Set-Associative Cache Simulation Using Generalized Binomial Trees

RABIN A. SUGUMAR
Cray Research Inc.
and
SANTOSH G. ABRAHAM
Hewlett-Packard Laboratories

Set-associative caches are widely used in CPU memory hierarchies, I/O subsystems, and file systems to reduce average access times. This article proposes an efficient simulation technique for simulating a group of set-associative caches in a single pass through the address trace, where all caches have the same line size but varying associativities and varying number of sets. The article also introduces a generalization of the ordinary binomial tree and presents a representation of caches in this class using the Generalized Binomial Tree (gbt). The tree representation permits efficient search and update of the caches. Theoretically, the new algorithm, GBF_LS, based on the gbt structure, always takes fewer comparisons than the two earlier algorithms for the same class of caches: all-associativity and generalized forest simulation. Experimentally, the new algorithm shows performance gains in the range of 1.2 to 3.8 over the earlier algorithms on address traces of the SPEC benchmarks. A related algorithm for simulating multiple alternative direct-mapped caches with fixed cache size, but varying line size, is also presented.

Categories and Subject Descriptors: B.3.3 [**Memory Structures**]: Performance Analysis and Design—*simulation*; E.1 [**Data**]: Data Structures—*trees*; I.6.8 [**Simulation and Modeling**]: Types of Simulation

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: All-associativity simulation, binomial tree, cache modeling, inclusion properties, set-associative caches, single-pass simulation, trace-driven simulation

1. INTRODUCTION

Caches are widely used in computer systems to decrease average access times. There are excellent references on caches in the literature, for example, Smith [1982] and Stone [1987] for CPU caches and Smith [1985] for disk caches. Set-associative caches are commonly used in caching systems. An

This research was supported in part by ONR contract no. N00014-93-1-0163.
Authors' addresses: R. A. Sugumar, Cray Research Inc., 900 Lowater Road, Chippewa Falls, WI 54729; email: rabin@cray.com; S G. Abraham, Hewlett-Packard Laboratories, 1501 Page Mill Road, MS3L-5, Palo Alto, CA 94304; email: abraham@hlp.hp.com.

important part of set-associative cache design is the determination of the three parameters: cache size, line size, and degree of associativity. The optimality of the cache parameters depends on the workload; so cache design is typically done by simulating various configurations on address traces of representative workloads. The other common evaluation options—analytical modeling and hardware prototyping—are not usually used for evaluating cache designs; analytical modeling lacks accuracy, and hardware prototyping is expensive.

Cache simulation is time intensive primarily because of (1) the trace length and (2) the number of alternate cache designs that have to be evaluated. Traces have to be long to make sure that they are representative of the program and that cold-start misses are a small fraction of the total misses. The number of designs to be evaluated is high owing to the many different parameters, such as cache size and associativity, that may be varied. Three complementary approaches have been proposed to speed up cache simulation: reduced-trace simulation, parallel simulation, and single-pass simulation. In the reduced-trace approach, a shorter version of the original trace is used to do the simulation in a fraction of the time it would take to simulate the entire trace. In parallel cache simulation, several processors are used to run simulations in parallel, reducing overall simulation time. In single-pass simulation, several cache configurations are simulated in a single pass through the address trace, and relations between cache configurations are exploited to reduce simulation time greatly. This article is on the single-pass simulation approach. Single-pass simulation may be used in combination with the reduced-trace and parallel-simulation approaches.

In this article, a cache simulation algorithm is presented for single-pass simulation of a group of set-associative caches with fixed line size, least recently used (LRU) replacement, bit selection,[1] but varying associativities and varying number of sets. A two-level memory hierarchy model is assumed with a single cache and main memory; the simulation algorithms may be extended to simulate secondary caches or split caches. The input to the simulation is a trace of addresses generated by the CPU, and the output is the miss ratio of each simulated cache.

Two algorithms have been proposed earlier to simulate this class of caches: all-associativity simulation [Hill and Smith 1989; Mattson et al. 1970] (called AA in the following) and a generalization of forest simulation [Hill and Smith 1989] (called FS + in the following). Our new algorithm, GBF_LS, based on a binomial forest representation of cache configurations, performs better than either of the two earlier algorithms because it eliminates some of the unnecessary comparisons done by AA and FS + . Consider the simulation of caches with one and two sets, and of maximum associativity n. The data structures maintained by FS + , AA, and GBF_LS are shown in Figure 1 ($n = 4$ in the figure). FS + maintains three lists of length n, one representing the caches with one set, and two more representing each of the two sets in the caches

[1]A bit field of the address determines the set.

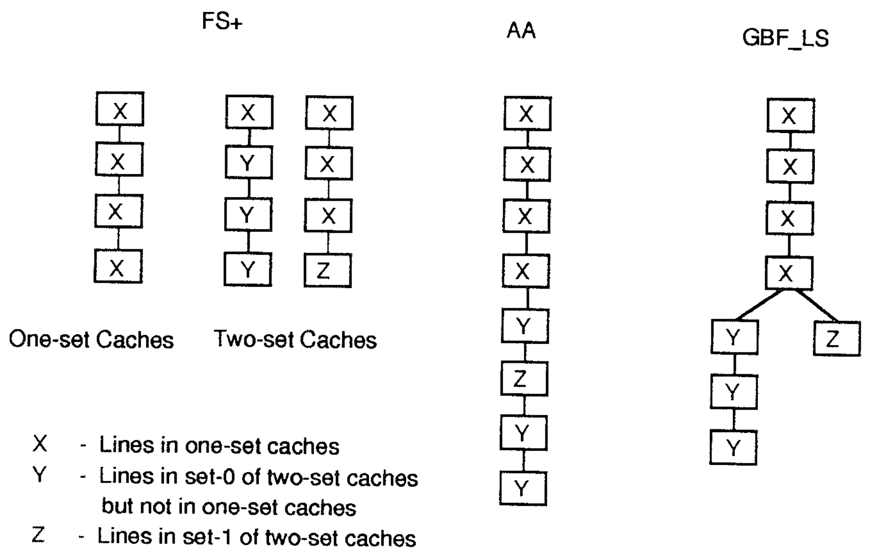

FS+
AA
GBF_LS
X
Y
Z
One-set Caches
Two-set Caches
X - Lines in one-set caches
Y - Lines in set-0 of two-set caches but not in one-set caches
Z - Lines in set-1 of two-set caches

Fig. 1 Data structures used in the three simulation algorithms.

with two sets. For each trace reference, FS + searches the list of the one-set caches first. If the referenced line is not found at the top of the list, one of the lists of the two-set caches is searched. AA maintains one list of length at least $2n$, and searches the list until the referenced line is found or until the end of the list is reached. In FS + , the n entries in the list for one-set caches are repeated in the lists for two-set caches. Therefore, the drawback with FS + is that the same line may be examined twice, once in the list for the one-set caches, and then again in one of the lists for the two-set caches. On the other hand, AA keeps just one entry for each line in its single list. The drawback with AA is that lines that belong to the "wrong set" may be examined, e.g., lines that are only in set-1 of the two-set caches while searching for a line that maps to set-0.

GBF_LS, described in this article, maintains the first list of n entries representing the one-set caches similar to AA. After that, however, GBF_LS maintains two separate lists, one of some length n_1 ($< n$) consisting of lines in set-0 of the two-set caches but not in the one-set caches ($n_1 = 3$ in the figure), and the other of length ($n - n_1$) consisting of the lines in set-1 but not in the one-set caches. On a reference the first list is searched; if the referenced line is not found, the set that the line maps to is determined, and the corresponding list is searched. Unlike FS + , this scheme does not examine any line twice, and unlike AA it does not examine lines from the wrong set. We call the tree-of-lists data structure used by GBF_LS a *Generalized Binomial Tree* (*gbt*); it can be extended in a recursive fashion to represent caches with four or more sets. A gbt is a combination of binomial trees and lists; the binomial tree structure captures the subsetting relationship among caches with varying number of sets, and the list structure captures the relationship among caches of varying associativities.

