Efficient Simulation of Multiple Cache Configurations using Binomial Trees

(Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991)

Rabin A. Sugumar and Santosh G. Abraham
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122.

ABSTRACT

Simulation time is often the bottleneck in the cache design process. In this paper, algorithms for the efficient simulation of direct mapped and set associative caches are presented. Two classes of direct mapped caches are considered: fixed line size caches and fixed size caches. A binomial tree representation of the caches in each class is introduced. The fixed line size class is considered for set associative caches. A generalization of the binomial tree data structure is introduced and the fixed line size class of set associative caches is represented using the generalized binomial tree. Algorithms are developed that use the data structures to determine miss ratios for the caches in each class. Analytical and empirical comparisons of the algorithms to previously published algorithms such as all-associativity and forest simulation are presented. Analytically it is shown that the new algorithms always perform better than earlier algorithms. Empirically, the new algorithms are shown to outperform earlier ones by factors of 1.0 to 5.0.

Key Words: Trace-driven simulation, direct mapped caches, set associative caches, binomial tree, inclusion properties, single-pass simulation, cache modeling

1 Introduction

CPU caches are small fast memories located between the CPU and the main memory. They contain a small subset of the contents of the main memory. CPU accesses are satisfied by the cache for the most part. The main memory is accessed only if the requested memory location is not present in the cache. Since the time to access the cache is usually much lower than the time to access main memory, caches help decrease the effective memory access time. There are excellent discussions on caches in the literature, for instance [13, 4].

In this paper new techniques for the efficient simulation of caches are presented. These techniques are expected to be helpful in the cache design process.

An important aspect of cache design at the architectural level is deciding on the three parameters: cache size, line size and degree of associativity. The optimality of the cache parameters depends on the workload to be run on the machine; 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 the trace length. Two factors contribute to large trace lengths: *cold start* misses and variation in trace characteristics. Cold start misses are the misses that occur at the start of the simulation on references to new addresses, because the cache is initially empty. If the trace is too short, cold start misses will account for a disproportionate fraction of the total misses and the simulation results can underestimate the performance of the underlying system. The second factor contributing to big traces is the difference in trace characteristics in different phases of the program. When cache evaluation is done with a small trace representing one phase of the program, the results are not reliable. This is illustrated in [2]. Complete traces of workloads, billions of addresses long, are simulated and it is shown that if smaller segments of the complete traces are used, the results can vary greatly depending on which segment of the trace is simulated.

Single-pass simulation algorithms are one approach to minimizing the time taken in doing cache simulation. In single-pass simulation more than one cache configuration is simulated at the same time; for instance multiple direct mapped caches of varying size but of the same line size. Single-pass simulation is efficient when properties relating the contents of one cache to another are exploited. Data structures that reflect these relations between caches facilitate the development of efficient algorithms. There are two other approaches to decreasing the time of cache simulation. One approach is to use reduced traces. A reduced trace is a shorter version of the original trace obtained by sampling or other means, so that the cache performance on the reduced traces matches or is a close approximation to that on the original trace. The other approach is parallel cache simulation, where the cache configurations are simulated in parallel. Single-pass simulation can be used in combination with these other approaches.

Apart from the improvement in simulation time, there are two other benefits to developing single-pass algorithms. Firstly, the development of single-pass algorithms requires a focus on the relations between cache configurations. This leads to a better understanding of the differences between alternate cache configurations. Secondly, single-pass simulation enables a differential comparison between caches. For example it is possible to instrument a single-pass program to output software events that cause a performance difference between two cache configurations. Such data is useful both in cache design and in compiler development.

Preliminaries

A two-level memory hierarchy model is assumed with a single cache and main memory. The trace is a list of addresses and the simulation algorithms output the miss ratio of each cache.

The line size is the same as the fetch size[1]. So on each miss all the bytes in the missing line are fetched. Subsequent references to any byte in the line are hits. There is no prefetching.

Bit selection mapping is assumed. LRU replacement is assumed where the degree of associativity is greater than one.

The output of the simulation algorithm is the miss ratio in each configuration simulated. The miss ratio may be used to get the effect of the cache on the CPI (cycles per instruction) to a first approximation. Miss ratio is not the only factor which determines the effect of the cache on the CPI; other factors include the distribution of misses, the number of dirty misses, and the CPU design. The miss ratio could be used to restrict the range of configurations of interest. More thorough simulation may be done later on this smaller subset of configurations.

Overview of Paper

The rest of the paper is organized as follows. Section 2 reviews related work and contrasts our results with that of others. In Section 3 an algorithm is presented that uses data inclusion properties to simulate multiple configurations of direct mapped caches with fixed line size and varying number of sets. A dual algorithm is then presented that uses tag

[1] Line size and fetch size are used as defined in [10]. Line (Block) size is the unit of data for which there is an address tag. Fetch size is the number of bytes fetched on a miss.

inclusion properties to simulate direct mapped caches with fixed size and varying line size. In Section 4 the fixed line size algorithm is extended to multiple configurations of set-associative caches of varying associativity by using a novel generalized binomial tree structure. All the algorithms reported in this paper including previously developed competing algorithms have been implemented. Section 5 reports on the empirical comparisons of the various algorithms done using these implementations.

2 Related Work

There is a naive single-pass simulation method which is applicable in the simulation of any collection of cache designs. In this naive simulation, a separate data structure is maintained for each cache configuration. Each data structure is inspected and updated for each address in the trace. Compared to the simulation of cache designs in separate passes, this method amortizes the amount of I/O required (if the trace is stored on a disk), or trace generation time (if the trace is generated on-the-fly). The single pass simulation methods reviewed in this section and those introduced in the paper utilize inclusion properties within certain classes of caches and are more efficient than this naive algorithm.

The initial work on single pass simulation of memory hierarchies was done by Mattson et al., [9] at IBM in the context of virtual-memory systems. They describe an algorithm for simulating a range of fully associative caches of varying sizes but fixed line size. They do this by proving that the contents of a fully associative cache is a subset of the contents of all larger fully associative caches with the same line size (called an inclusion property) for a class of replacement policies. In their algorithm the state of all caches is maintained in a single stack. For each incoming address, caches are examined in increasing order of size. By the inclusion property, once the address is known to hit a cache, the address will also hit in all bigger caches.

Hill and Smith present the *forest simulation* algorithm in [7], for simulating direct mapped caches of varying sizes but fixed line size. This algorithm also uses an inclusion property. Caches are searched from the smallest to the largest for each incoming address and the search is stopped once the address hits in a cache. In Section 2, we introduce an algorithm based on binomial trees for simulating the same class of caches. The new algorithm takes fewer comparisons than forest simulation on the average.

Single-pass simulation of caches with varying line sizes has not been considered much

in the literature. The only work that the authors are aware of is that of Slutz and Traiger [12], where the original algorithm of Mattson et al. is extended to handle multiple line sizes. In Section 2 we present an algorithm based on the binomial tree to simulate direct mapped caches of the same size but varying line sizes in one pass. Such a collection of cache designs is expected to occur frequently in the design process, because important constraints for physical (and virtual) caches fixes the primary cache size to the virtual memory page size [4]. The algorithm uses a novel inclusion property between tag stores.

A method for simulating set associative caches of varying numbers of sets and varying associativities is described by Mattson et. al. [9]. This algorithm assumes that the set mapping is done by bit selection. This algorithm is generalized to work for set mapping functions other than bit selection by Hill and Smith [7] and is referred to as *all-associativity simulation* in the following. In Section 3 we present an algorithm to simulate the same class of set associative caches using a generalized form of the binomial tree. The worst case number of comparisons per address, of the new algorithm, is linear in the number of caches simulated, whereas that of all-associativity simulation is exponential in the number of caches simulated. More importantly, the new algorithm never takes more comparisons and outperforms all-associativity simulation significantly on practical traces, especially when the associativity is low. The new algorithm also handles memory better than all-associativity simulation.

Another technique for speeding up simulation is to use reduced traces. A reduced trace is obtained by passing the original trace through a filter. The reduced trace is significantly smaller than the original trace but has sufficient information to give accurate results for the caches of interest. In [11] a technique along these lines is presented for simulating direct mapped and set associative caches of a fixed line size. But the results are not accurate if caches of a different line size are simulated. Wang and Baer [18] suggest obtaining a *universal* reduced trace which is a union of reduced traces for various line sizes. They note that this universal reduced trace is only about 50% bigger than a single line size reduced trace. Laha et.al. [8] present a sampling technique to obtain reduced traces that approximate the behavior of the original trace. The single-pass simulation approach is complementary to the reduced trace approach. All the algorithms described here are compatible with reduced traces, i.e., the input trace to the single-pass algorithms can be a reduced trace. Parallelizing cache simulation is another approach to decreasing the simulation time. A simple parallelizing scheme is to simulate one cache configuration in each processor. But

extra work is done compared to single-pass simulation, since relations between caches are not exploited. Single-pass algorithms may themselves be parallelized.

3 Simulation of Direct Mapped Caches

The parameters of a direct mapped cache include the line size (LS), the number of sets (NS) and the cache size (CS) with the relation $CS = LS \times NS$. Single-pass simulation of direct mapped caches with constant LS and varying NS is considered in the first subsection. Single-pass simulation of caches with constant CS but varying NS is considered in the subsequent subsection.

The following notation is used. A direct mapped cache with 2^S sets and line size 2^L is denoted as C_S^L (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. $[X]_{tags}$ denotes the tags contained in the tag store corresponding to X. 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 Constant LS - Varying NS

In this subsection we consider the single-pass simulation of a range of direct mapped caches C_{S+i}^L, $i = 0, \ldots, M$, with a fixed line size 2^L, and varying number of sets. In the first part of the section relations are proved between the contents of caches in the class and the data structure and algorithm are presented. In the next part implementation issues are discussed and in the last part the complexity of the new algorithm is compared with earlier algorithms.

The terms *address* and *line* are used interchangeably in this subsection. This is acceptable because the line number[2] $a = x[W-1:L]$ of an address x is the same for all the caches.

[2]Line number is the part of the address left after truncating the line field.

