavior of fluids and also uses finite difference methods to solve equations on hydrodynamic behavior. FFT is a radix-2 fast Fourier transform application. Each reference record consists of a one-byte CPU number (ranging from 1 to 64), a one-byte operation code (for data/instruction read/write), and a four-byte memory address. The length of each trace, i.e., the number of references, is respectively 7461123(FFT), 27172624(Simple), and 31777053(Weather).

[1]Thompson first observed this phenomena while comparing different data structures in the implementation of stack evaluation of uniprocessor write-back caches[Thompson 87].

4.2 Implementation

We will compare the run-time efficiency of the one-pass evaluation algorithm for all set-associative mappings with that of conventional multiple-pass evaluation algorithm for a single set-associative mapping. The data and instruction accesses are "unified"; i.e., we treat them as being cached together. To obtain fair and convincing results, we tried to make each implementation of an algorithm run as fast as possible.

Preliminary tests showed that, for the single set-associative algorithm, an implementation of the linked-list stack structure without using hashing ran significantly slower. So we applied the hashing technique in all the implementations and do not consider any non-hashing implementation for performance comparisons.

4.2.1 Single set-associative algorithm

For a set-associative cache, space is partitioned according some congruence set-mapping scheme; different sets are independent of one another. This translates to one (sub)stack for each set. So the conventional single set-associative algorithm maintains as many stacks as there are different sets in the trace data. Of course, the total number of distinct blocks (hence stack entries for valid data blocks) is the same regardless of the associativity.

One simple technique for efficient stack simulation is to maintain a hash table of all data blocks currently residing in the stack[Thompson 87]. It helps eliminate fruitless searches for blocks not even in the stack. Hashing proves to be very effective on uniprocessor traces[Thompson 87].

We use a two-level hashing table to hold all *valid* data blocks currently in stacks. It should provide faster look-up than one-level hashing, particularly when the number of distinct blocks is not small. At the beginning of each stack search, the program first determines whether the referenced block is currently in the corresponding stack (by looking in the hash table), and whether there are any marker entries in the stack (by checking a counter variable). If neither holds, then there is no need to search the stack.

The two-level hashing table, shown in Figure 7, is organized for a set length α under study. For a block number b, its associated set number $s = b \bmod 2^{\alpha}$ is used to probe the first level to find the set of the block, and its block number b to probe the second level to find the block itself. The reason for using a hash table on set numbers is as follows: when the value of α is not trivial, for example $\alpha = 20$, the number of

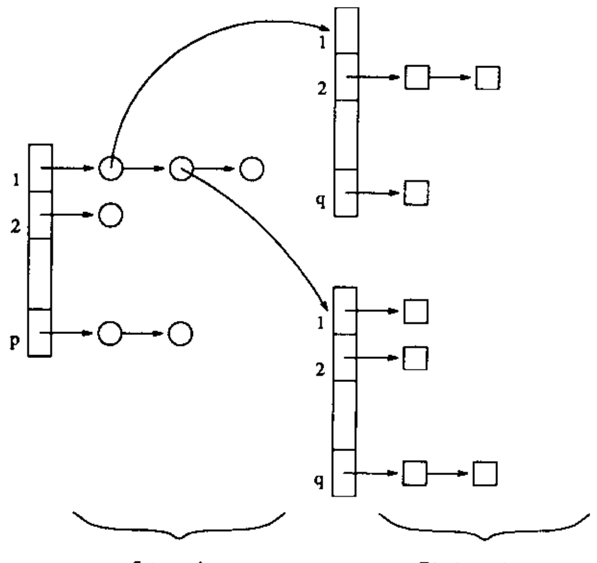


Figure 7: Two-level hash table

different sets is not small. An array indexed by set numbers requires a lot of space, some of which may be unused; a dynamically allocated linked-list has a relatively long search time.

At the first level, there is an array of p pointers, each one of which points to a list containing distinct set numbers with equal value mod p. The set number lists dynamically grow when new set numbers are encountered in trace. It is more efficient than static allocation of hash table entries for all possible sets; some application traces might not utilize all possible sets, especially for big values of set length α. Not shown in the figure, each element in the set lists has a pointer to the substack (implemented with either a list or a tree, see section4.2.2) of the set whose set number is contained in this element.

At the second level, each element in the set lists contains q pointers, each of which points to a list of elements containing block numbers with equal value mod q. These block number lists dynamically grow (when a new data block is referenced) or shrink (when an existing data block is invalidated) during trace processing. When a valid data block in a stack gets zapped, its corresponding block element is deleted from the second level in the hash table.

As stated before, any marker at the tail of the stack is void and is promptly dropped by all implementations. This eliminates unnecessary memory consumption and improves algorithm performance.

To save space, instead of using an array of counters with fixed dimension and having a lot of zero elements, stack distance counters are also dynamically implemented with a linked-list, sorted with increasing stack distance value. Even though incrementing a counter is no longer done in constant time now, thanks to locality in memory reference, stack distances tend to be small, and the time spent in looking for the right counter is negligible. We found virtually no difference in execution time of simulation whether array counters or link list counters are used.

4.2.2 Data structures for stack implementation

The most natural data structure for a stack is a linked-list of entries, where the stack updating procedure is readily carried out by entry deletion from the middle of list and entry addition to the head of list. The program for a linked-list stack is a simple one.