This article also presents a cache simulation algorithm, BF_CS, for simulating a group of direct-mapped caches of constant size, but varying line sizes using binomial trees. Traiger and Slutz [1971] present an adaptation of AA for simulating multiple line sizes in addition to multiple associativities and number of sets. Their algorithm can simulate a wider range of configurations than can BF_CS. However, for simulating direct-mapped caches of a range of line sizes and fixed cache size (useful in primary-cache design, for instance) BF_CS is faster.

The rest of the article is organized as follows. Section 2 reviews related work and contrasts our work with that of others. Section 3 develops the gbt representation and the new algorithm. We also discus the complexity of the algorithms. Section 4 discusses the direct-mapped cache simulation algorithm. Section 5 reports the results of empirical comparisons. Section 6 concludes the article.

2. RELATED WORK

The pioneering work on single-pass simulation of memory hierarchies was done by Mattson et al. [1970] at IBM in the context of virtual-memory systems. In that work they introduced single-pass algorithms for simulating different types of fully associative caches, and the AA algorithm for simulating set-associative caches assuming bit selection. Later, Hill and Smith [1989] generalized AA to work for set-mapping functions other than bit selection.[2] Hill and Smith also describe the forest simulation algorithms, FS and FS + , for simulating direct-mapped and set-associative caches. As mentioned earlier the algorithm introduced in this article simulates the same class of caches as AA and FS + , but more efficiently.

Thompson [1987] describes an extension to Mattson et al's fully associative cache simulation algorithm that permits counting the number of dirty misses with a write-back cache. Wang and Baer [1990] report an extension to AA to count dirty misses along the same lines. Wang and Baer's extension may be incorporated into the gbt algorithm we propose. Traiger and Slutz present an extension to AA for simulating caches of multiple line sizes as well. Their extension may also be incorporated into the gbt algorithm we propose, provided all the caches are represented in one tree. We also present in Section 4 a novel algorithm for simulating direct-mapped caches of a range of line sizes and fixed cache size. Trees structures have been used to speed up fully associative searches by Olken [1981], Bennett and Kruskal [1975], and Sugumar and Abraham [1993].

Cache simulation can also be speeded up by using reduced traces that are obtained by filtering out a large fraction of the references in the complete trace. The early work on reduced traces was done by Smith [1977] who proposed deleting references that hit at top levels of the LRU stack. Puzak [1985] proposed filtering out references that hit in a small direct-mapped cache; the resulting reduced trace can be used to simulate direct-mapped and

[2]The name all-associativity is due to Hill and Smith [1989].

set-associative caches with more sets but the same line size. Wang and Baer [1990] generalized this approach to work with varying line sizes too. Laha et al. [1988] proposed a time-sampling technique, where only a few segments of the complete trace are selected for simulation. Puzak also proposed set-sampling where references mapping to a few sets are selected for simulation. The single-pass simulation methods described here can be used for simulating reduced traces as well.

Parallelization is another approach to speeding up cache simulation. A simple technique to exploit parallelism is to simulate the different configurations on separate machines. Work has also been done on parallelizing single-pass simulation methods. Heidelberger and Stone [1990] parallelize cache simulation by breaking the trace up into segments, simulating the segments in parallel, and merging the results of the segment simulations. Sugumar [1993] proposes partitioning the fully associative stack among the processors and performing the stack search in parallel.

3. SET-ASSOCIATIVE CACHE SIMULATION

The following notation is used: a set-associative cache with 2^S sets, line size 2^L, and associativity n is denoted as $C_S^L(n)$ (S is the width of the set field, and L is the width of the line field). $[X]_{lines}$ denotes the lines contained in X, where X may be a set, a group of sets, or a cache. The input to the simulation is a trace $x_1, x_2, \ldots, x_{TL}$ of addresses. For any address, set number, or line number y, $y[i{:}j]$ denotes the bit field between bits i and j (inclusive) in the binary representation of y. The least significant bit is numbered 0. The number of bits in an address is denoted by W.

3.1 Cache Relations and Data Structure

In this subsection we prove some basic properties used by all three simulation algorithms (FS + , AA, and GBF_LS) and review AA and FS + .

LEMMA 3.1.1. *For each set p in $C_S^L(n)$ there are exactly 2^k ($k \geq 0$) sets, P, in $C_{S+k}^L(n)$ such that $[p]_{lines} \subseteq [P]_{lines}$ and $([P]_{lines} - [p]_{lines}) \cap [C_S^L(n)]_{lines} = \phi$.*

PROOF. k extra bits are used for selecting a set in $C_{S+k}^L(n)$ than in $C_S^L(n)$. So lines mapping to a single set, p, in $C_S^L(n)$ map to one of 2^k sets in $C_{S+k}^L(n)$, given by $P = \{s \text{ s.t. } s[S-1{:}\,0] = p\}$. Only n of the $n2^k$ lines of P are present in $C_S^L(n)$, and those are the n lines that arrived most recently. That is, $[p]_{lines}$ consists of the n lines that arrived most recently in P.

Conversely, lines mapping to one of the sets in P in $C_{S+k}^L(n)$ can only map to p in $C_S^L(n)$. So $C_S^L(n)$ does not have any of the contents of P apart from $[p]_{lines}$. □

The following two corollaries follow from the lemma and the proof.

COROLLARY 3.1.2. *The least significant S bits of the set numbers of the sets in P are identical, and give the set number of p.*

COROLLARY 3.1.3. $[C_{S_1}^L(n_1)]_{lines} \subseteq [C_{S_2}^L(n_2)]_{lines}$, *if $S_1 \leq S_2$ and $n_1 \leq n_2$.*

Corollary 3.1.3 states an inclusion property between caches in this class and is the sufficient part of Theorem 1 in Hill and Smith [1989] except that bit selection is assumed here. This inclusion property is used in FS + and AA. Let the caches to be simulated be $C^L_{S+i}(j)$, $i = 0, \dots M$, $j = 1, \dots, n$. In FS + a separate two-dimensional array is maintained for each of the caches $C^L_{S+i}(n)$, $i = 0, \dots, M$, with sets along one dimension and the n lines mapping to the set in LRU order along the other. From the inclusion property the contents of $C^L_{S+i}(j)$ for $j < n$ are the first j lines in each set in $C^L_{S+i}(n)$. For each incoming line, the appropriate set in each array is searched, starting at the array with the minimum number of sets. If the reference hits at depth m in cache $C^L_{S+k}(n)$, the reference hits in caches $C^L_{S+k}(j)$, $j = m, \dots, n$ by the inclusion property. Hit information for these caches is updated, and the referenced line is moved to the top of its set. Also when $m = 1$ the line is known to hit in all the remaining caches by the inclusion property, and the simulation moves on to the next trace reference.

AA uses this inclusion property, and the additional property that lines that map to the same set in $C^L_{S_2}(n)$ map to the same set in $C^L_{S_1}(n)$ (called *set refinement* [Hill and Smith 1989]). Consider the simulation of two caches, $C^L_{S_1}(n)$ and $C^L_{S_2}(n)$, $S_1 < S_2$. For each set in the smaller cache $C^L_{S_1}(n)$, the complete LRU stack, consisting of all the lines that map to this set in the course of the simulation, is maintained. For each incoming line, the LRU stack of the set it maps to is searched starting at the most recently referenced line. The line hits in $C^L_{S_1}(n)$ if it is found at a depth less than or equal to n in the stack. Furthermore, by the set refinement property, the lines examined while searching for a specific line, a, in the LRU stack of $C^L_{S_1}(n)$ are a superset of the lines that would be examined while searching for a in the LRU stack of $C^L_{S_2}(n)$. By maintaining a count of the lines that would map to the same set as a in $C^L_{S_2}(n)$ while searching the stack of $C^L_{S_1}(n)$,[3] the depth at which the reference hits in $C^L_{S_2}(n)$ may also be determined. The referenced line is moved to the top of its stack.