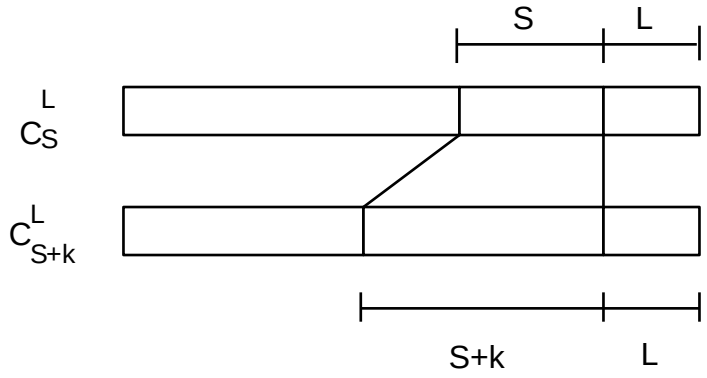


Figure 1: Bit fields in C_S^L and C_{S+k}^L

Cache Relations and Data Structure

A lemma relating the contents of two caches with the same line size but varying number of sets is proved below. A previously known inclusion property follows as a corollary to this lemma. Based on this inclusion property, an algorithm has been developed earlier; this algorithm is reviewed. The data structure that we use, which is based on this lemma, is then introduced.

Lemma 1: For each set p in C_S^L there are exactly 2^k sets P in C_{S+k}^L such that $[p]_{lines} \subset [P]_{lines}$ and $([P]_{lines} - [p]_{lines}) \cap [C_S^L]_{lines} = \phi$.

Proof :

C_{S+k}^L uses k extra bits for selecting a set when compared to C_S^L, as shown in Fig. 1. So lines mapping to a single set p in C_S^L map to one of 2^k sets in C_{S+k}^L, given by $P = \{s \text{ s.t. } s[S-1:0] = p\}$. Only one of the 2^k sets of C_{S+k}^L is present in C_S^L and that is the set that was updated last, i.e., $[p]_{lines}$ is the same as the contents of that set in P which was updated last. Therefore, $[p]_{lines} \subset [P]_{lines}$.

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

The following corollaries are stated without proof. They follow directly from the lemma and the proof.

Corollary 1

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

Corollary 2

$[C^L_{S_1}]_{lines} \subseteq [C^L_{S_2}]_{lines}$, if $S_1 \leq S_2$.

Corollary 2 above is the same as Corollary 5 of Theorem 1 in [7]. The inclusion property of Corollary 2 is exploited in forest simulation [7]. In forest simulation for each of the caches being simulated, a separate array with the contents of the tag-store is maintained. [3] A hit-array with one entry for each cache is also maintained. An address is read from the trace and the tag-store arrays of the caches are searched starting from the smallest. If the tags do not match in a particular cache, the old tag is replaced with the new tag. Otherwise, the corresponding hit-array entry is incremented and bigger caches are not searched. Since larger direct mapped caches contain smaller ones, the line will be present in all bigger caches. After all the trace addresses have been processed, the number of hits in any cache is obtained by summing the hit counts of all caches in the hit-array, excluding those for bigger caches.

The new algorithm presented here uses a binomial forest data structure. Lemma 1 is the basis of a recursive combining procedure which builds up this data structure. Consider the sets in some cache. Set pairs are formed such that each pair consists of sets which differ only in the most significant bit of their set numbers. In each pair, the more recent set is placed above the other. The contents of the next smaller cache, with half the number of sets, is the same as the contents of those sets that are at the top of their pair. This combining may be done recursively so that, at each stage the contents of the sets at the top represent the contents of some smaller cache. The structure that results from such a combining procedure is a forest of binomial trees.

Binomial trees may be defined inductively as follows [17] (Fig. 2): A binomial tree of degree[4] 0 (B_0 in Fig. 2) has one node. A binomial tree of degree x (B_x) is built by connecting the roots of two binomial trees of degree $x-1$ (B_{x-1} and B'_{x-1}) with an edge. The number of nodes in a binomial tree is therefore 2^{degree}. B_{x-1} is called the *subtree* of degree $x-1$

[3] A representation as a forest of balanced binary trees is also possible, hence the name.

[4] In literature the term index is commonly used instead of degree. For this application "degree", with its implication of size, seems more appropriate.

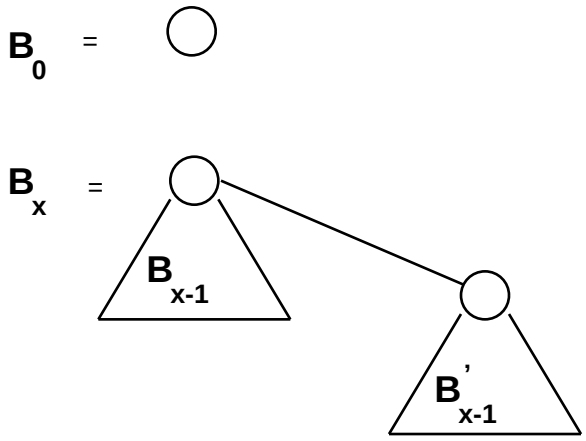


Figure 2: Definition of the binomial tree

of B_x. Similarly B_{x-1} has a subtree of degree $x-2$, B_{x-2}. B_{x-2} is also called a subtree of degree $x-2$ of B_x. Similarly B_x has subtrees of degree $x-3$, $x-4$, ...0. B_x is the maximal subtree (degree x) of itself. The tree rooted at a node refers to the maximal subtree rooted at that node. [5] The *rank* of a node is defined as the degree of the tree rooted at the node.

It can be shown that any node in a binomial tree of rank k has children of rank 0 to $k-1$. The following is another definition of subtree for a binomial tree. A subtree of degree $k-r$ at a node of rank k in a binomial tree consists of the node and the trees rooted at children of rank 0 to $k-r-1$. The children are attached to the node as in the original tree.

Below we describe the combining procedure for building up the binomial forest more formally. Let C^L_{S+i}, $i = 0, 1, \ldots, M$ denote the caches to be simulated. The binomial forest representing the caches is obtained using the combining procedure as follows:

Combining Procedure

Input: Sets in cache C^L_{S+M} and the order in which they were updated.

Procedure:

1. Form a forest of trees of degree one by connecting pairs of sets s_1 and s_2 with an edge, with the more recently updated set as the root, if $s_1[S+M-2:0] = s_2[S+M-2:0]$.

2. For $i = 2$ to M,

Form a forest of trees of degree i by connecting the roots of pairs of trees t_1 and t_2 of degree $i-1$ with an edge, with the more recently updated root as the new root, if

[5] The term subtree is used in a specific sense in the paper. For instance, B'_{x-1} is not a subtree of B_x.

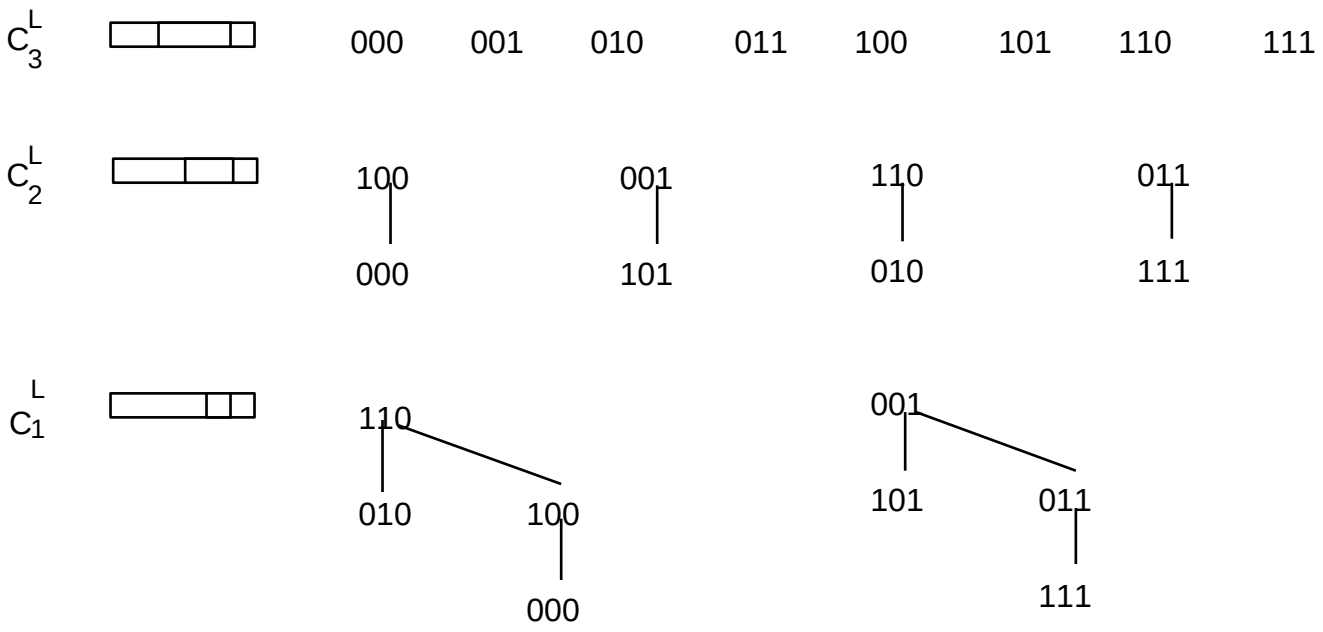


Figure 3: Recursive combining of sets

$s_1[S+M-i-1:0] = s_2[S+M-i-1:0]$, where s_1 is any set in t_1 and s_2 is any set in t_2. *Output:* Forest consisting of 2^S trees of degree M.

This procedure is illustrated in Fig. 3, where $S = 1$ and $M = 2$. The sets in the top row are the sets in C_3^L. Each set has an associated tag indicating the contents of the set, but the tags are not important in the present discussion and are not shown. The data structure after one combining step is shown in the second row. The set at the root is the most recently updated in each tree. The contents of the sets at the root are the contents of C_2^L. The corresponding set numbers in C_2^L is obtained by stripping the most significant bit off the set number in C_3^L. The result of a second combining step is shown in the third row. Now the contents of the sets at the root are the contents of C_1^L. The set number in C_1^L is obtained by stripping off the two most significant bits from the set number in C_3^L.