The potential drawback with a list is its linear search time; this might be significant for traces such as data base applications with long average stack distance. However, as we will see in our experiment results, a simple linked-list competes well with other complex data structures; thanks to reference locality, the referenced data block tends to be close to the stack head.

Sophisticated data structures with lower asymptotic time complexities such as binary search trees can be used to implement the stack. Bennett and Kruskal used the leaves of a fixed-structure sparse tree to represent stack entries, and Olken used an AVL balanced search tree to represent stack entries with both the external and internal tree nodes (see [Thompson 87]).

In a binary search tree stack implementation, all data blocks in the left subtree of any node are higher in the stack, while those in the right subtree are lower in the stack. The embedded stack order is the *inorder* traversal of the tree. The tree node containing the currently referenced data block can be quickly found with the hashing table, in which each element in the block lists has a pointer to the corresponding tree node. The stack distance of a data block can be found by walking up the tree to the root and counting the number of tree nodes to the left along the way. This can be done by storing in each tree node the number of nodes in its left subtree[Thompson 87]. Alternatively one can store in each node the number of nodes in the subtree with itself as the root.

In order to know if there is any marker on the left (i.e., ahead in the stack) while walking up the tree from the currently accessed node, each node also stores the number of marker nodes in its left subtree. Finding the left-most (i.e., top-most in the stack) marker node requires a walk down from the root in the tree.

Stack updating is achieved through normal node deletion and insertion in binary search trees; only here node insertion always occurs on the left end of the tree, which corresponds to the stack head.

Because of the extremely biased node insertion, the search tree can quickly degenerate into a linear list. In fact, it becomes the reversed stack, performing much worse than a simple linked-list stack which benefits from reference locality. A search tree with rebalancing is desirable. Among the many kinds of search trees, three are considered: AVL tree[Wyk 88] and red-black tree (also called 2-4 tree)[Guibas 78, Wyk 88] are balanced trees, and splay tree[Sleator 85] is a self-adjusting tree.

It was observed[Guibas 78] that AVL and red-black trees have similar performance for basic operations (node rotation) on some sequence of 20,000 random accesses. There was no comprehensive performance comparison for splay trees. We implemented these three data structures and informally tested them with some random input data. For short sequences of random accesses, red-black tree performs the best and is twice as fast as AVL tree; for a long sequence of 200,000 random accesses, the splay tree is the fastest, while AVL tree remains the slowest.

It has been proven[Sleator 85] that, in terms of *amortized time*, which is defined as the time per operation averaged over a *worst-case sequence* of operations, a splay tree is within a constant factor as efficient as any uniformly balanced tree and any fixed search tree for a sufficiently long sequence of accesses; more interestingly, the time to access an item is approximately the logarithm of one plus the number of distinct items accessed since the last time the given item was accessed[Sleator 85]. Based on the theoretical results and our preliminary experiments, we choose the splay tree to implement the stack.

Splay tree

The splay tree a self-adjusting binary search tree. The central idea is splaying, a restructuring heuristic that moves a designated node to the root of a tree through a series of rotations which approximately halves the depths of all nodes along the path. All tree operations, including access, insertion, and deletion, are implemented using splaying[Sleator 85]. Splaying can be done both bottom-up and top-down. Bottom-up is appropriate if there exists direct access to the node at which splaying is to occur, while top-down if efficient for a to-be-splayed node which has to be searched from the root. Details are described in [Sleator 85].

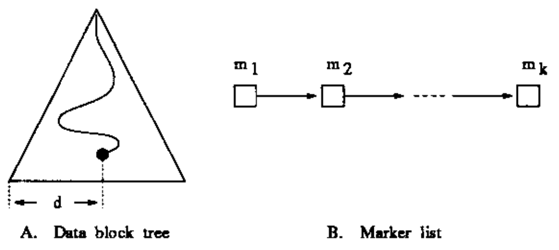


Figure 8: Stack distance calculation

We will use the bottom-up splaying for the data block obtained through hashing, and top-down splaying for the left-most marker node in the tree.

Separate marker-list

If there is any marker node to the left of the currently accessed data block in the tree, a top-down search has to be initiated to locate the left-most marker node. Olken[Thompson 87] suggested the use of a separate list to store the markers, sorted by the last access time; the tree at any time only contains valid data blocks. The benefits of this approach are that locating the top-most marker in stack is a constant time operation, and that it is easy to check whether there is a marker above the currently referenced data block by simply comparing their last access times. The drawback is slow invalidation: when a data block is zapped by an invalidation, before we only need to locate the data block node in the tree by hashing and change its flag to make it a marker, taking almost constant time; now we have to delete the node from the tree, and put it into the proper position (by sorted last access time) in the marker-list, not a constant time operation anymore.

We implemented the stack with splay tree using both approaches. There is a better way to implement the separate marker-list, though. Instead of using last access time, we use, equivalently, the actual *stack position* of a marker in the stack as its sorting key in the marker-list. As the stack distance has a much smaller value than the (potentially unbounded) trace length, the space needed to hold a key is less.

The stack updating affects the mark-list in the following way: when a data block is accessed which is not in the stack, then the first marker (if any) in the marker-list is thrown away; when a valid data block is accessed, and its stack distance is bigger than that of the first marker, then the stack distance of this first marker is set to the data block's stack distance, and moved down the marker-list to its proper new position, keeping the list ordered by distance keys. Clearly, the stack distances of all other markers are unchanged.