Figure 2 shows an example illustrating FS + and AA. Here the number of sets ranges from one to four, and the associativities are one and two. The states of the data structures after line 1011 is processed are shown. If the next reference is to line 0101, it is found at the top only in $C^L_2(2)$, and FS + requires five comparisons; AA requires just two comparisons. FS + requires more comparisons in this case, because it reexamines lines that it has seen in earlier caches. However, if the next reference is to line 0011, AA requires eight comparisons, whereas FS + requires only six comparisons. FS + is better in this case because the LRU stack for $C^L_0(2)$ contains lines that are not in the set 0011 maps to in any of the caches, $C^L_0(2)$, $C^L_1(2)$, or $C^L_2(2)$. In the

[3]A line in $C^L_{S_1}(n)$ is known to map to the same set as a in $C^L_{S_2}(n)$, if the right match between line a and the line in $C^L_{S_1}(n)$ is greater than $S_2 - S_1$. *Right match* between two binary numbers a_1 and a_2 is defined as the minimum i such that $a_i[i{:}0] \neq a_2[i{:}0]$.

Trace: 0011, 1010, 0100, 0001, 0110, 1000, 0101, 1011

Conventional representation (for FS+)

Cache	Tag	Set No.
$C_0^L(2)$	1011, 0101	-
$C_1^L(2)$	100, 011	0
	101, 010	1
$C_2^L(2)$	10, 01	00
	01, 00	01
	01, 10	10
	10, 00	11

Stack representation (for AA)

1011
0101
1000
0110
0001
0100
1010
0011

Incoming Address	Method	No. of Comparisons
0101	FS+	5
	AA	2
0011	FS+	6
	AA	8

Fig. 2. Examples illustrating FS + and AA.

rest of this section we develop the new algorithm which always does as well as or better than both FS + and AA.

3.2 Generalized Binomial Forest Representation

Here we define the generalized binomial tree data structure. (Other details and definitions relating to this data structure are included in the Appendix.) We then describe the gbt representation of the class of caches.

The gbt is a new data structure that we introduce and is obtained by composing binomial trees with lists. While a binomial tree is described by its degree, a gbt is described by its degree and order, where the degree is the degree of the underlying binomial tree, and the order is the size of the underlying lists.

Definition 3.2.1. The following is a definition by construction of a gbt of order n (gbt(n)). (Figure 3). A gbt(n) of degree zero, $B_0(n)$, is a list of length n. A gbt(n) of degree x, $B_x(n)$, is constructed by putting together two gbt(n)s of

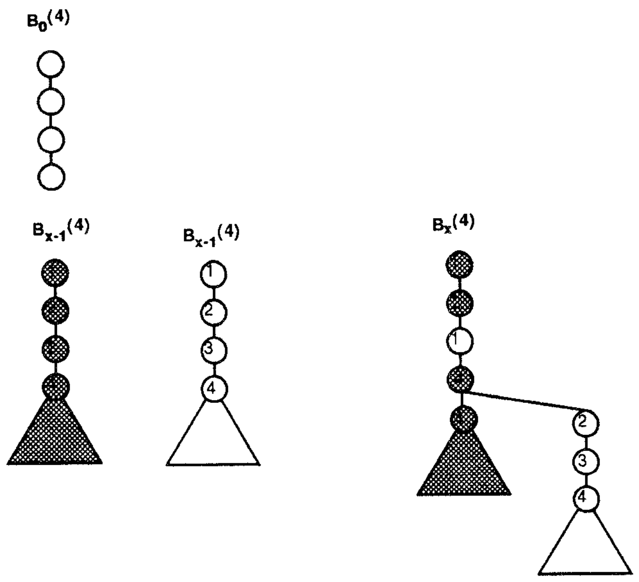


Fig. 3. Definition of the generalized binomial tree.

degree $x - 1$, $B_{x-1}(n)$, and $B'_{x-1}(n)$, as follows:

(1) Two segments of lengths n_1 and n_2 beginning at the roots of $B_{x-1}(n)$ and $B'_{x-1}(n)$ are removed so that $n_1 + n_2 = n$.

(2) These two segments are merged in an order, determined by the application, to form the root list of $B_x(n)$.

(3) The remaining parts of $B_{x-1}(n)$ and $B'_{x-1}(n)$ are attached to the end of this root list.

Lemma 3.1.1 leads to the gbt representation of caches used in the new algorithm. Consider the sets in cache $C^L_{S+2}(n)$, where each set contains a list of n lines. Group sets in $C^L_{S+2}(n)$ mapping to the same set in $C^L_{S+1}(n)$ into pairs. Combine the line lists in the two sets in each pair by forming a list of the n most recently referenced lines in either set. Use this list as the root list, leaving the remaining n lines in two separate branches, each branch containing lines from distinct sets in $C^L_{S+2}(n)$. Clearly, we now have a forest of gbt(n)s of degree one. The lines in the root list are in $C^L_{S+1}(n)$, while the other lines are only in $C^L_{S+2}(n)$. The structures formed can be further grouped into pairs and combined resulting in a gbt(n) of degree two with the line in the root list being in $C^L_S(n)$. Further combining may similarly be done obtaining gbt(n)s of higher degrees. Figure 4 illustrates the construction of the gbt representation for the example of Figure 2. Here $S = 0$: the tag field for $C^L_{S+2}(n)$ is shown within parentheses; the line field is not shown.

To simulate caches $C^L_{S+i}(j)$, $i = 0, \ldots, M$, $j = 1, \ldots, n$, the sets in $C^L_{S+M}(n)$ are combined until there are 2^S distinct gbt(n)s of degree M. The generalized binomial forest (gbf) representation of the caches obtained after combining

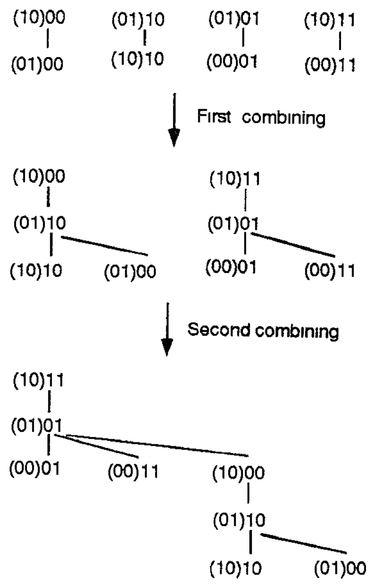


Fig. 4. An example construction of a gbt representation.

has the following properties. In the following, set number and tag are with respect to $C^L_{S+M}(n)$ unless stated otherwise. (For the definitions of terms such as degree and subtree of any gbt please refer to the Appendix).

Property 3.2.2. In a tree or subtree of degree k, the least significant $S + M - k$ bits of the set numbers are identical, and two nonoverlapping subtrees of degree k differ in at least one of the least significant $S + M - k$ bits of the set numbers.

Property 3.2.3. A line of rank k is in caches $C^L_{S+i}(n)$, $i = M - k, \dots, M$.

Property 3.2.4. The path from the root to any line a contains exactly those lines that are in sets that a maps to in one or more of the caches considered, and are more recently referenced than a.

The validity of Properties 3.2.2 and 3.2.3 may be seen by considering the state of the structure just before trees of degree k are combined. The lines in the root list at this stage are of rank k (Lemma A.5) and are present in cache $C^L_{S+M-k}(n)$. By the inclusion property the lines in the root list are in larger caches of associativity n too, i.e., caches $C^L_{S+i}(n)$, $i = M - k + 1, \dots, M$. This is Property 3.2.3. Since at this point all the lines in a tree map to the same set in $C^L_{S+M-k}(n)$ the least significant $S + M - k$ bits of the lines in a tree have to be the same, and among trees they have to be different. This is Property 3.2.2.