The binomial forest obtained using the combining procedure has the following properties. In the following, set number and tag are with respect to C^L_{S+M} unless stated otherwise.

Property 1 In a subtree of degree k, the least significant $S+M-k$ bits of the set numbers are identical, and two subtrees of degree k differ in at least one of the least significant $S+M-k$ bits of the set numbers.

Property 2 The line in a set of rank k is in caches C^L_{S+i}, $i = M-k, \ldots, M$.

The validity of the two properties may be seen by considering the state of the structure just before trees of degree k are combined. The contents of the root sets at this stage are the same as the contents of cache C^L_{S+M-k}. By the inclusion property the line in a root set is in all bigger caches too, i.e., caches C^L_{S+i}, $i = M-k+1, \ldots, M$. This is Property 2. Since at this point all the sets in a tree map to the same set in C^L_{S+M-k} the least significant $S+M-k$ bits of the sets in a tree have to be same and between trees they have to be different. This is Property 1.

One operation is defined on the binomial forest data structure, SWAP. If v is a node of rank k in a binomial tree, SWAP(v) exchanges the tree rooted at v with the subtree of degree k rooted at its parent. SWAP(v) is undefined if v does not have a parent. It is seen that SWAP(v) reverses the order in which the tree rooted at v and the tree rooted at its parent are combined.

Algorithm BF_LS (Binomial Forest – fixed Line Size) simulates the caches $C^L_{S+i}, i = 0, \ldots, M$. The algorithm uses the binomial forest data structure outlined above. At each node of the binomial forest the corresponding set number and the tag at the set in C^L_{S+M} are stored. When an address x comes in, the algorithm uses the set number of the address in the smallest cache $x[S+L-1:L]$ to locate the binomial tree that could contain it. The tree is then searched for the set s such that $s = x[S+M+L-1:L]$. At any point in the search procedure, if the right-match[6] between the set number at the node being examined and $x[S+M+L-1:L]$ is k, the child to be searched next is the child of rank $S+M-k-1$. The reason for this is as follows: Consider B_{S+M-k}, the subtree of degree $S+M-k$ at p. Any set outside B_{S+M-k} would differ from any set inside in at least one of the least significant k bits (Property 1). Since the right-match of $x[S+M+L-1:L]$ with the root

[6] *Right-match* between two set fields s_1 and s_2 is defined as the minimum i such that, $s_1[i:0] \neq s_2[i:0]$.

of B_{S+M-k} is k, s is in B_{S+M-k}. By the definition of subtrees, B_{S+M-k} has the children of rank 0 to $S+M-k-1$ of p. But the children of rank less than $S+M-k-1$ are part of smaller subtrees of p and have right-matches greater than k with p; so by Property 1, s is not in the trees rooted at these children. Therefore s is in the tree rooted at the rank $S+M-k-1$ child and this child is searched next. This child has a right-match of k with p and so it has a right-match of at least $k+1$ with $x[S+M+L-1:L]$. Note that in this justification it is assumed that p is of rank greater than or equal to $S+M-k$ so that it has a child of rank $S+M-k-1$. But this is true because at the start of the search, the right-match at the root of the binomial tree is at least S and the root is of rank M. Subsequently when the search moves to a child of rank $S+M-k-1$, the right-match at the child is at least $k+1$.

Once the set s is located its tag is compared with the tag of the address. If the tags match the rank of the set is noted in the hit-array. The caches the address hits may be determined from the rank using Property 2. If the tag is different the address does not hit in any cache. The tag at the set is updated. In either case, the set s is now the most recently updated set and should move to the top of the tree. This is accomplished through a series of SWAPs, till the set reaches the top. [7]

An illustrative example is given in Fig. 4. Here $S=0$ and $M=3$. The line field is not shown and L may be taken to be 0. The state of the caches before address 0101 comes in, as well as the state after, are shown. Both the conventional and binomial forest representations are shown. The tag is shown within parenthesis in the binomial forest representation. When address 0101 comes in, it misses in all caches but the biggest. All the caches need to be checked in the forest simulation method, which takes four comparisons. However, in the binomial forest method (since there is only one binomial tree there is no need to locate the tree in the forest in this example) the right-match of the set field of the address with the set number of the root is determined. The right-match is 2, i.e., $k=2$ in the algorithm. The algorithm branches to the rank $S+M-k-1=0+3-2-1=0$ child of the root by swapping up the corresponding subtree. The right-match now is 3 and the tags match. The hit-array is incremented and the algorithm finishes with just two comparisons in this example.

[7] In this description the set is first located and the SWAPs are done at the end. In Algorithm BF_LS the SWAPs are done while searching. It is easily seen that both are equivalent.

Algorithm BF_LS

```
sim_bf_ls()
    Initialize()
    For every address x in trace
        set_no_C0 = Set number of address in C_S^L (x[S + L − 1 : L])
        set_no_CM = Set number of address in C_{S+M}^L (x[S + M + L − 1 : L])
        tag = Tag of address for C_{S+M}^L (x[W − 1 : S + M + L])
        cur_node = Root[set_no_C0]
        /* Search tree till set in C_{S+M} is located. Swapping is done with search */
        while ((k = Right_Match(cur_node→set_no, set_no_CM)) != S+M)
            child_node = Child(cur_node, S+M-k-1)
            SWAP(child_node)
            cur_node = child_node
        end while
        /* Update hit-array if tags match, else update tag */
        if (cur_node→tag == tag)
            D = Degree(cur_node)
            ++Hit_Array[M-D]
        else
            cur_node→tag = tag
        Root[set_no_C0] = cur_node
    end for
Number of hits in C_{S+k} is Σ_{i=0}^{k} Hit_Array[i]

Child(x, d)
    Return(Child of x rooted at tree of degree d)

Right_Match(a, b)
    Return(Number of bits starting from the least significant bit which match in a and b)

Initialize()
    Build 2^S binomial trees from 2^{S+M} sets
        (Using the combining procedure, assuming arbitrary ordering)
    Set all tags to invalid
```

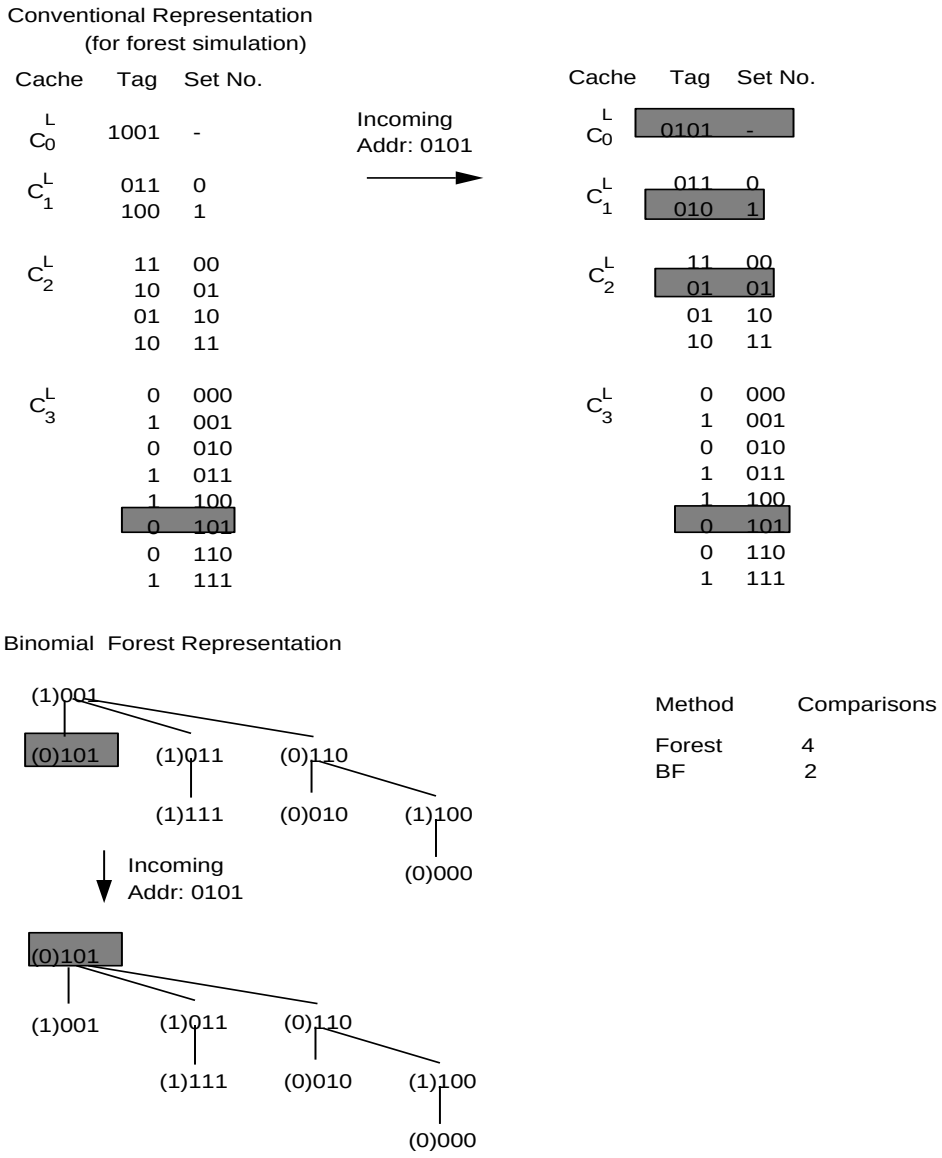


Figure 4: Example showing operation of Algorithm BF_LS

Implementation Issues

Implementations of binomial forests based on binary trees are given in [17] and [3]. But these implementations take up pointer space and incur high overheads for manipulation. Below we describe an array implementation of the binomial forest that is based on Property 1.

In the array implementation each binomial tree is represented as a one-dimensional array. The forest is a two-dimensional array. The array for a binomial tree of degree M consists of 2^M locations. The 2^M locations are divided into $M+1$ levels. Level 0 has one location. Level k ($k > 0$) has 2^{k-1} locations numbered 0 through $2^{k-1} - 1$.