The remaining question is how to calculate the stack distance of a valid data block, since walking in the tree can only tell how many nodes are to its left in the tree, i.e., how many valid data entries are above it in the stack. Suppose that a data block is the d-th node in the inorder traversal of the tree, and the marker-list has elements with stack distances $m_1 < m_2 < \ldots < m_l$, we want to find the real stack distance sd for this data block. Let $m_i < sd < m_{i+1}$, then among the first sd stack entries, i of them are marker entries, and the remaining $(sd - i)$ are data block entries, as shown in Figure 8. We know there are exactly d data blocks in the range, hence $sd = d+i$. From $m_i < sd = d + i < s_{i+1}$, we have

$$m_i - i < d \leq s_{i+1} - (i+1),$$

the criterion for finding i.

4.2.3 Arbitrary set-associative algorithm

From the description of the arbitrary set-associative algorithm before, one clearly needs to collect the relevant dirty level variables for all the set lengths from higher entries in the global stack. Each entry above the currently accessed block entry in global stack may contain some of those required dirty levels, and has to be examined. A linear list is therefore a natural data structure choice for the global stack. Implementation of this algorithm is simple and similar to the linked-list implementation of single set-associative algorithm, but using one global stack instead of many substacks. It involves slightly complicated bit-vector manipulation when the current stack entry under scanning is a marker. Every 0/1-bit vector (such as covering vectors) is just an integer variable, taking up little extra memory than the single set-associative algorithm. The vector operations in the algorithm are done with concise and efficient bit-wise operations of the C programming language.

As above, a two-level hashing scheme is used to at the beginning of each stack search to quickly check whether the referenced data block is in the stack. Since there is no specific set-associative mapping here, we pick an arbitrary hashing function, instead of a congruence set-mapping scheme, for hashing at the first-level. To the arbitrary set-associative algorithm which only processes a trace only, we believe the choice of a hashing function is not crucial.

Stack distance counters are stored in linked-lists, with one list per set length, for up to a maximum of 33 lists. As before, markers are dropped as soon as there is no valid block entries below them in the stack.

4.3 Simulation results

The stack simulation algorithms have simple logic and control flow, and execution time is mostly spent in memory manipulation and data input, not complex CPU operations. Using the Unix code-profiling tool *gprof* indicates that for the arbitrary set-associative algorithm, its disk I/O accounts for about half of the entire running time. Since all trace files have the same data format, and reading trace data is an integral part of any trace-driven simulation, we count I/O time as part of the entire simulation time. The ever-increasing disparity of speed among CPU, main memory, and I/O can make I/O become a more important factor in trace-driven simulation.

4.3.1 Output data

All the simulation programs produce exactly the same output, i.e., stack distance distributions, on all input traces and various block-size specifications, verifying not only the correctness of our one-pass algorithm, but also that all the implementations for single set-associative evaluation using different data structures are done right.

4.3.2 Memory

From the description of arbitrary set-associative algorithm, it is clear that its memory consumption is just a little more than that for the linked-list stack structure implementation of single set-associative algorithm. Specifically, each stack entry uses one more integer field (4 bytes) to hold the covering vector. Using the Unix command *top*, we find that the total program size (code + data + stack) of the arbitrary set-associative algorithm is approximately the same as that of the single set-associative algorithm with the linked-list stack implementation.

The splay-tree stack implementations of single set-associative algorithm have three more fields per each stack entry than the linked-list implementation (one more pointer field and two more integer fields). Their run-time memory size is about 20% more than their linked-list counterpart.

The number of marker entries in the arbitrary set-associative program can also be more than that in the single set-associative program. But experiments show the number of extra marker entries in the algorithm is quite insignificant. One reason is that we keep dropping the marker at the tail of the stack, to prevent their number from growing; another reason: the number of markers might be scarce anyway.

Table 1: Simulation times on FFT (including i/o)

b	mul	sin.link	sin.splay	sin.mlist
13	86.9	764.4	764.0	765.8
12	86.1	800.3	803.3	805.1
11	87.9	853.0	849.2	866.9
10	91.8	899.3	904.3	901.7
9	96.7	929.9	936.6	940.5
8	100.2	1001.7	991.2	985.5
7	108.3	1017.1	1023.1	1030.3
6	107.4	1064.8	1071.1	1087.1
5	113.4	1111.8	1120.7	1147.0
4	123.4	1171.8	1176.8	1204.8
3	118.2	1249.2	1250.0	1301.8
2	111.7	1297.7	1314.7	1317.8
1	116.1	1373.1	1393.0	1395.1

Block-size = 2^b bytes. Run-time in seconds.

4.3.3 Running time

Simulations were run on a lightly loaded Sun SPARC-Station 10. As there is little disturbance from other activities on the machine, the measurements were stable and repeatable. We use the Unix *time* command to measure the execution times of each simulation, and the real times are very close to the sums of user times and system times, due to light load on the test machine. The arbitrary set-associative algorithm runs much faster than the single set-associative algorithm on all three traces.

Tables 1, 2, 3, and 4 illustrate the running times of the various implementations of the algorithms for the three traces. These results are from tests done for a typical CPU. For each trace, we randomly selected a number of CPUs and did stack simulation on them; their results were nearly identical. It is probably due to the fact that the traces were produced in a very symmetrical way (see [Cherian 89]).

The tests are done for a variety of block sizes. The first columns (b) indicate the base-2 logarithmic values of block sizes. The second columns (mul) are the running times of the arbitrary set-associative algorithm; rest columns are the running times of the single set-associative algorithm implemented with various data structures (sin.link for linked-list, sin.splay for splay tree, and sin.mlist for splay tree with separate marker list).