For Property 3.2.4, consider the lines in the root list of the degree-k tree which contains the line a. These lines are in $C^L_{S+M-k}(n)$ in the set a maps to, by Properties 3.2.2 and 3.2.3. All the other lines in the gbt above a are similarly part of the root list at some stage in the combining procedure. So they are in at least one of the caches $C^L_{S+M-i}(n)$, $i = k + 1, \dots, M$, in the set

a maps to. Conversely, lines that are not above a in the gbt are not in sets that a maps to. This is Property 3.2.4.

Two operations are defined on this data structure: SWAP and EXCHANGE. SWAP(v) is permitted only when v is a tree child (Defn. 5 in appendix) and causes the tree of degree k rooted at v to be swapped with the subtree of degree k rooted at its parent. By the definition of subtrees (Definition A.9), the parent of a tree child of degree k has a subtree of degree k, and so SWAP is always possible for a tree child. EXCHANGE(v) is permitted only when v is a list child (Definition A.6) and causes v to be exchanged with its parent node. The children of v become the children of its parent and vice-versa (it is possible that a tree child of v becomes a list child for its parent or vice-versa after the EXCHANGE). SWAP and EXCHANGE are used to move a line up the tree when its priority changes. They basically alter the various combinings to reflect new priority information.

3.3 Algorithm

Algorithm GBF_LS (Generalized Binomial Forest—fixed Line Size) is the algorithm based on the gbt representation. For each address x, the tree that will contain the corresponding line is first identified using the set field in the smallest cache $x[S + L - 1: L]$. The tree is then searched for the line as follows:

(1) On a right match of k at a node, the tree child of rank $S + M - k - 1$ is searched next if the tree child is present. If the node does not have a tree child of rank $S + M - k - 1$ the list child is of rank greater than or equal to $S + M - k - 1$ (Lemma A.8) and is searched next.

(2) When there is a complete set match and tag match at a node, the line is found, and the search is successful.

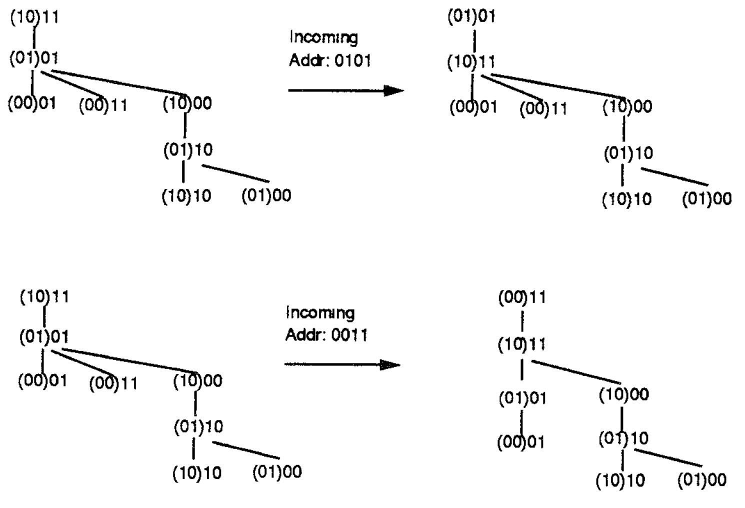
(3) When there is a complete set match at a node, and the node does not have a list child, the search has failed. The line at the node examined last is replaced with the incoming line, which is now the most recently referenced line, and is moved to the root by a series of SWAPs and EXCHANGEs.[4]

Property 3.2.3 lets us infer which caches of associativity n are hit. Furthermore, using Property 3.2.4 and right matches done along the search path we can determine which caches of associativities smaller than n are hit as follows. The number of right matches of $S + i$ or greater is calculated for $i = 0, \dots, M$ from the right-match counts obtained during the search. When the number of right matches of $S + i$ or greater is t, the reference hits in

$$C^L_{S+i}(j), j = t + 1, \dots, n.$$

Initialization involves building the forest by combining sets as described earlier. All the lines are assumed invalid at the start, and so they may be ordered arbitrarily. If a warm start is desired, the simulation should be allowed to run for some time before collecting statistics.

[4] In the algorithm, the SWAPs are done during the search which is equivalent.



Incoming Address	No of Comparisons
0101	2
0011	3

Fig. 5. Examples illustrating Algorithm GBF_LS.

In Figure 5, the examples of Figure 2 are shown processed by Algorithm GBF_LS.

(1) For the incoming line 0101, the first line checked is 1011. The set field of the referenced line does not match the set field of 1011; the right match of one is noted, and the search moves to the only child 0101. Here the set field and the tag field match. One right match of one is seen along the path, and so the reference hits in $C_0^L(2)$, $C_1^L(2)$, $C_2^L(1)$, and $C_2^L(2)$. 0101 is now moved to the top of the tree, which in this case is accomplished through an EXCHANGE.

(2) For the incoming line 0011, the tag does not match at the root; the right match of two is noted, and the search goes to 0101. Here the set field does not match. The right match of the set fields of 0011 and 0101 is one; so $(S + M - k - 1) = (0 + 2 - 1 - 1) = 0$. Since 0011 is a tree child of rank zero, the subtree rooted at 0011, {0011}, is swapped with the subtree {0101, 0001} and searched. Both the set and tag fields match at 0011. The search is successful. Since right matches of one and two were seen along the search path, the reference hits only in $C_2^L(2)$. 0011 is moved to the root through an EXCHANGE.

The search takes two comparisons for 0101 and three comparisons for 0011. These are fewer than or the same as the number of comparisons for either of the algorithms shown in Figure 2. We show later that the number of comparisons required in GBF_LS is always less than or equal to the number of comparisons required in AA and FS + .

```
Algorithm GBF_LS
SIM_GBF_LS()
  Initialize()
  for every reference $x$ in trace
    set_no_C0 ← Set number in $C_0$ ($x[S+L-1:L]$)
    set_no_CM ← Set number in $C_M$ ($x[S+M+L-1:L]$)
    tag ← Tag in $C_M$ ($x[W-1:S+M+L]$)
    cur_node ← Root_Map[set_no_C0]
    found ← 0; end_of_tree ← 0
    for $i$ ← 0 to $M$
      RM_Count[$i$] ← 0
    end for
    while (NOT found AND NOT end_of_tree)
      if ((cur_node → set_no = set_no_CM) AND (cur_node → tag = tag))
        /* Search successful */
        found ← 1
        for $i$ ← $M$ to 0
          sum ← sum + RM_Count[$i$]
          Hit_Array[$i$][sum] ← Hit_Array[i][sum] + 1
        end for
      else
        $k$ ← RIGHT_MATCH(cur_node → set_no, set_no_CM)
        next_node ← Child(cur_node, $S+M-k-1$)
        if (next_node == NULL)
          /* Search failed */
          cur_node → tag ← tag
          end_of_tree ← 1
        else
          /* Increment Right Match count and continue search */
          RM_Count[$k$] ← RM_Count[$k$] + 1
          if (next_node is a tree-child of cur_node)
            SWAP(next_node)
          cur_node ← next_node

  end while
      Move_to_Top(cur_node)
end for
for $m$ ← 1 to $n$:
  Hits in cache $C^L_{S+k}(m) = \sum_{i=0}^{m} Hit\_Array[k][i]$
end for

Move_to_Top(node)
  while (node not at root of tree)
    EXCHANGE(node)
  end while
  Root_Map[set_no_C0] ← node
```

```
Child(node, $d$)
  if ($d < 0$)
    /* Complete match at node */
    if (node has no list-child)
      return(NULL)
    else
      return(list-child)
  else if (node has tree-child of rank $d$)
    return(tree-child of rank $d$)
  else
    return(list-child)
Initialize()
  Build $2^S$ gbt's from $2^{S+M}$ line lists
    (Using the combining procedure, assuming arbitrary ordering)
  Set all tags to invalid
```

3.4 Implementation Issues

In this section we discuss issues relating to the implementation of generalized binomial trees[5] and a microoptimization which saves some simulation time.