Sets of rank $M - k$ in a binomial tree map to level k of the corresponding array. The least significant $S + k - 1$ bits of a set of rank $M - k$ determine the location it maps to in level k; the least significant S bits determine the binomial tree in the forest and the other $k - 1$ bits determine the location in level k of the corresponding array. From Property 1 it follows that two sets cannot map to the same location.

Since no links are maintained, some way of associating a set at level k with its child at level $k + r$ is necessary. From Property 1, the set and its child match in the least significant $S + k + r - 1$ bits. Since the location of a set in level $k + r$ is determined by its least significant $S + k + r - 1$ bits, the child can be located.

The operation SWAP changes the rank of only the root nodes of the subtrees that are swapped. In the array implementation therefore, SWAP is just an exchange of the contents of two locations in the array.

An array representation of the example in Fig. 4 is shown in Fig. 5. Levels are separated by double lines. The tag at a set is shown within parenthesis. The tag is not important in the present discussion. $M = 3$ in the example. Set 001 is at the root and is of rank 3. It maps to $3 - 3 = 0$ level of the array. Set 111 is of rank 0. It maps to level 3 and occupies location 11 in that level. Other sets are similarly mapped.

Since each comparison requires a right-match the efficiency of the right-match function is important for the performance of the algorithm. On machines such as the Cray, the first-one function[8] is performed in constant time by the hardware. The first-one function can also be done efficiently in constant time with a table lookup on machines that do not have

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

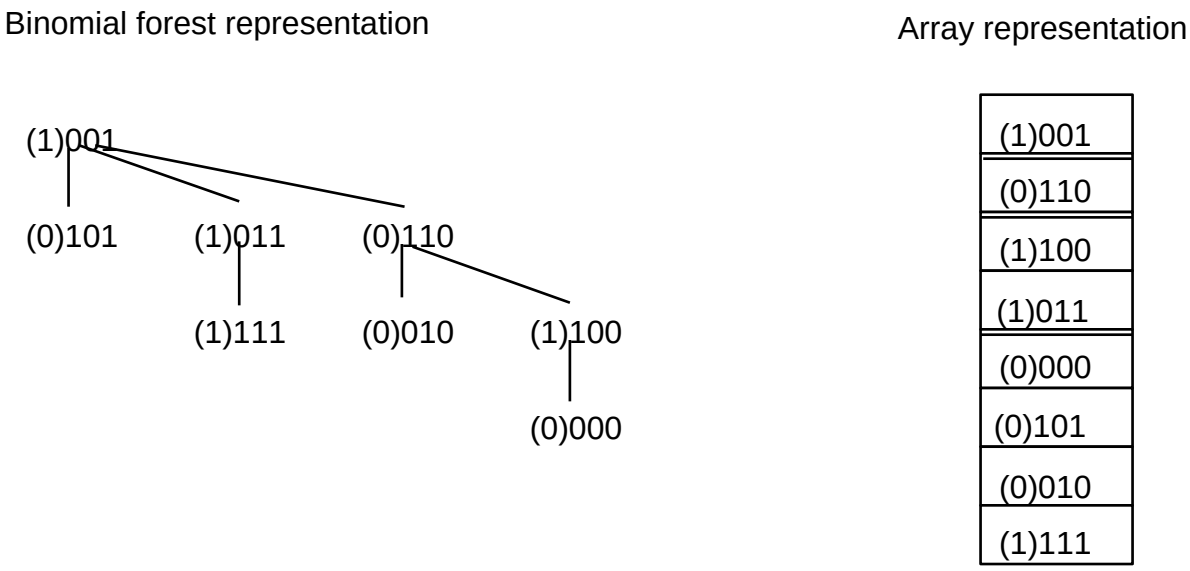


Figure 5: Mapping from binomial tree to array

this facility. For simulating a range of ten cache sizes (a factor of 1024 difference between the smallest and largest caches), the size of the table is only 1 Kwords.

Complexity

In the binomial forest method, the worst case number of comparisons and SWAPs is $O(M)$. In forest simulation too, the worst case complexity is $O(M)$. But on the average the binomial forest method takes fewer comparisons. The example given in Fig. 4 illustrates the reason. In the binomial forest method, a range of caches is skipped when there is a large right-match at a node along the search path. But there is no skipping in forest simulation. Each cache has to be checked till there is a hit. The binomial forest method makes at most as many comparisons as forest simulation to process an address. This is because in a binomial tree of degree M, a node at depth d (where the root of the tree is at depth 0) is at most of rank $M - d$. So the number of comparisons to locate a set of rank $M - d$ is less than or equal to $d + 1$ in the binomial forest method. But the smallest cache containing the line at a set of rank $M - d$ is C^L_{S+d}, and forest simulation takes at least $d + 1$ comparisons.

In [6] the expected number of comparisons required by forest simulation is given as

$$1 + m_0 + \sum_{i=1}^{M-1} m_i$$

where m_i is the miss ratio for cache C^L_{S+i}. Along the same lines the expected number of comparisons for the binomial forest method is:

$$1 + m_0 + \frac{1}{2} \sum_{i=1}^{M-1} m_i - m_M$$

assuming that if an address misses in cache C^L_{S+p}, and hits in C^L_{S+p+1} it is equally likely to hit any of the 2^p sets of C^L_{S+p+1} whose contents are not in C^L_{S+p}. Comparing the two expressions above we see that, at best the binomial forest method can outperform forest simulation by a factor approaching two. However, in the binomial forest method a right-match and an exchange are required with each comparison, as against the simple update in forest simulation. We present the results of empirical comparisons in Section 5.

There may be a space saving in the binomial forest method over forest simulation. Forest simulation maintains the tag store for all the caches whereas the binomial forest method maintains the tag store for just the biggest cache. Though the binomial forest method requires storage for the set number, the tag and set field can be combined into one word. If storing each tag in forest simulation requires one word as well, then the binomial forest method achieves about a factor of two saving. This factor may be important when large caches are being simulated on machines with relatively small amounts of physical memory, or on machines where good cache performance is crucial.

3.2 Constant CS - Varying LS

We consider the single-pass simulation of a range of direct mapped caches, C^{c-S-i}_{S+i}, $i = 0, \ldots, M$, with a fixed cache size, 2^c, and varying line size. Since the line size varies and the cache size is fixed, the number of sets also varies. One application of an algorithm for simulating this class of caches is in the design of first level caches. The size of first level direct mapped caches is usually constrained by the page size so that the cache access can be done in parallel with the TLB lookup. Once the page size is fixed, the design space consists of caches in this class.

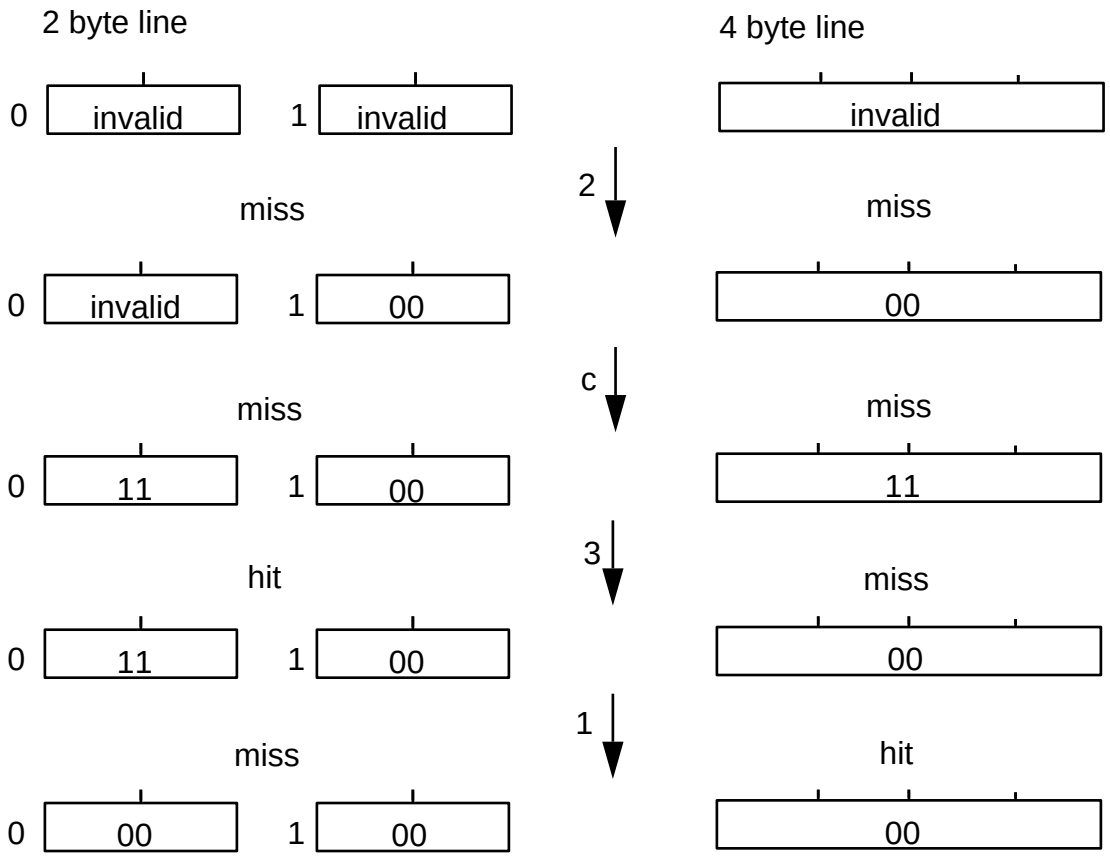


Figure 6: Inclusion does not hold for the constant CS - varying LS class

The presentation in this subsection is along the lines of the previous subsection. The data structure is first developed and the algorithm is then presented. Implementation and complexity issues are discussed. There is a strong similarity between the data structure and algorithms of this subsection and the ones of the previous subsection. The final part of this subsection explores the similarities and differences.