Tables 1, 3, and 4 include the disk i/o time of trace-reading in the total simulation time for the three traces, and Table 2 excludes that from the simulation time of FFT trace. Compare Table 1 and Table 2, we see that i/o played a small role in the simulations. Overall, our arbitrary set-associative algorithm ran approximately ten times faster than all implementations of the single set-associative algorithm.

Table 2: Simulation times on FFT (excluding i/o)

b	mul	sin.link	sin.splay	sin.mlist
13	64.8	742.3	741.9	743.7
12	64.0	756.1	759.1	760.9
11	65.8	786.8	783.0	800.7
10	69.7	811.0	816.0	813.4
9	74.6	819.5	826.2	830.1
8	78.1	869.2	858.7	853.0
7	86.2	862.6	868.6	875.8
6	85.3	888.2	894.5	910.5
5	91.3	913.1	922.0	948.3
4	101.3	951.0	956.0	984.0
3	96.1	1006.4	1007.2	1059.0
2	89.6	1032.8	1049.8	1052.9
1	94.0	1086.1	1106.0	1108.1

Block-size = 2^b bytes. Run-time in seconds.

Table 3: Simulation times on Simple (including i/o)

b	mul	sin.link
13	296.5	3231.6
12	302.1	3436.1
11	318.3	3644.3
10	350.3	3897.3
9	409.0	4131.6
8	428.0	4304.5

Run-time in seconds.

Table 4: Simulation times on Weather (including i/o)

b	mul	sin.link
13	350.2	3798.2
12	373.6	4024.1
11	408.7	4298.1
10	483.1	4599.2
9	567.4	4912.1
8	582.4	5167.4
7	584.4	5416.7

Run-time in seconds.

For the single set-associative algorithm, the linked-list implementation with hashing performs best, confirming previous studies on uniprocessor stack simulation implementations [Thompson 87]. For the splay tree version, we did a faithful implementation of the original data structure, not dealing specially with the fact that all insertions occur at the left end. Specializing the implementation to exploit this characteristic might speed up execution, but the potential gain is probably small.

Ideally, one wants to implement the stack with all other balanced tree structures (AVL, red-black) and compare their performances. However, our simulation results indicate that any improvement using sophisticated data structures will be minimal and hardly worth the effort.

For completeness, we also ran the algorithms on randomly generated long synthetic trace data. The same magnitude speed-up in simulation time on the part of arbitrary set-associative algorithm over the single set-associative algorithm still holds. This indicates the stability of the algorithm's performance on different traces.

As most trace files are quite large, they are often stored in compressed format and are uncompressed on-the-fly for a simulation. We piped the results of uncompressing some *.Z files into the various simulation programs and did not see any noticeable change in simulation time. The reason is that the "uncompress" program runs faster than the simulations, the overall speed of the pipeline still depends on the simulation. Therefore using compressed trace files will get almost the same speed-up result for the above simulations.

We did one more comparison. Instead of using one process to run the single set-associative algorithm for each set length, we lumped all of them into one program, which controls multiple independent groups of stacks, one per each set length. On each trace reference, the program sequentially does stack processing on all the stacks. This considerably reduces the I/O time of the single set-associative program. But its memory consumption is, however, much larger than that of the arbitrary set-associative simulation. This excessive space requirement can become a big burden when testing large traces with many distinct addresses. While testing the Weather trace, this kind of simulation failed to finish in a reasonable amount of time.

As a one-pass evaluation method, the arbitrary set-associative algorithm can be run on-the-fly, i.e., simultaneously with a trace generating program[Hill 89], and the saving of a long trace data onto disk can be avoided. This kind of on-the-fly simulation is especially useful, when the amount of *distinct* addresses in the trace can be safely accommodated by the main memory, but the entire trace is extremely long, in which case the required disk space could be overwhelming. As the disk space needed for trace storage is becoming too large even for a short operating period of time of a moderately fast computer nowadays, investigation of on-the-fly techniques is becoming necessary[Baer 91]. Our arbitrary set-associative algorithm is also an applicable tool in this regard.

4.3.4 On concurrent simulation

One might consider concurrently running all the simulation programs of the single set-associative algorithm, using the Unix piping mechanism to pass trace data sequentially from the first simulation program through other simulation programs, relieving them of the necessity of getting trace data through slow disk I/O. The problem with this concurrent-execution approach is again the enormous amount of main memory required. The memory demand of each single set-associative program is approximately equal: each uses the same number of stack entries for valid data blocks; and the difference in the number of stack entries for invalid data blocks (markers) is small, since the number of markers is kept small in any stack by the dropping of markers from the tail of stack. For K set lengths, the main memory demand of the concurrent simulation is approximately K times that of the sequential execution. For small block sizes (hence large set length ranges), that becomes a serious burden on the testing machine's memory system. Excess demand on main memory can cause frequent memory paging and context switching in virtual memory, generating new disk I/O for paging and swapping. Consequently, the real running times of the simulations would be much larger than the sums of their respective user times and system times. Our test on the FFT trace found that the overall running time of this kind of concurrent simulation was comparable to that of the sequential simulation.

As our arbitrary set-associative algorithm uses almost the same amount of memory space as the single set-associative algorithm, while one does concurrent simulations with the single set-associative algorithm on one trace, we can instead run concurrent simulations on different traces with the arbitrary set-associative algorithm–using the same resources of CPU, memory, and time to simulate more traces.