An array implementation for the gbt is most efficient. A two-dimensional array with $2^{M+1} - 1$ rows and n columns is used to implement a gbt(n) of degree M (there are roughly twice the number of locations in the array as there are nodes in the gbt). The rows of the array are grouped into $M + 1$ levels, where level k has 2^k rows, numbered 0 through $2^k - 1$. Lines of rank k in the gbt map to row $s[S + M - 1{:}S + k]$ in level $M - k$ of the array, and are in sequence in the row. The rank of the list child, d_l, of the line is maintained for lines at the end of rows. ($d_l = -1$ when the line does not have a list child.) If it exists, the list child is the first child in row $x[S + M + L - 1{:}\ S + L + d_l]$ in level $M - d_l$. The rank-d tree child of a line may be located as follows: if $d_l \leq d$, the tree child exists and is located at level $M - d$ in row $x[S + M + L - 1{:}\ S + L + d]$ (x is the incoming address); otherwise ($d_l > d$), the rank-d tree child does not exist. The children of a line may also be located by just searching the appropriate row in each level greater than the current level, until a nonempty row is located. In this scheme, locating a child is not a constant-time operation, but it appears to work well in practice.

Since SWAP and EXCHANGE change just the ranks of the two lines directly involved they involve just two moves in the array implementation. The line at the lower level moves to the higher level while the line at the higher level comes down to the lower level.

Figure 6 shows the mapping from tree to array for the example of Figure 2. Here $M = 2$ and $n = 2$, and there are accordingly 7 rows in the array, with two entries per row. Levels are separated by double lines. The degree-2 lines 1011 and 0101 map to level 0. The degree-1 lines 1000 and 0110 map to the

[5]The forest is represented by duplicating the tree structure.

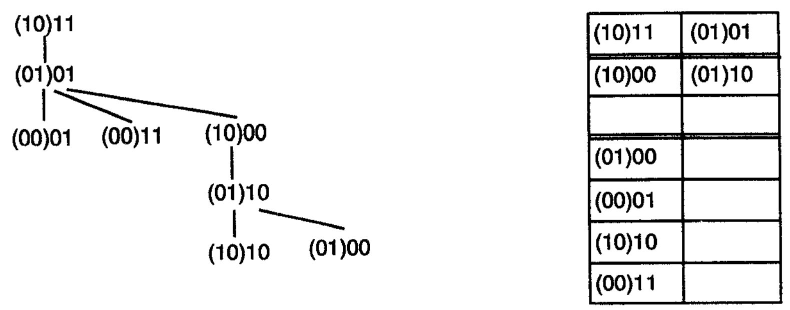


Fig. 6. Example mapping from gbt to array.

appropriate row in level 1. The degree-0 lines map to the appropriate rows in level 2. To search for line 0011, lines 1011 and 0101 are examined first in level 0. Since the tag match is not complete with either, the search moves to level 1. In level 1, 0011 (the incoming line) maps to row 1, but this row has no entry. So no lines are examined there, and the search moves to level 2. In level 2, 0011 maps to row 11, and there is a complete set and tag match. This search took 3 comparisons which is the same as earlier.

In GBF_LS, the RM_Count array is processed at the end of each successful search, which is an $O(M)$ operation. The following microoptimization allows this step to be skipped while the trace is processed, at the expense of a postprocessing step at the end of the simulation. The RM_Count array has $M + 1$ locations, and the maximum count at each location is n; and so there are $(n + 1)^{(M+1)}$ possible count combinations. During simulation, the number of times each combination occurs is maintained in an array of size $(n + 1)^{(M+1)}$. The array is processed at the end of the simulation to obtain the miss ratios. Since the size of the array is exponential in M, i.e., $(n + 1)^{(M+1)}$, the range of caches that may be simulated is limited by the available memory. This optimization may be turned off to simulate an arbitrary range of caches.

When a node is examined in GBF_LS, a right match is required. Right match is most efficiently implemented using the *first-one* function,[6] with the first-one function being implemented as a table lookup.

3.5 Complexity

In this section analytical comparisons of the three algorithms—FS + , AA, and GBF_LS—are presented. The complexity is presented in terms of the number of lines each algorithm examines to process a reference. We comment on the costs of examining a line at the end of the section. We ignore the processing of the right-match counts array that is required in AA and GBF_LS. As noted earlier, this step may be skipped in GBF_LS when

[6]The first-one function gives the position of the first nonzero bit starting from the least significant bit; right match (a, b) = first-one (a XOR b).

sufficient memory is available. Throughout, the expressions are for simulating caches $C_{S+i}^{L}(n)$, $i = 0, \dots, M$.

$O((M + 1)n)$ comparisons are required in the worst case in GBF_LS and in FS + . In FS + this worst case occurs when the reference misses in all the caches, and in GBF_LS it occurs when all combinings occur with $(n_1, n_2) = (n, 0)$ or $(0, n)$ and when the search is along the longest path from the root to a leaf. $O(n2^M)$ comparisons are required in the worst case in AA simulation (assuming that there are no additional nodes in the stack), where the worst case occurs when the reference misses in all the caches.

GBF_LS never makes more comparisons than either FS + or AA. This may be justified, as follows. In GBF_LS, lines that are *in* sets that the incoming address x maps to, and referenced after the earlier reference to x, are examined once. The FS + algorithm examines the same set of lines as GBF_LS, but it examines some lines more than once. In AA, lines that *map* to the same sets as x and that are referenced after the earlier reference to x are examined once. The lines that GBF_LS examines are clearly a subset of the lines that AA examines.

Owing to locality characteristics of the traces, worst-case complexity is not of much practical importance. Below we give expressions for complexity in terms of miss ratios. While the expression is exact for FS + , the expressions for GBF_LS and AA are derived under some uniformity assumptions. The miss ratio of cache $C_{S+k}^{L}(n)$ is denoted as $m_{k,n}$.

The number of comparisons per trace reference for FS + is

$$1 + 2\sum_{i=0}^{M-1} m_{i,1} + m_{M,1} + \sum_{i=0}^{M}\sum_{j=2}^{n-1} m_{i,j}, \quad n > 1.$$

Intuitively, each miss at the first line in the set costs two comparisons (except at the cache with M sets): one to check the next line in the same set and one to check the first line in the next cache. The rest of the misses, except the miss at the last line in a set, cost one comparison to check the next line in the set. A miss at the last line in a set does not cost any extra comparisons.

To derive similar expressions for GBF_LS and AA, we need to relate miss ratios with positions in the tree or stack. We identify every cache in the simulation space by the ordered pair (number of sets, associativity) and define a lexicographic ordering on them, i.e., $(i, j) < (k, l)$ if $i < k$ or $i = k$ and $j < l$. Also, every node in the gbf corresponds to a group of caches in the space of caches considered, in the sense that complete set and tag matches at that node imply a hit in that group of caches. We define the minimal cache for a node as the lexicographically lowest cache in the group of caches corresponding to the node. In deriving the complexity expression for GBF_LS we assume that all combinings are of type $(n/2, n/2)$, and that the probability of hitting at a node is the probability of hitting the minimal cache of the node minus the probability of hitting the minimal cache of the node above it. For AA we assume first that the stacks do not have any lines not in the gbf, so that each entry in the stacks corresponds to a node in the gbf, and then assume that the probability of a line being hit is the same as that of the corresponding gbf node. We assume further that the entries are ordered in the stacks in increasing order of their minimal caches, so that entries that are more likely to be hit are closer to the top.

Under the above assumptions the number of comparisons per trace reference for GBF_LS is

$$1 + \sum_{j=1}^{n} m_{0,j} + \sum_{i=1}^{M} \sum_{j=n/2+1}^{n} m_{i,j} - m_{M,n}$$

and for AA is

$$1 + \sum_{j=1}^{n} m_{0,j} + \sum_{i=1}^{M} \sum_{j=n/2+1}^{n} 2^{i} m_{i,j} + \sum_{i=0}^{M-1} 2^{i-1} m_{i,n} - \left(2^{M-1} + \frac{1}{2}\right) m_{M,n}$$

Below we give differences between the expression for GBF_LS and those of the other two algorithms and present some intuitive explanations for the differences:

(Complexity of FS +) − (Complexity of GBF_LS).

$$\sum_{i=1}^{M} \sum_{j=1}^{n/2} m_{i,j} + \sum_{i=0}^{M-1} (m_{i,1} - m_{i,n})$$

The first term is for the difference that arises because FS + reexamines lines that it has already seen. The second term is for the extra comparison that FS + takes on a miss at the most recent referenced line, and for the extra comparison that GBF_LS takes on a miss at the last line.