Cache Relations and Data Structure

Data inclusion does not hold between the caches in this class. This can be seen from the example in Fig. 6. The figure shows two caches C_1^1 and C_0^2, and their action under a

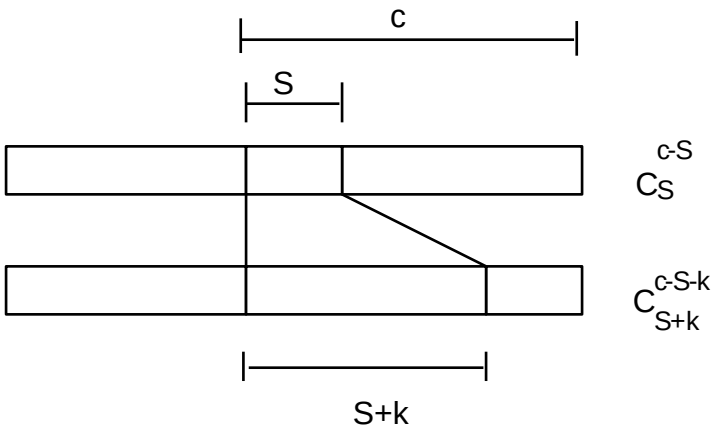


Figure 7: Bit fields in C_S^{c-S} and C_{S+k}^{c-S-k}

trace. The third address in the trace misses in C_0^2 only, while the fourth address misses in C_1^1 only, illustrating that data inclusion does not hold either way. The cache with the larger line size brings in more data on a miss, but the cache with the smaller line size has greater flexibility in what it may keep.

An important uniformity between caches in this class is that all of them have the same tag field width. This permits us to prove an inclusion property between the tag store contents of caches in this class. Clearly tag inclusion is not the same as data inclusion, since the data locations a tag refers to depend on the set number and the line size. But tag inclusion may be exploited in a different way to obtain an efficient simulation algorithm. The following lemma is similar to Lemma 1 but tag inclusion is proved rather than data inclusion.

Lemma 2: 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 $[p]_{tag} \subset [P]_{tag}$

Proof :

Since the cache size, 2^c, is constant, the set field in C_S^{c-S} is a part of the set field in C_{S+k}^{c-S-k} and is smaller in width by k bits Fig.7. So addresses mapping to a single set, p, in C_S^{c-S} map to one of the 2^k sets $P = \{s \text{ s.t } s[S+k-1:k] = p\}$ in C_{S+k}^{c-S-k}. The tag in p has to be the same as the tag of the last updated set in P. $\square$

The following is an obvious corollary.

Corollary

The most significant S bits of the set number of any set in P gives the set number of p.

(Note that unlike in Lemma 1, $([P]_{tag} - [p]_{tag}) \cap [C_S^L]_{tag} \neq \phi$, because the same tag may be found in more than one set unlike lines in the previous subsection.)

A binomial tree structure is constructed as in the previous subsection. Consider the sets in C_{S+2}^{c-S-2}. Pairs of sets are combined together with the more recently updated set of a pair at the top. The tags in the sets in the top layer are the same as those in the tag store of C_{S+1}^{c-S-1} by Lemma 3. The set fields of sets in a pair differ only in the LSB (for the constant LS class it was the MSB) and the set number in C_{S+1}^{c-S-1} is obtained by stripping off the LSB. Set pairs are themselves combined together and the top layer then gives the tag store of C_S^{c-S}. Tag stores of smaller caches can be obtained by combining further. A binomial forest results. More formally: To simulate caches C_{S+i}^{c-S-i}, $i = 0, \ldots, M$, a binomial forest representing the caches is obtained as follows:

Combining Procedure

Input: Sets in cache C_{S+M}^L and order in which they were updated.

Procedure:

1. Form a forest of trees of degree one by connecting pairs of sets s_1 and s_2 with an edge, with the more recently updated set as the root, if $s_1[S+M-1:1] = s_2[S+M-1:1]$.
2. For $i = 2$ to M,

Form a forest of trees of degree i by connecting the roots of pairs of trees t_1 and t_2 of degree $i-1$ with an edge, with the more recently updated root as the new root, if $s_1[S+M-1:i] = s_2[S+M-1:i]$, where s_1 is any set in t_1 and s_2 is any set in t_2.

Output: Forest consisting of 2^S trees of degree M.

(This is similar to the combining procedure of the previous section but with the least significant bits progressively dropped in determining pairs, instead of the most significant bits.)

The following two properties are derived from the construction procedure. In the following, set number refers to the set number in C_{S+M}^{c-S-M} unless stated otherwise.

Property 1 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 2 The tag in a set of rank k is in the tag stores of caches C_{S+i}^{c-S-i}, $i = M-k, \ldots, M$. 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.

Property 1 is similar to Property 1 of the previous section but with most significant replacing least significant. Property 2 is similar to that of the previous section but with tag replacing line. The same tag represents different data in the various caches. For instance, a tag at the root, represents a line of size 2^{c-S} in C_S^{c-S} but a line of size 2^{c-S-M} in C_{S+M}^{c-S-M}. In Property 2 a set number is associated with the tag in each of the caches, to uniquely identify the data represented by the tag in each cache.

Fig. 8 illustrates the binomial tree representation of caches in this class. Here $S = 0$ and $M = 3$. Both the binomial tree and conventional representations are shown. In the binomial tree representation, the tag (within parenthesis) 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 line of bytes 1000-1111; 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 operation SWAP is defined exactly as it was defined in the previous subsection. SWAP at a node of degree k exchanges the tree rooted at the node with the subtree of degree k rooted at the parent.

Algorithm and Implementation Issues

Algorithm BF_CS uses the binomial tree structure to simulate caches in this class. At each node of the binomial forest the corresponding set number and the tag at the set in C_{S+M}^{c-S-M} are stored. When an address x comes in, the algorithm uses the field $x[c-1:c-S]$ of the address to locate the binomial tree which could contain it. The tree is then searched for the set s such that $s = x[c-1:c-S-M]$. At any point in the search procedure, if the left-match[9] between the set number at the node being examined and $x[c-1:c-S-M]$

[9] *Left-match* between two set fields s_1 and s_2 of width w is defined as the minimum i such that, $s_1[w-1:w-1-i] \neq s_2[w-1:w-1-i]$, where w is the width of the set field.

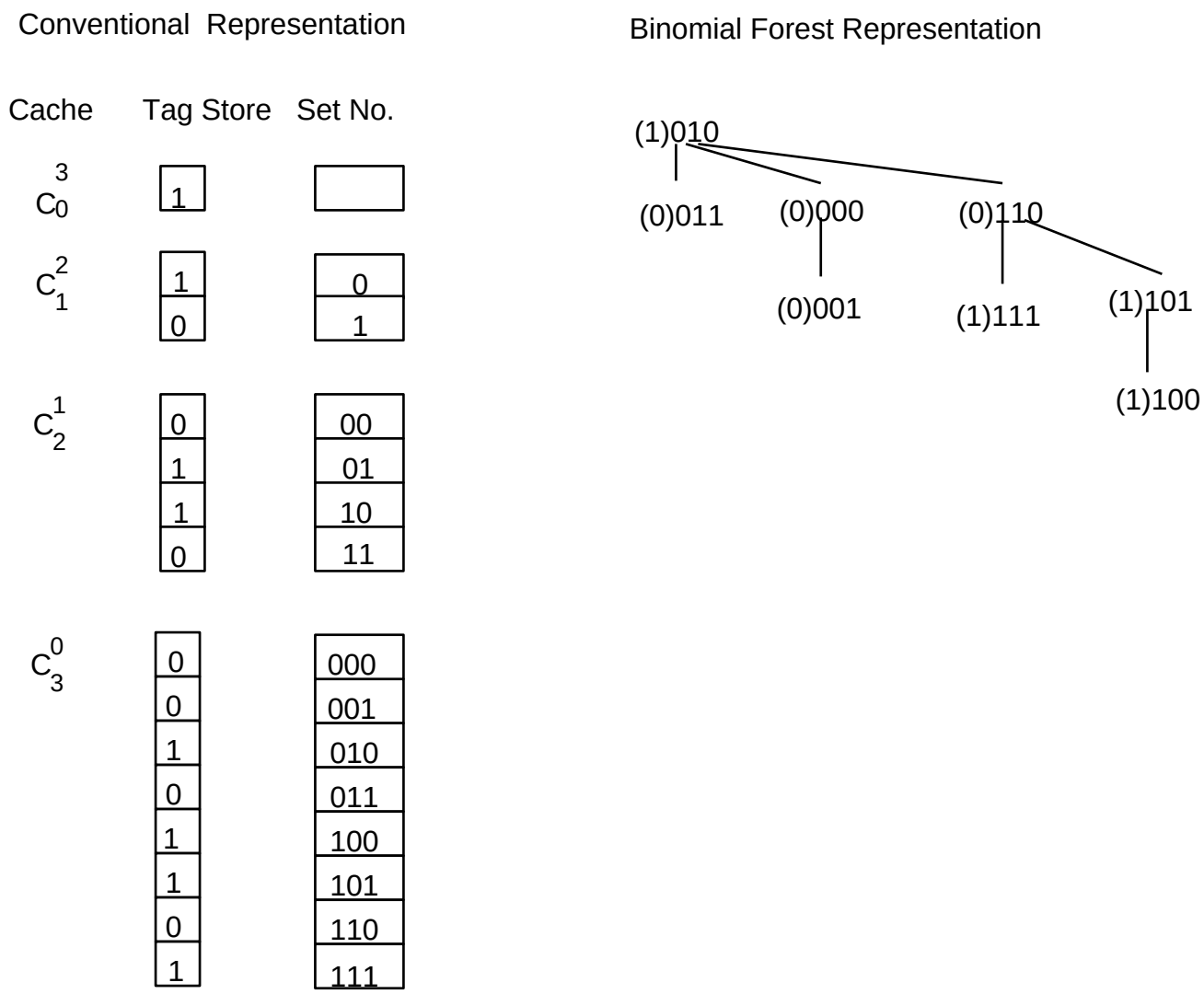


Figure 8: Example showing binomial tree representation of constant CS – varying LS caches.

is k, the child to be searched next is the child of rank $S + M - k - 1$. The reason for this is similar to the reason given in the explanation of Algorithm BF_LS. When the set with the complete match is located, its tag is compared with the tag of the address. If the tags match, then the address hits in caches as given in Property 2. The set with the complete match is swapped to the top, as in Algorithm BF_LS.