4.3.5 Example of Simulation Results

With the new simulation algorithm, we can get miss ratios for arbitrary cache size (in number of fixed-size blocks) and arbitrary set-associative mapping function in one-pass trace processing. Figure 9, illustrates, regarding a particular CPU for all traces, the relationship between miss ratio and set length α on a cache of 128 blocks, each block with 64 data bytes. For a given cache size, generally (but not always) the fully associative mapping has a lower miss ratio than a set-associative mapping; but occasionally some set-associative mapping has the same or even lower miss ratio than the fully-associative, such as $\alpha = 3$ on trace FFT and $\alpha = 2$ on trace SIMP as shown by Figure 9.

Various performance quantities can be studied using the stack distance distribution data obtained from the efficient one-pass simulation. For example, given

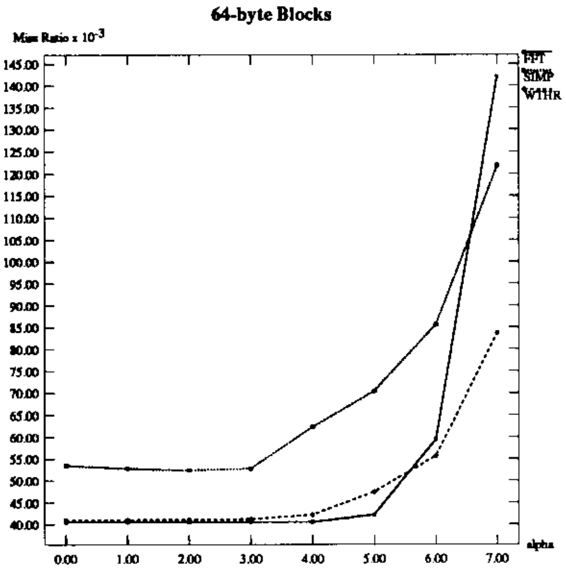


Figure 9: Miss ratio v.s. α for a cache of 128 64-byte blocks.

the cache capacity, one might need to find the optimal set-associative mapping scheme that has the lowest miss ratio. Figure 10 through Figure 19 illustrate, for the same CPU as above, the relationship between the set length α that yields the minimum miss ratio, and the cache size C in number of blocks (the x-axis uses the base-2 $\log C$). Each figure is for a specific block size, ranging from 4-byte block to 8192-byte block. When there is a tie in minimum miss ratio, we break the tie by choosing the α with a larger value, since for a fixed cache size, more sets (i.e., larger α) provide quicker cache searching.

5 Summary

We show that efficient stack analysis can be extended to arbitrary two's power congruence set-associative mapping for LRU caches on multiprocessors. For block addresses between 0 and $2^k - 1$, instead of running stack evaluation on the same trace $k + 1$ times for all the possible set lengths, one run of stack evaluation on the trace can give us the same hit ratio function for all set lengths. Thanks to the locality property of CPU access in real applications, the necessity of using a simple linear list stack structure for the arbitrary set-associative evaluation does not compromise its simulation time. Simulation on real multiprocessor trace data show an order-of-magnitude speed up by our algorithm in simulation time.

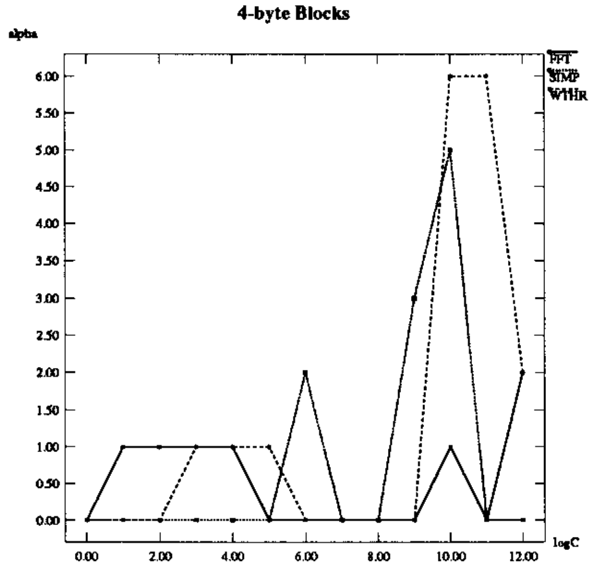


Figure 10: Optimal α v.s. cache size in 4-byte blocks.

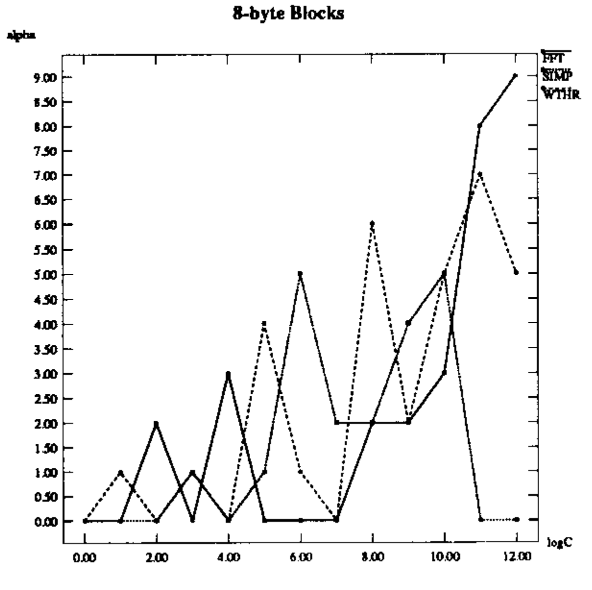


Figure 11: Optimal α v.s. cache size in 8-byte blocks.