(Complexity of AA) − (Complexity of GBF_LS).

$$\sum_{i=1}^{M} \sum_{j=n/2+1}^{n} (2^{i} - 1) m_{i,j} + \sum_{i=0}^{M-1} 2^{i-1} m_{i,n} - \left(2^{M-1} + \frac{1}{2}\right) m_{M,n} + m_{M,n}$$

Intuitively, the differences arise because AA examines all the degree-k nodes before it moves to degree-$(k-1)$ nodes, whereas GBF_LS examines only nodes along the search path.

GBF_LS and AA require a right match each time a line is examined, whereas FS + just requires a comparison. The search process is also slightly simpler in FS + , and the processing of the right-match counts array is not required in FS + . Memory management is difficult in AA because there is no localized way to determine which lines to throw away.[7] Hence large amounts of unnecessary memory may be required. Furthermore, the unnecessary nodes add to the number of comparisons that have to be made. A periodic

[7] The last stack entry cannot always be deleted since it is possible that it is one of the few mapping to a particular set, and is hence in some cache.

cleanup step is usually required to remove unnecessary nodes. The array implementations of FS + and GBF_LS use a fixed amount of memory.

4. DIRECT-MAPPED CACHES

In this section we describe a binomial tree-based[8] algorithm for simulating direct-mapped caches of a range of line sizes and a fixed cache size.

A fixed cache size implies a fixed tag size in direct-mapped caches. This permits us to prove an inclusion property between the tag store contents of caches in this class. Tag inclusion is not the same as data inclusion; the data locations a tag refers to depend on the set number and line size. However, tag inclusion may be exploited in a different way to obtain an efficient simulation algorithm. The following lemma is similar to Lemma 3.1.1, but it is for tag inclusion rather than data inclusion. The fixed cache size is 2^c.

LEMMA 4.1. *For each set p in C_S^{c-S}, there are exactly 2^k sets P in C_{S+k}^{c-S-k} such that $[p]_{tag} \subset [P]_{tag}$. The most significant S bits of the set numbers of the sets in P are identical, and give the set number of p.*

Note that in contrast to Lemma 3.1.1, $([P]_{tag} - [p]_{tag}) \cap [C_S^L]_{tag} \neq \phi$, since the same tag may be found in more than one set. A binomial tree may be constructed as done earlier by combining sets, but using the most significant bits to identify trees to be combined. When the caches to be simulated are C_{S+i}^{c-S-i}, $i = 0, \dots, M$, the following properties are true in the resulting binomial forest.

Property 4.2. In a subtree of degree k, the most significant $S + M - k$ bits of the set numbers are identical, and two subtrees of degree k differ in at least one of the most significant $S + M - k$ bits of the set numbers.

Property 4.3. The tag in a set of rank k is in the tag stores of caches C_{S+i}^{c-S-i}, $i = M - k, \dots, M$. The set number of the set containing this tag in C_{S+i}^{c-S-i} is given by the most significant $S + i$ bits of the set field.

Figure 7 illustrates the binomial tree representation of caches in this class. Both the binomial tree and conventional representations are shown for a range of four caches ($M = 3$) starting with a one-set cache ($S = 0$). In the binomial tree representation, the tag (within parentheses) and the set number with respect to C_3^0 are shown. The line in cache C_0^3 is the eight-byte line corresponding to the address 1010, i.e., the set of bytes addressed by 1000 through 1111. Address 1010 is at the root of the tree and is of rank 3. Similarly, the lines in C_1^2 are the four-byte lines corresponding to addresses 1010 and 0110, which are of rank 2 and greater in the tree and so on.

The simulation algorithm is shown as Algorithm BF_CS. On each reference the appropriate binomial tree is located using the field $x[c-1: c-S]$ of the address, and the tree is searched for the set s such that $s = x[c-1: c-S-M]$. The search procedure is similar to that of GBF_LS, except that

[8]When only direct-mapped caches are being simulated the gbt is of order one, i.e., it is an ordinary binomial tree.

Conventional Representation

Cache	Tag Store	Set No.
C_0^3	1	
C_1^2	1	0
	0	1
C_2^1	0	00
	1	01
	1	10
	0	11
C_3^0	0	000
	0	001
	1	010
	0	011
	1	100
	1	101
	0	110
	1	111

Binomial Forest Representation

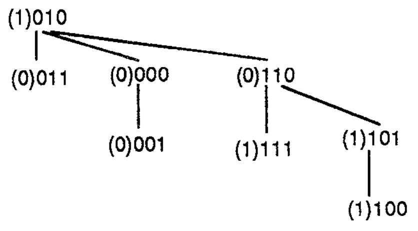


Fig. 7. Example binomial tree representation of varying line size direct-mapped caches.

left matches are used instead of right matches. However, the nonuniqueness of tags requires that tag comparisons be done and the hit array updated along the search path too. When a tag match occurs on the search path at a set p of rank k, the tag is in caches C_{S+i}^{c-S-i}, $i = M - k, \ldots, M$ (by Property 4.3). The corresponding set number in cache C_{S+i}^{c-S-i} is given by the most significant $S + i$ bits of the set field. If the left match between the set field of the incoming address and the set p is $S + t$, the address does not map to the set in C_{S+i}^{c-S-i}, $i = t + 1, \ldots, M$. So from the tag match and the left match at the set p it follows that the address hits in caches C_{S+i}^{c-S-i}, $i = M - k, \ldots, t$, and the hit arrays for those caches are updated.

```
*Algorithm* BF_CS
SIM_BF_CS()
  Initialize()
  **for** every reference $x$ in trace
    set_no_CM = Set number of address in $C_{S+M}^{c-S-M}$ ($x[c-1{:}c-S-M]$)
    set_no_level0 = Most significant $S$ bits of set_no_CM ($x[c-1{:}c-S]$)
    tag = Tag of address ($x[W-1{:}c]$)
    cur_node = Root[set_no_level0]
    found = 0
    **while** (NOT found)
      $k$ = Degree(cur_node)
      $t$ = Left_Match(cur_node $\rightarrow$ set_no, set_no_CM)
```

```
            if (cur_node → tag == tag)
             ++Hit_Array[$M - k$][$t$] /*Hit in caches $C^{c-(S+M-k)}_{S+M-k}$ to $C^{c-(S+t)}_{S+t}$ */
            end if
            if ($t == S + M$)   /* Complete Match found */
              cur_node → tag = tag
              found = 1
            else
              child_node = Child(cur_node, $S + M - t - 1$)
              SWAP(child_node)
              cur_node = child_node
            end if
          end while
          Root[set_no_level0] = cur_node
        end for
      Hits in $C^{c-S-k}_{S+k} = \sum_{i=0}^{k} \sum_{j=k}^{M} Hit\_Array[i][j]$
      Child($x, d$)
        Return(Child of $x$ rooted at tree of degree $d$)
      Left_Match($a, b$)
        Return(Min. $i$ such that $a[S + M - 1:S + M - 1 - i] \neq b[S + M - 1:S + M - 1$
          $- i]$)
      Initialize()
        Build $2^S$ binomial trees from $2^{S+M}$ sets
          (Using the combining procedure, assuming arbitrary ordering)
        Set all tags to invalid
```

Figure 8 shows the algorithm working on the example of Figure 7. The incoming address is 1111. The tree is located (in this example this is trivial). The left match at the root is determined to be zero. Since the tags match, the address hits in cache C^3_0. The degree-2 subtree at the root is swapped with the tree at its rank-2 child. The left match with the new root is 2. Since there is no tag match the address does not hit in C^2_1 or C^1_2. The rank-0 child is swapped with the root. Now the set and the tag field of the root and address match. The address hits in C^0_3.