The non-uniqueness 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 2). 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 $C_{S+i}^{c-S-i}, i = M-k, \ldots, t$.

Fig. 9 shows the algorithm working on the example of Fig. 8. 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_0^3. 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_1^2 or C_2^1. The rank 0 child is swapped with the root. Now the set and tag field of the root and the address match. The address hits in C_3^0.

On each tag match the address hits in a range of caches. This necessitates updating many caches on a tag match. This update process is time consuming. The two dimensional hit-array is used to circumvent this inefficiency. On each tag match a location in the two-dimensional array corresponding to the range of caches hit is updated. This decreases the number of increments required on a tag match to one.

The binomial trees can be efficiently implemented as an array with a mapping similar to the one described previously but with the most significant $S + k - 1$ bits determining the position of a rank $M - k$ set in level k. SWAP can be done in constant time if left-match is a constant time function. This is true if left-match is implemented as a table lookup.

Algorithm BF_CS

```
sim_bf_cs()
    Initialize()
    For every address 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→set_no, set_no_CM)
            if (cur_node→tag == tag)
                ++Hit_Array[M-k][t]    /* Hit in caches C_{S+M-k}^{c-(S+M-k)} to C_{S+t}^{c-(S+t)} */
            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 while
        Root[set_no_level0] = cur_node
    end for
Hits in C_{S+k}^{c-S-k} = Σ_{i=0}^{k} Σ_{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(Number of bits starting from the most significant that match in a and b)

Initialize()
    Build 2^S binomial trees from 2^{S+M} sets
        (Using the combining procedure, assuming arbitrary ordering)
    Set all tags to invalid
```

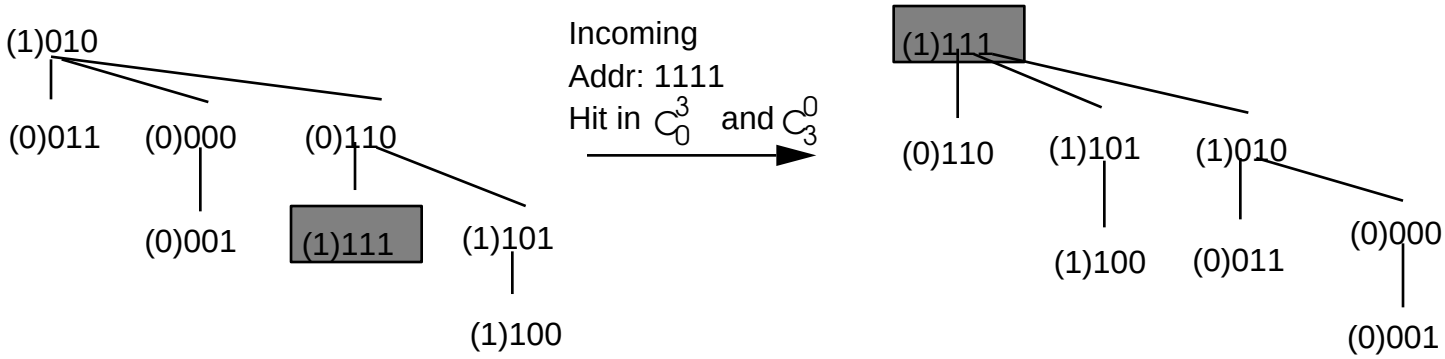


Figure 9: Example showing operation of Algorithm GBT_LS.

Complexity

The worst case number of comparisons and SWAPs, per address, of the binomial tree algorithm described above is $O(M+1)$. The performance of the algorithm on actual traces is likely to be better. Unlike the previous case, average-case analysis is difficult because there is no direct correlation between miss ratio and the number of comparisons required. The following approximate analysis considers two simple trace models — a random model (poor locality) and a simple looping model (good spatial and temporal locality) — and derives average case number of comparisons for the two. The actual performance is likely to be between the extremes.

For a random trace, the set searched for is equally likely to be in any position in the tree. In a binomial tree of degree M, the number of sets at depth 0 is $\left(\begin{array}{c} M \\ 0 \end{array} \right)$, at depth 1, $\left(\begin{array}{c} M \\ 1 \end{array} \right)$ and in general at depth d, $\left(\begin{array}{c} M \\ d \end{array} \right)$ (this binomial distribution is the reason for the name). So the expected number of comparisons when $M+1$ caches are simulated (trees of degree M) is $(M+1)/2$.

Next assume that the trace is a list of successive addresses repeated endlessly. For instance, such traces are generated when a loop is executing repeatedly. Assume that the entire stream maps to a single binomial tree. If they map to two or more trees, we can just consider the part which maps to one tree. Further, assume that each trace address requests a word and that the cache with the smallest line size has one word. The average number of comparisons, ignoring the initial misses, as a function of the loop size is shown in the

Size of Loop	1	2	3	4	5	6	7	8	k
No. of comps.	1	2	2.33	2.5	2.6	2.67	2.71	2.75	$2\ \frac{k-2}{k}$

Table 1: Variation in average comparisons with loop size.

Table 1. The average number of comparisons approaches 3 as the loop size increases. When the smallest line size is greater than one word, the average number of comparisons is less than two.

Practical traces consist of interleaved runs of successive addresses [1]. All misses and hits occurring on references to starting addresses of runs may be modeled as random accesses. The loop model may be used for hits that occur on references to second and later references in runs. The number of comparisons would be somewhere between three, two or less (depending on the line size of the smallest cache) and $(M + 1)/2$, where $M + 1$ is the number of caches simulated. For instruction traces or data traces of matrix oriented programs where many accesses are sequential, the number of comparisons would be closer to two or three. For irregular traces, the comparisons would be closer to $(M + 1)/2$.

We have not come across other algorithms for simulating this class of caches in the literature. In the naive algorithm every address would require $M + 1$ comparisons and updates. The cost of a comparison is greater in the binomial forest method but is much less than the gain from the decrease in number of comparisons. Results of empirical comparisons are presented in Section 5.

Constant LS and Constant CS Classes

The data structures used in the constant CS and constant LS algorithms are identical. The search procedure is also similar except for the left/right-match difference. We note the following transformation of addresses to go from the constant CS class to the constant LS class.

For each address x
1. Remove the field $x[M + L - 1 : L]$ from the address and reverse it, i.e., the MSB of the field becomes the LSB and vice versa.
2. Slide the field $x[S + M + L - 1 : M + L]$ to the right by M bits into the gap created by the above removal.
3. Insert the reversed field in the M vacant bits created by the above slide.

Caches C_{S+i}^{c-S-i} may be simulated using Algorithm BT_LS and a trace of transformed addresses except for the update of the hit-arrays. To update the hit-arrays correctly tag matches have to be done along the search path as described in Algorithm BF_CS . This difference arises because we are interested in data hits. Tags which are the basic unit in the constant CS algorithm and lines which are the basic unit in the constant LS algorithm relate differently to data. As the boundary moves in the constant CS case the amount of data represented in a set grows, but this does not happen in the constant LS case.

The relation between the two algorithms can be carried over to the domain of localities. Two kinds of locality are commonly identified — temporal locality and spatial locality. The constant LS algorithm exploits the fact that the temporal and spatial locality of a program are largely exploited by smaller caches, by keeping the data contained in smaller caches close to the root of the binomial trees. The constant CS algorithm exploits spatial locality for good performance. It tends to bring neighboring data items closer to the top of the binomial tree so that fewer comparisons are performed if the program refers to these data items.

For a cache of a given size increasing the line size decreases the relative ratio of temporal and spatial locality hits. The constant cache size class is useful for investigating this temporal locality - spatial locality trade-off. The constant line size class is useful for investigating the miss ratio - cost trade-off.

4 Set Associative Caches

The parameters of a set associative cache include the line size (LS), the number of sets (NS), the cache size (CS) and, the degree of associativity (DA) with the relation $CS = LS \times NS \times DA$. Single-pass simulation of set associative caches of a range of NS and DA but fixed LS is considered in this section. CS varies with NS and DA.

This section is structured along the lines of Subsection 3.1. It consists of three parts. In the first part the data structure and algorithm are presented. In the next part implementation issues are discussed and the last part compares the complexity of the new algorithm to earlier algorithms.

Cache Relations and Data Structure

A lemma relating the contents of two caches in this class of set associative caches is proved below. A previously known data inclusion property follows as a corollary to the lemma. Earlier algorithms which are based on this inclusion property are reviewed. The new data structure for representing this class of set associative caches is then introduced. This data structure is a generalization of the binomial tree. The authors are not aware of earlier references to this data structure in literature. Some of the properties of this data structure are explained. The new algorithm is then presented.

Lemma 3: For each set p in $C_S^L(n)$ there are exactly 2^k sets, P, in $C_{S+k}^L(n)$ such that $[p]_{lines} \subset [P]_{lines}$ and $([P]_{lines} - [p]_{lines}) \cap [C_S^L(n)]_{lines} = \phi$.

Proof :

Compared to $C_S^L(n)$, k extra bits are used for selecting a set in $C_{S+k}^L(n)$. 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 of 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 1

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

Corollary 2

$[C_{S_1}^L(n)]_{lines} \subseteq [C_{S_2}^L(n)]_{lines}$, if $S_1 \leq S_2$.

Corollary 1 is the sufficient part of Theorem 1 in [7] except that bit selection is assumed here. It states an inclusion property between caches in this class. This inclusion property can be used to develop an algorithm similar to forest simulation [7], referred to as *generalized forest simulation* (GFS) in the following. In GFS a separate two-dimensional array is

maintained for each cache. The set number varies along one dimension and for each set number there is an array of entries. For each incoming line, the appropriate set in each cache is searched, starting at the cache with the least number of sets. If it hits at depth one in any cache, then subsequent caches need not be searched. We call this *generalized forest simulation* (GFS) in the discussion that follows.