Acknowledgements

We would like to thank Prof. Anant Agarwal and Mr. David Chaiken at MIT for making the multiprocessor trace data available; David Chaiken was very helpful in answering our questions about the trace files. The first author would also like to thank Prof. Daniel Sleator at Carnegie Mellon for helping him understand the proof of the Working Set Theorem on splay trees and providing his implementation of top-down splaying.

References

[Mattson 70] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Hierarchical Storage Evaluation Techniques", *IBM Systems Journal*, Vol. 17, No. 2, pp. 78-117, February 1970.

[Gecsei 74] J. Gecsei, "Determining Hit Ratio for Multilevel Hierarchies", *IBM Journal of Research and Development*, Vol. 18, No. 4, pp. 316-327, July 1974.

[Guibas 78] L. J. Guibas, R. Sedgewick, "A Dichromatic Framework for Balanced Trees", *Proceedings of the 19th Annual Symposium on foundations of Computer Science*, pp. 8-21, October 1978.

[Censier 78] L. M. Censier, P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems", *IEEE Transactions on Computers*, Vol. C-27, No. 12, pp. 1112-1118, December 1978.

[Smith 82] A. J. Smith, "Cache Memories", *ACM Computing Surveys*, Vol. 14, No.3, pp. 473-530, September 1982.

[Dubois 82] M. Dubois, F. Briggs, "Effects of Cache Coherency in Multiprocessors", *IEEE Transactions on Computers*, Vol. C-31, No. 11, pp. 1083-1099, November 1982.

[Papamarcos 84] M. S. Papamarcos, J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 348-354, June 1984.

[McCreight 84] E. McCreight, "The Dragon Computer System: An Early Overview", Technical Report, Xerox Corporation, September 1984.

[Sleator 85] D. D. Sleator, R. E. Tarjan, "Self-Adjusting Binary Search Trees", *JACM*, Vol. 32, No. 3, pp. 652-686, July 1985.

[Archibald 86] J. Archibald, J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, Vol. 4, No. 4, pp. 273-298, November 1986.

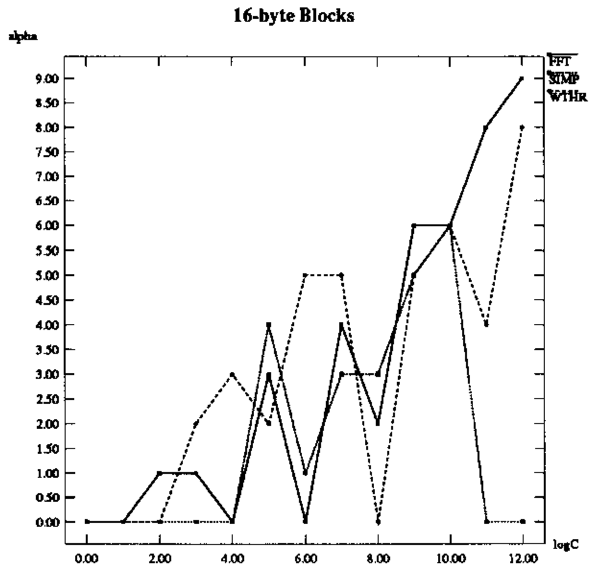


Figure 12: Optimal α v.s. cache size in 16-byte blocks.

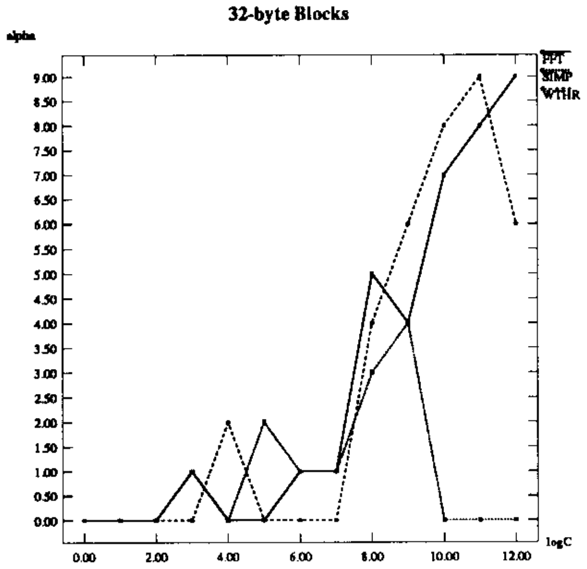


Figure 13: Optimal α v.s. cache size in 32-byte blocks.

[Thompson 87] J. G. Thompson, "Efficient Analysis of Caching Systems", T.R. UCB/CSD 87/374, Ph.D. dissertation, Univ. of California, Berkeley, Oct. 1987.

[Wyk 88] C. J. Van Wyk, *Data Structures and C Programs*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[Dubois 88] M. Dubois, C. Scheurich, F. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors", *IEEE Computer*, pp. 9-21, February 1988.

[Cheong 88] H. Cheong, A. V. Veidenbaum, "A Cache Coherency Scheme with Fast Selective Invalidation", *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 299-307, May 1988.