5. EMPIRICAL PERFORMANCE EVALUATIONS

This section presents an experimental comparison of the performance of GBF_LS to AA and FS + .

5.1 Traces

The traces were obtained using the *pixie* utility provided on machines such as the *DEC3100* which uses the *MIPS* family of microprocessors. The programs traced are from the SPEC89 benchmark suite (except intmc). The SPEC89 suite is a collection of numeric and nonnumeric programs widely used for benchmarking microprocessor-based workstations. A mix of benchmarks was chosen including scientific code spice2g6, utilities gcc1.35, and integer benchmarks espresso and intmc. The trace is a combined instruction-

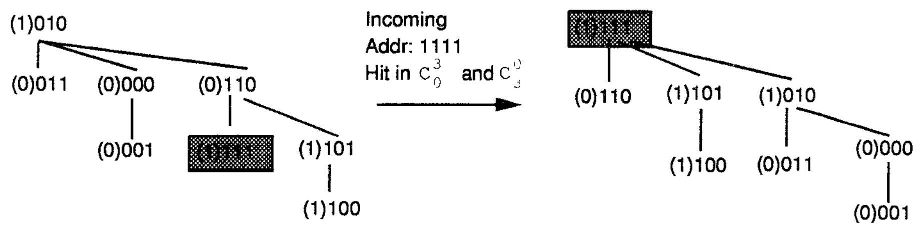


Fig. 8. Example illustrating Algorithm BF_CS.

Table I. Performance of Constant Line Size Set-Associative Cache Algorithms: Caches $C_i^3(j)$, $i = 7, \dots, 13$, $j = 1, \dots, 4$

	Fully associative stack			
Workload	Mean depth	SD	Top-of-stack (%)	Top-of-gbt (%)
gcc	3278	12287	4 56	32.17
espresso	620	3323	1.37	49.43
spice2g6	1640	4693	4.6	46.00
intmc	88	435	0	56.54

and-data trace. Reads and writes are handled identically. Some characteristics of the trace are shown in Table I. (The gbt of column 5 is for simulating caches $C_i^3(j)$, $i = 7, \dots, 13$, $j = 1, \dots, 4$, same as the range simulated in the experiments below.)

All the algorithms were implemented in C, GBF_LS was implemented using arrays as described in Section 4.3. The FS + algorithm was also implemented using arrays. The stack in AA simulation was implemented using a linked list, since its size cannot be bounded. Hits at the first line examined were treated as a special case, and processed fast in all three implementations. In all cases, the first 100 million references were simulated.

5.2 Experimental Results

Table II gives the set-associative cache simulation times and average number of comparisons for caches with a line size of 8 and with the number of sets ranging from 128 to 2048 and associativity ranging from 1 to 4.

The mean simulation time of AA is about a factor 2.4 greater than the mean simulation time of GBF_LS, and the mean number of comparisons of AA is about a factor of 2.3 greater than that of GBF_LS. For the gcc trace, the simulation time of AA is about a factor of 3.8 greater than that of GBF_LS. gcc references many distinct lines, and the resulting big stack is the cause for the poor performance of AA. In general when the working set size is large AA seems to perform badly.

The mean simulation time of FS + is about a factor of 1.22 greater than that of GBF_LS. The mean number of comparisons of FS + is about a factor of 1.585 more than GBF_LS. The ratio of simulation times is greater than

Table II. Performance of Constant Line Size Set-Associative Cache Algorithms: Caches $C_i^3(j)$, $i = 7, \ldots, 13$, $j = 1, \ldots, 4$

	No. of Comps.			Exec. Time		
Workload	FS+	All Ass.	GBF_LS	FS+	All Ass.	GBF_LS
gcc	4.7973	10.4914	2.6851	737.3	2286.9	605.5
espresso	2.2834	1.7155	1.5515	416.6	478.1	324.6
spice2g6	3.3290	4.0130	2.1007	554.4	878.6	478.7
intmc	1.6982	1.3959	1.3035	325.4	384.7	258.2
mean	3.027	4.404	1.9102	508.4	1007 1	416.8

the ratio of mean comparisons, because FS + does not require right matches and because its search procedure is simpler. The mean simulation time of FS + is less than that of AA. The good performance of FS + is probably because of the good locality that SPEC traces show typically.

Tycho (written by M. D. Hill), a public-domain simulator using the AA algorithm, took 6993.0 sec. for simulating the first 100 million references of spice2g6, for the same caches as in Table II. It is unclear what contributes to this factor of 8 difference between the simulation time of our implementation of AA and Tycho. We believe that most of the difference occurs in input processing; specifically the ASCII to binary conversion required in Tycho. Since most trace references hit at the smallest cache, very little simulation work is required in most cases, and the input overhead dominates. Our implementation of AA reads addresses as binary numbers in chunks of 1000, processes smallest cache hits as a special case, and does the right match as an array lookup. This shows that in addition to the simulation algorithm, simulation time is also significantly affected by factors such as input processing.

In Table III, the performance of BF_CS is compared to a naive simulation algorithm where all the caches are examined on each reference. The simulation of a range of 6 caches of size 16 KB and line sizes ranging from 8 bytes to 256 bytes is considered. BF_CS outperforms the naive algorithm consistently by about a factor of 1.7. BF_CS takes about 2 comparisons per trace reference; this indicates good spatial locality in the trace. For the naive algorithm, of course, the number of comparisons is constant at 6.

6. CONCLUSION

The increasing difference between CPU and memory speeds has made caches important. This has resulted in larger caches and greater interest in cache design. In this article we have presented an algorithm to simulate a class of set-associative caches of varying number of sets and associativity, but constant line size in a single pass through an address trace. This algorithm is based on a representation of the caches using a novel data structure called the generalized binomial tree (gbt) which is an extension of the ordinary binomial tree. Analytically it is shown that the number of lines the new

Table III. Performance of Variable Line Size Direct-Mapped Cache Algorithms: Caches C^{i}_{14-i}, $i = 3,\ldots,8$

	No. of Comps.		Exec. Time (secs)	
Workload	Naive	BT	Naive	BT
gcc	6.0	1.9530	802.7	468.5
espresso	6.0	1.9309	789.0	465.6
spice2g6	6.0	1.9500	802.5	464.4
intmc	6.0	2.0566	806.7	499.2
mean	6.0	1.9726	800.2	474.4

algorithm examines on a reference is the same as or fewer than AA and FS + . Empirical evaluations show the new algorithm running faster than AA and FS + by factors ranging from 1.2 to 3.8. We also present a single-pass simulation algorithm for direct-mapped caches of a range of line sizes, but fixed line size. Empirical evaluations show this algorithm running about 2 times faster than the naive algorithm.

There are two major directions in which the work may be extended. First, the data structures introduced here may be used to develop single-pass algorithms for other classes of caches, such as those used in multiprocessor systems. Second, the gbt representation may be used to develop models for set-associative cache behavior, which could lead to improved insight into the caching process.

APPENDIX

Generalized Binomial Trees

Here we define terms such as rank, degree, and subtree, and prove some properties for gbts. (See Definition 3.2.1.) We illustrate the definitions below using Figure 9. The figure shows a gbt(2) of degree 2 being constructed by combining gbt(2)s of lesser degrees. The top figure shows four gbt(2)s of degree zero. The next figure is after one combining step and shows two gbt(2)s of degree one. The bottom figure shows one gbt(2) of degree two.

LEMMA A.1. *A gbt(n) of degree x has $n2^x$ nodes.*

PROOF. By induction on the degree. □

Definition A.2. The *rank* of a node in a gbt is defined to be[9]

$$\log\left(\left\lceil \frac{\text{Number of descendants (inclusive)}}{n} \right\rceil\right).$$

Definition A.3. The *tree* rooted at a node in a gbt consists of the node and all its descendants.

[9]All logarithms are to base two.