All-associativity simulation [9, 7] is another algorithm for the simulation of this class of set-associative caches. It works as follows: Consider 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 appropriate LRU stack is searched. Lines that map to the same set in $C^L_{S_2}(n)$ also map to the same set in $C^L_{S_1}(n)$, when bit-selection is used. [10] Therefore the lines examined while searching for a specific line, a, in the LRU stack of $C^L_{S_1}(n)$ is a superset of the lines that would be examined while searching for a in the LRU stack of $C^L_{S_2}(n)$. While searching $C^L_{S_1}(n)$, a count is maintained of the lines that map to the same set as a in $C^L_{S_2}(n)$ too. [11] When the address is located therefore, the position of the address in the LRU stack of both $C^L_{S_1}(n)$ and $C^L_{S_2}(n)$ is known. The algorithm can be extended to simulate more than two caches.

Both GFS and all-associativity simulation may examine lines that the other does not. This is best illustrated with an example. Consider three caches of degree of associativity two, with one, two and four sets as shown in Fig. 10. Their tag contents are also shown. If the incoming address is 0101, GFS requires five comparisons, whereas all-associativity requires just two. This happens because GFS reexamines lines that it has seen in earlier caches. But if the incoming address is 0011, all-associativity requires eight comparisons, whereas GFS requires only six comparisons. This happens because all-associativity does not limit the size of the LRU stack. The complete LRU stack for $C^L_0(2)$ contains lines that are neither in $C^L_0(2)$ nor in the relevant sets of $C^L_1(2)$ or $C^L_2(2)$. In the rest of this section we develop the new algorithm which always does as well as or better than both GFS and all-associativity simulation.

Lemma 3 can be used to develop a data structure for the class of set associative caches under consideration using an approach similar to the one in the previous section. Consider

[10] This property is called set refinement in [7].

[11] Determination of whether a line in $C^L_{S_1}(n)$ maps to the same set as a in $C^L_{S_2}(n)$ is done using the right-match function.

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

Conventional representation
(for generalized forest simulation)

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 all-associativity)

1011
0101
1000
0110
0001
0100
1010
0011

Incoming Address	Method	No. of Comparisons
0101	GFS	5
	All-associativity	2
0011	GFS	6
	All-associativity	8

Figure 10: Example — Difference in required number of comparisons.

the lines in a cache $C^L_{S+2}(n)$. Lines in pairs of sets in $C^L_{S+2}(n)$ mapping to the same set in $C^L_{S+1}(n)$ are combined by forming a list of the n most recently referenced lines in either cache at the root. This is called the *root-list*. The remaining n lines are in two separate branches, each branch containing lines from distinct sets in $C^L_{S+2}(n)$. The n lines in the root-list are in $C^L_{S+1}(n)$. The other lines are only in $C^L_{S+2}(n)$. The structure formed can be further combined with another similar structure. The lines in the root-list of the resulting structure are in $C^L_S(n)$. Further combining may be done similarly. We call the structure formed by combining lists of length n recursively as described above a Generalized Binomial Tree of order n, GBT(n). An ordinary binomial tree is a GBT(1).

GBTs are defined and explained in the Appendix. The terms *list-child*, *tree-child*, *subtree*, *rank* and *degree* are defined in the Appendix. Some properties of GBTs are also proved in the Appendix. These properties are useful in justifying the algorithm presented below. We repeat the definitions for list-child and tree-child here since these are new terms which are used often in the following. 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.

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 till there are 2^S distinct GBT(n)s. The construction procedure can be written along the lines of Section 3.1.

Combining Procedure

Input: Lines in cache $C^L_{S+M}(n)$ and the order in which they were updated. Lines in the same set in $C^L_{S+M}(n)$ are in a list LRU order.

Procedure:

1. Combine the list of lines in Set s_1 with the list of lines in Set s_2, if $s_1[S+M-1:0] = s_2[S+M-1:0]$, so that the n most recently referenced lines are in the root-list in LRU order.

2. For $i = 2$ to M,

Let s_1 and s_2 be the set fields of the lines at the top of their trees. Combine two trees, if $s_1[S+M-1:i] = s_2[S+M-1:i]$, so that the n most recently referenced lines are in the root-list of the resulting tree in LRU order.

Output: Forest of 2^S GBTs of degree M.

The generalized binomial forest obtained using the combining procedure has the following properties. In the following, set number and tag are with respect to $C^L_{S+M}(n)$ unless stated otherwise.

Property 1 In a tree or subtree of degree k, the least significant $S+M-k$ bits of the set numbers are identical, and two subtrees of degree k differ in at least one of the least significant $S+M-k$ bits of the set numbers.

Property 2 A line of rank k is in caches C^L_{S+i}, $i = M-k, \ldots, M$.

Property 3 The path from the root to any line a contains exactly those lines that are in sets that a maps to in any of the caches considered and are referenced after the previous reference to a.

The validity of the three properties 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 5) and are present in cache $C^L_{S+M-k}(n)$. By the inclusion property the lines in the root-list are in smaller caches too, i.e., caches $C^L_{S+i}(n)$, $i = M-k+1, \ldots, M$. This is Property 2. 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 same and between trees they have to be different. This is Property 1. All the lines in a GBT are at some point in the combining procedure part of the root-list. So all the ancestors of any line in a GBT are in a set the line maps to in one of the caches considered and are referenced after the previous reference to line. This is Property 3.

Two operations are defined on this data structure. SWAP is similar to the SWAP defined for a binomial tree but here SWAP(v) is permitted only if v is a tree-child of its parent. If v is a node of rank k then SWAP(v) swaps the tree of degree k rooted at v with the subtree of degree k rooted at its parent. SWAP(v) is always possible when v is a tree-child, since the parent of v has a subtree of degree k, not including v (by the definition of subtrees). The other operation is EXCHANGE. This reflects the fact that in set associative caches not only do sets move ahead of other sets, but also lines in the same set move ahead of each other. EXCHANGE(v) is permitted only if v is the list-child of its parent. EXCHANGE(v) exchanges v with its parent node. The children of the child become the children of the parent and vice versa. It is seen that SWAP and EXCHANGE preserve Property 1.

Algorithm

Algorithm GBF_LS (Generalized Binomial Forest – fixed Line Size) simulates the class of set associative caches with a fixed LS and varying NS and DA. 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. The search procedure is similar in principle to the search procedure of Algorithm GBF_LS. On a right-match of k at a node, the tree-child of rank $S+M-k-1$ is searched next. But in a GBT, a node of rank greater than or equal to $S+M-k$ may not have a tree-child of of rank $S+M-k-1$ (Lemma 8); in such a case the list-child is of rank greater than or equal to $S+M-k-1$ and is searched next. When there is a complete set match and tag match at a node, the line is found and the search is successful. When there is a complete set match at a node and the node does not have a list child the search has failed (If there is another node with a complete set match, it is a list-child of the current node at the start of the combining procedure. Since a node loses a list-child only when it gets another one, and current node does not have a list child, it follows that there is no node later in the tree with a complete set match.). The line at the node examined last is replaced with the incoming line. The incoming line is now the most recently referenced and it is moved up to the root of the tree by a series of SWAPs and EXCHANGEs.

Property 2 lets us infer which caches of associativity n are hit. But using Property 3 and right-matches done along the search path we can determine hits in all caches in the class, i.e., $C^L_{S+i}(j), i=0,\ldots,M; j=1,\ldots,n$. The number of right-matches of $S+i$ or greater is determined for $i=0,\ldots,M$ using the right-match counts formed during the search. When t right-matches of $S+i$ are seen, the address hits in $C^L_{S+i}(j), j=t+1,\ldots,n$. This is because all lines referenced later and in the same set in some cache are examined in the search path (Property 3). These lines are the ones that have pushed the line out.

In Fig. 11, the examples of Fig 10 are shown processed by Algorithm GBF_LS.

1. For the incoming address 0101, the first line checked is 1011. The set field of the address does not match the set field of this line; the search moves to the only child 0101. Here the set field and the tag field match. Right-match counts are used to determine the caches the address hits. 0101 is moved to the top of the tree, in this case just an EXCHANGE.

2. For the incoming address 0011, the search goes to 0101 as before. The set field does

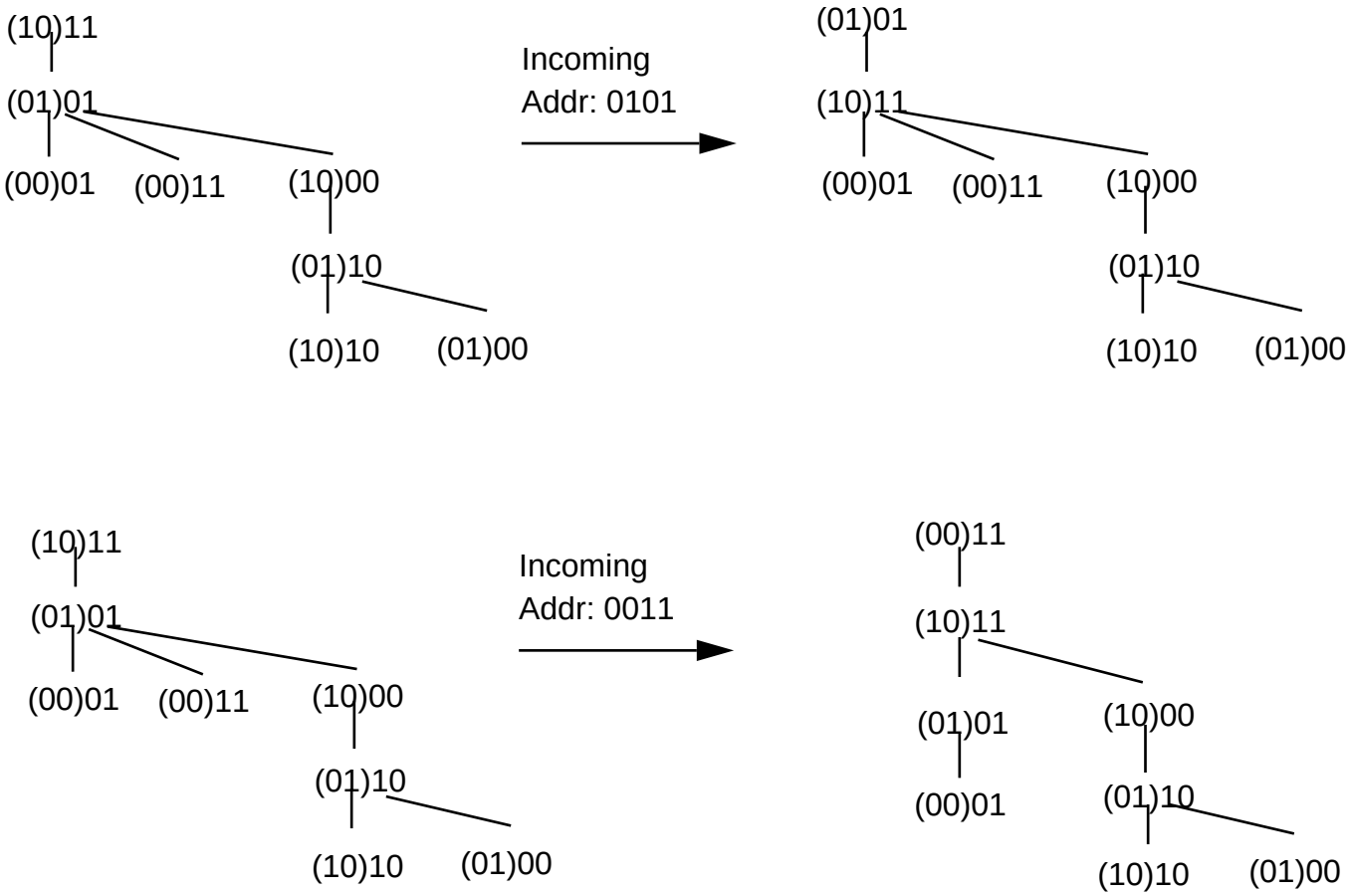
Algorithm GBF_LS

```
sim_gbt_ls()
    Initialize()
    For every address 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 = end_of_tree = 0
        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 += RM_Count[i]
                    ++Hit_Array[i][sum]
                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] += 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:
Hits in cache C^L_{S+k}(m) = Σ_{i=0}^{k} Hit_Array[m][i]

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 GBTs from 2^{S+M} line lists
        (Using the combining procedure, assuming arbitrary ordering)
    Set all tags to invalid
```