[Thompson 89] J. G. Thompson, A. J. Smith, "Efficient (Stack) Algorithms for Analysis of Write-Back and Sector Memories", *ACM Transactions on Computer Systems*, Vol. 7, No. 1, pp. 78-116, February 1989.

[Weber 89] W.-D. Weber, A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors", *Proceedings of the ASPLOS III*, pp. 243-256, April 1989.

[Cherian 89] M. M. Cherian, "A Study of Backoff Barrier Synchronization", MIT/LCS/TR-452, June 1989.

[Wang 89] W. Wang, "Multilevel Cache Hierarchies", DCS TR 89-09-13, Ph.D Dissertation, Univ. of Washington, Seattle, September 1989.

[Li 89] K. Li, P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321-359, November 1989.

[Hill 89] M. D. Hill, A. J. Smith, "Evaluating Associativity in CPU Caches", *IEEE Transactions on Computers*, Vol. C-38, No. 12, pp. 1612-1630, December 1989.

[Stenström 90] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, pp. 12-25, June 1990.

[Chaiken 90] D. Chaiken, C. Fields, K. Kurihara, A. Agarwal, "Directory-Based Cache Coherence in Large-Scale Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, pp. 49-59, June 1990.

[Koldinger 91] E. J. Koldinger, S. J. Eggers, H. M. Levy, "On the Validity of Trace-Driven Simulation for Multiprocessors", *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 244-253, May 1991.

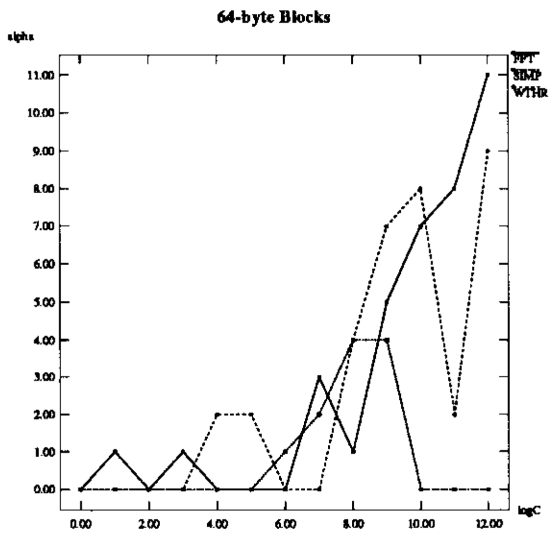


Figure 14: Optimal α v.s. cache size in 64-byte blocks.

[Wang 91] W. Wang, J.-L. Baer, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis", *ACM Transactions on Computer Systems*, Vol. 9, No. 3, pp. 222-241, September 1991.

[Baer 91] J.-L. Baer, *private communication*, December 1991.

[Wu 92] Y. Wu, J. Popek, R. R. Muntz, "Efficient Evaluation of Arbitrary Set-Associative Caches on Multiprocessors", *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pp. 507-514, December 1992.

[Smith 93] A. J. Smith, *private communication*, April 1993.

[Chame 93] J. Chame, M. Dubois, "Cache Inclusion and Processor Sampling in Multiprocessor Simulations", *Proceedings of ACM Sigmetrics'93*, pp. 36-47, May 1993.

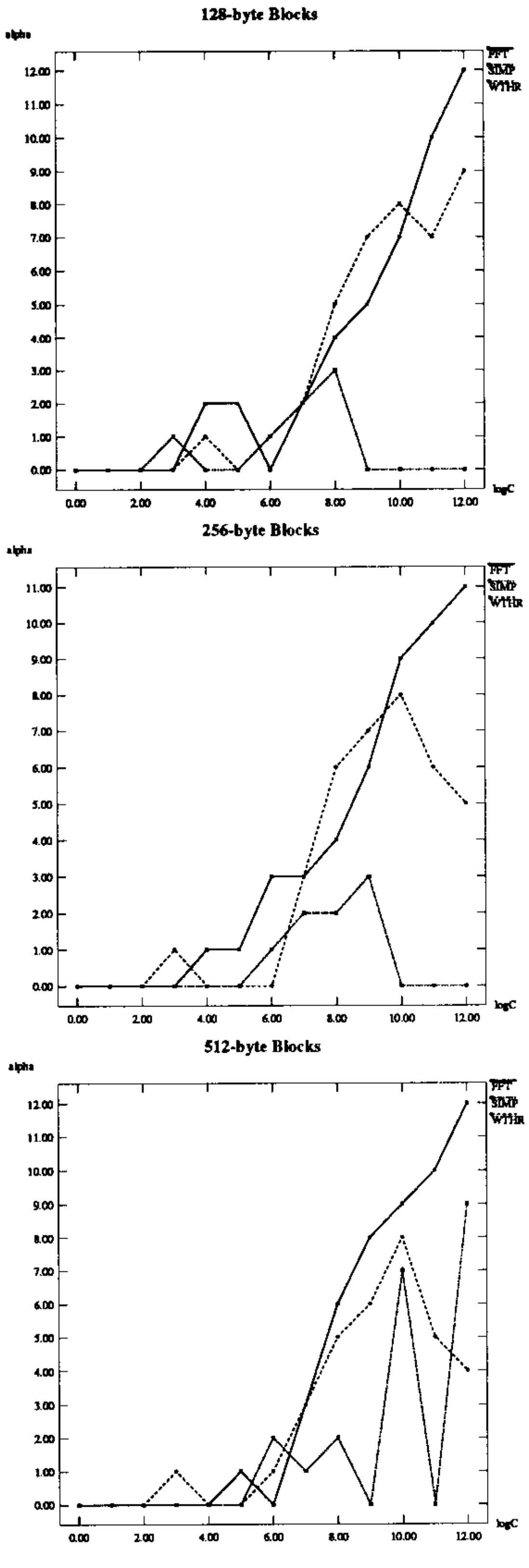


Figure 15: Optimal α v.s. $\log C$ in 128,256,512-bytes.