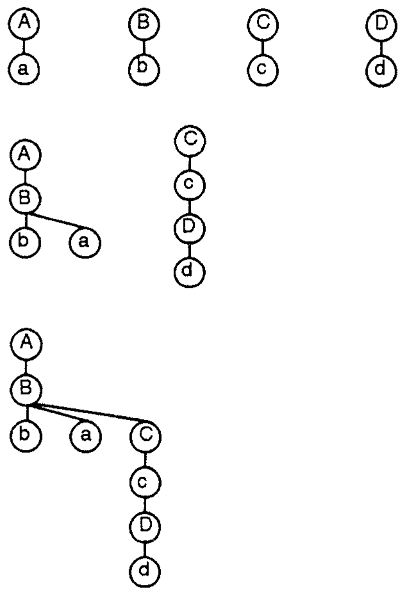


Fig. 9. Example gbt illustrating definitions.

All the trees rooted at nodes contained in a gbt are not gbts. The following definition of *degree* is general and is applicable to any tree contained in a gbt. The lemma following the definition proves that this definition of degree is consistent with its usage in the definition of gbts.

Definition A.4. The *degree* of a tree is the rank of the root of the tree.

LEMMA A.5. *All the nodes in the root list of a gbt of degree x are of rank x.*

PROOF. There are n nodes in the root list. The rank of the top node is: $\log n2^x/n = x$. The rank of the last node is:

$$\log\left\lceil \frac{n2^x - (n-1)}{n} \right\rceil = \log\left\lceil 2^x - 1 + \frac{1}{n} \right\rceil = x.$$

Clearly, the ranks of the other nodes in the root list are also x. □

Definition A.6. The *list child* of a node in a gbt is that child which at some stage in the combining process was a child of that node in the root list. A *tree child* of a node in a gbt is any child that is not the list child.

Referring to Figure 9, in the degree-2 tree, B is the list child of A, and b is the list child of B. Node c does not have a list child.

LEMMA A.7. *A node can have at most one list child.*

PROOF. A node gets a list child only when it is one of the top $n - 1$ nodes in the root list. But a node in the top $n - 1$ of the root list has only one child. So when a node gets a new list child during the construction procedure, it loses its previous list child if it had one. □

LEMMA A.8. *(i) A node of rank 0 in a gbt either has no children (leaf node) or has one list child. (ii) a node of rank k, $k > 0$, (a) has exactly k tree children of ranks 0 through $k - 1$, if it does not have a list child, or (b) has exactly r, $r \leq k$, tree children of ranks $k - r$ through $k - 1$, if it has a list child of rank $k - r$.*

PROOF. Both parts (i) and (ii) are proved by induction on the degree of a gbt.

Part *(i)*. A gbt of degree 0 is a list, and (i) is clearly true. When gbts of degree 0 are combined to form a gbt of degree 1, as noted before, there is a change in the children of only the nodes in the root list of the resulting tree, but these nodes are of rank 1 and are not covered by this part. The same argument applies for further combinings, and so nodes of rank zero either have no child or one list child.

Part *(ii)*. Consider a gbt of degree 1. The nodes in the root list excepting the last node have one list child of rank one and no tree children. The last node either has one list child of rank zero and one tree child of rank zero, or no list child and one tree child of rank zero. The lemma thus holds for the nodes in the root list (the only ones of rank greater than zero).

Assume the statement is true for a gbt of degree j (≥ 1). A gbt of degree $j + 1$ is formed by combining two gbts of degree j. All nodes not in the root list of the degree-($j + 1$) gbt have the same children as they did before combining. The lemma holds for such nodes by the induction assumption. The nodes in the root list of the degree-($j + 1$) gbt, excepting the last node, have only list children of rank $j + 1$; the lemma is hence true for these nodes. The rank of the last node in the root list is $j + 1$ after the combining. If this node has a list child of rank $j - r$ (or does not have a list child), by the induction assumption the node has tree children of rank $j - r$ to $j - 1$ (or 0 to $j - 1$); the new child is obtained as a result of the combining is of rank j. The lemma thus holds for the last node of the root list too. The proof follows by induction. □

The following defines the subtrees of a node in a gbt.

Definition A.9. A node of rank k has a subtree of degree r, $r < k$, *iff* it has a tree child of rank r, and this subtree is the tree left after pruning tree children of rank r and greater from the tree rooted at the node. A node of rank k always has a subtree of degree k which is the tree rooted at the node.

It follows from Definition A.9 that the largest subtree at a node is the tree rooted at the node, and the smallest subtree at a node is tree left after pruning all tree children from the node. Referring to the degree-2 gbt in Figure 9, the subtrees rooted at B are $\{B, b\}$ of degree zero, $\{B, b, a\}$ of degree one, and $\{B, b, a, C, c, D, d\}$ of degree two. The subtrees rooted at c are $\{c\}$ of degree zero, and $\{c, D, d\}$ of degree one. The following lemma specifies precisely the subtrees that a node has.

LEMMA A.10. *If a node of rank k in a gbt does not have a list child, it has subtrees of degree 0 through k. If the node has a list child of rank $k - r$, it has subtrees of degree $k - r$ through k.*

PROOF. Follows from the definition of subtrees and Lemma A.8. □

In the definition of subtrees, *degree* is defined in the context of the children that are removed. It may be shown that this definition of degree for a subtree is consistent with the earlier definition of degree for a tree so that

$$\text{Degree of subtree} = \log\left\lceil \frac{\text{Number of nodes in the subtree}}{n} \right\rceil.$$

ACKNOWLEDGMENTS

We thank the reviewers for their helpful suggestions.

REFERENCES

BENNETT, B. T AND KRUSKAL, V. J. 1975. LRU stack processing. *IBM J. Res. Devel.* (July), 353–357.

HEIDELBERGER, P. AND STONE, H. S. 1990. Parallel trace-driven cache simulation by time partitioning. In *Proceedings of the Winter Simulation Conference* IEEE, New York, 734–737.

HILL, M. D AND SMITH, A. J. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38*, 12 (Dec.), 1612–1630.

LAHA, S., PATEL, J. H., AND IYER, R. K. 1988. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. Comput. C-37*, 11 (Nov.), 1925–1936.

MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9*, 2, 78–117.

OLKEN, F. 1981. Efficient methods for calculating the success function of fixed space replacement policies. Tech. Rep. LBL-12370, Lawrence Berkeley Laboratory, Berkeley, Calif.

PUZAK, T. R. 1985. Analysis of cache replacement algorithms. Ph.D. thesis, Univ. of Massachusetts, Amherst, Mass.

SMITH, A. J. 1985. Disk cache—miss ratio analysis and design considerations. *ACM Trans. Comput. Syst. 3*, 3, 161–203.

SMITH, A. J. 1982. Cache memories. *ACM Comput Surv. 14*, 3 (Sept.), 473–530.

SMITH, A. J. 1977. Two methods for the efficient analysis of memory address trace data. *IEEE Trans. Softw. Eng. SE-3*, 1 (Jan.), 94–101.

STONE, H. S. 1987. *High-Performance Computer Architecture.* 2nd ed. Addison-Wesley, Reading, Mass.

SUGUMAR, R. A. 1993. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Ph.D. thesis, Univ. of Michigan, Ann Arbor, Mich.

SUGUMAR, R. A. AND ABRAHAM, S. G. 1993. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of ACM SIGMETRICS Conference.* ACM, New York, 24–35.

THOMPSON, J. G. 1987. Efficient analysis of caching systems. Ph.D. thesis, Univ. of California, Berkeley.

TRAIGER, I. L. AND SLUTZ, D. R. 1971. One pass techniques for the evaluation of memory hierarchies. Tech. Rep. RJ 892, IBM, Armonk, N.Y.

WANG, W.-H. AND BAER, J.-L. 1990. Efficient trace-driven simulation methods for cache performance analysis. In *Proceedings of ACM SIGMETRICS Conference.* ACM, New York, 27–36.

Received January 1992; revised September 1993; accepted March 1994