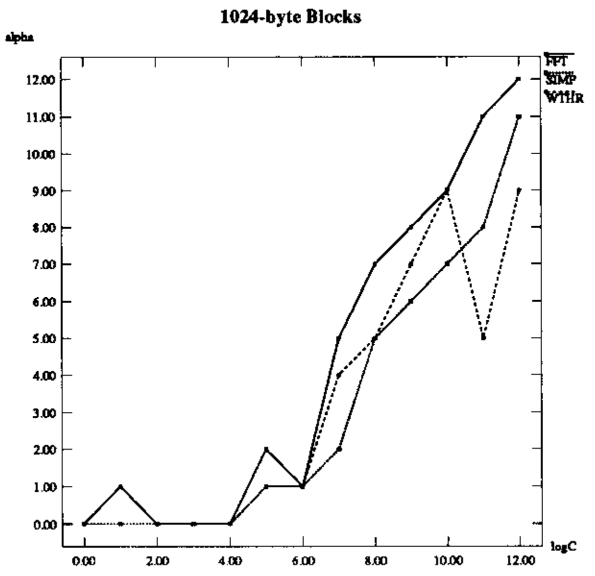


Figure 16: Optimal α v.s. cache size in 1024-byte blocks.

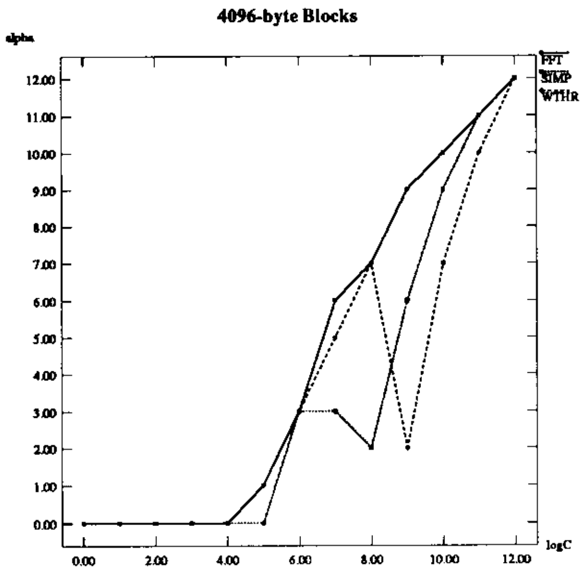


Figure 18: Optimal α v.s. cache size in 4096-byte blocks.

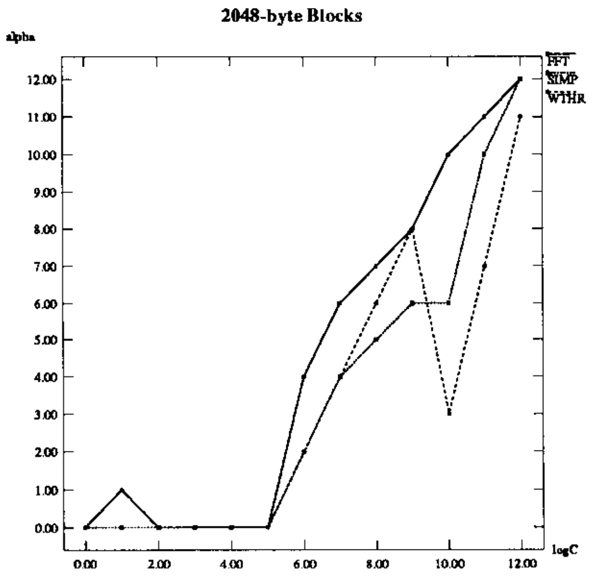


Figure 17: Optimal α v.s. cache size in 2048-byte blocks.

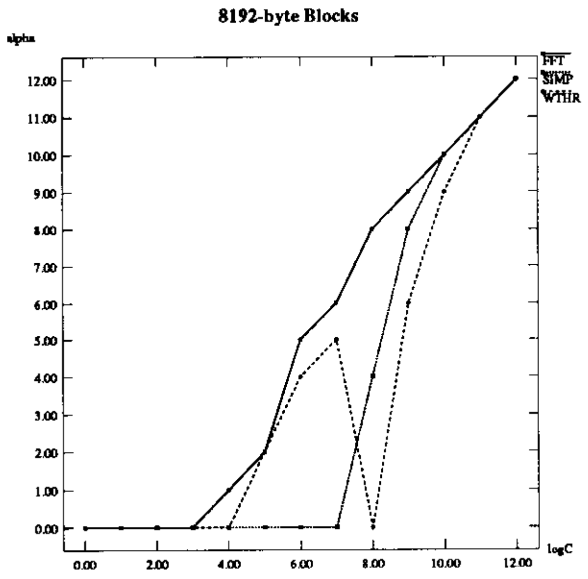


Figure 19: Optimal α v.s. cache size in 8192-byte blocks.