Incoming Address	No. of Comparisons
0101	2
0011	3

Figure 11: Example illustrating Algorithm GBF_LS

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, it is the next node to be searched and is swapped with the subtree {0101, 0001}. Both the set and tag fields match at 0011. The search is successful. Right-match counts are used to determine the caches the address hits and 0011 is EXCHANGEd upto the top.

For 0101 the search took two comparisons and for 0011 the search took three comparisons. These are less than or the same as the number of comparisons for either of the algorithms shown in Fig 10. We show later that this is true in general.

Implementation Issues

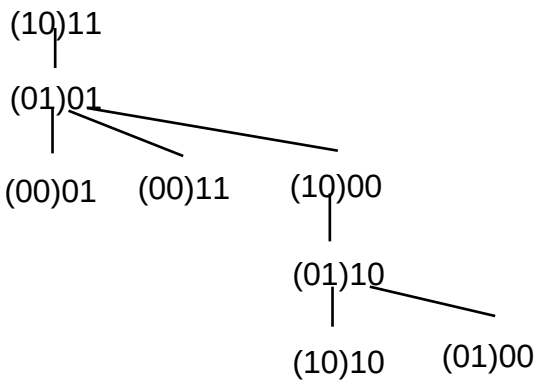
Using Property 1 a forest of GBTs is implemented as a three-dimensional array; each GBT is a two-dimensional array. A GBT(n) of degree M is represented by a two-dimensional array with $2^{M+1} - 1$ rows and n columns. The rows are divided into $M + 1$ levels. Level 0 has one row. Level k has 2^k rows, numbered 0 through $2^k - 1$. A GBT of degree M has $n2^M$ nodes (Lemma 4. Since the array structure allots $n(2^{M+1} - 1)$ locations, there is approximately a factor of two redundancy in the array implementation.

A line of rank k maps to level $M - k$. The row in the level is determined by the least significant $S + k$ bits of the line's set number; the least significant S bits determine the GBT in the forest and the remaining k bits determine the row number in level k of the corresponding array. At most n lines can map to the same row. When more than one line maps to a row, the lines are arranged in the order in which they were updated.

A link is required from a node to its children in the array representation. This is not straightforward because in a GBT a node's children depends on its ancestors too. The right-match counts of the ancestors are required to identify the children of a node. Here we do not describe this mapping. Instead we just give a simpler search procedure.

In the search procedure one row in each level is checked. Right-matches and tag matches are done with the lines in that row. If the incoming address is x, the row examined in level k is the row numbered $x[S + L + k - 1 : S + L]$. We state without proof that the lines examined by the simple search procedure would be the same as the lines examined by Algorithm GBF_LS searching the GBT.

The above array implementation of GBTs is similar in principle to the array implement-



(10)11	(01)01
(10)00	(01)10
(01)00	
(00)01	
(10)10	
(00)11	

Figure 12: Example: Mapping from GBT to array

ation of binomial trees presented in Subsection 3.1. The variability of the GBT structure introduces a difference. This is handled by introducing a factor of two redundancy in the array for GBTs.

Fig. 12 shows an example of the mapping, where $M = 2$ and $n = 2$. 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 appropriate row in level 1. The degree 0 lines map to the appropriate rows in level 2. To search for line 0011, 1011 and 0101 are examined first in level 0. In level 1 there is no entry in row 1, which 0011 (the incoming address) maps to. In level 2, 0011 maps to row 11 and 0011 is found there. This search takes three comparisons which is the same as earlier.

SWAP and EXCHANGE just change the ranks of the two sets directly involved; so they just involve two moves: one line moves to the higher level while the line in the higher level comes down to the lower level.

The RM_count array (of Algorithm GBF_LS) has to be processed at the end of each successful search. The following microoptimization saves some of this processing time: Since *RM_count* has $M + 1$ locations and the maximum count at any location is n, it is possible to assign integer values to the various possible value sets of *RM_count*. An array may be maintained, and at the end of a successful search the array location corresponding to the value set of *RM_count* may be incremented. This array is processed after all the addresses in the trace have been processed. The processing of the *RM_count* array which was done at the end of each successful search is now done once for each array location. The counts of

many of the array locations is greater than one and hence the saving. The size of the array is exponential in the range of set sizes simulated. So if this micro-optimization is turned on, the range of set sizes that may be simulated is limited.

Complexity

For the simulation of caches $C^L_{S+i}(n), i = 0, \ldots, M$, $O((M+1)n)$ worst case comparisons are required in the GBT method, per address. This is compared to the worst case $O(2^M n)$ comparisons necessary in all-associativity simulation. The worst case complexity of GFS is $O((M+1)n)$. But GFS does not make good use of the locality of program traces. To illustrate this, suppose the incoming line is in the $k^{th}, k < n$ position in the $C^L_S(n)$ stack and suppose it does not occupy the most recently used position in any of the other caches. In GFS all the caches need to be checked and updated. In all-associativity simulation and in the GBT algorithm however, only k positions have to be checked. A large percentage of the addresses in program traces hit in $C^L_S(n)$. GFS performs rather poorly on practical traces.

The GBT algorithm never makes more comparisons than either the simple method or all-associativity simulation. This can be justified as follows: In the GBT algorithm, lines that are *in* sets that the incoming address x maps to, and referenced after the earlier reference to x are examined once. In all-associativity simulation 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 the GBT algorithm examines are clearly a subset of the lines that all-associativity simulation examines. The GFS algorithm examines the same set of lines as the GBT algorithm but it examines some lines more than once. To bound the performance improvement, the GBT algorithm can make $M+1$ times fewer comparisons than the GFS algorithm and $2^M/(M+1)$ times fewer (this is assuming no redundant nodes in the stack) comparisons than all-associativity simulation.

Memory is handled poorly in all-associativity simulation. The LRU stack of a set contains all distinct addresses that map to the set, even if some addresses are no longer present in any of the caches. So the size of the stacks cannot be bounded. This has two bad effects. One is the large amount of unnecessary memory required, especially in big simulations. The other is that the unnecessary nodes add to the number of comparisons that have to be made. A periodic clean-up is required to overcome these problems.

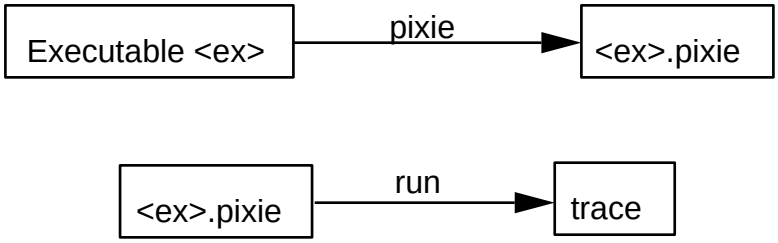


Figure 13: Address tracing with pixie

5 Empirical Performance Comparisons

In this section, results of empirical comparisons of implementations of our algorithms described above, and algorithms previously published in literature are presented. In the first subsection details of the traces and algorithm implementations is presented. In the subsequent subsection the execution times of the various implementations on the traces is given.

5.1 Traces

The traces were obtained using the *pixie* utility provided on machines such as the *DEC3100* which use the *MIPS R2000* family of microprocessors. A schematic showing how pixie works is given in Fig. 13. Pixie is run on the executable of the program to be traced. An instrumented executable with a *.pixie* suffix is created. When this executable is run an address trace is generated, which is piped to the cache simulator program. The programs traced are from the SPEC benchmark suite. This is a a collection of numeric and non-numeric programs widely used for benchmarking microprocessor-based workstations. A mix of benchmarks were chosen including scientific code *spice2g6*, utilities *gcc1.35* and integer benchmarks *espresso* and *intmc*. The trace is a combined instruction and data trace. Reads and writes are handled identically. While this approach may not be appropriate for evaluating the cache performance for some types of machines, it is less likely that this affects the relative performance of the simulation algorithms.

All the algorithms were implemented in C. The binomial tree algorithms were implemented using arrays. Programs for forest simulation, GFS, and all-associativity simulation were also implemented by the authors. Software packages like DineroIII and Tycho which implement forest simulation and all-associativity simulation were not used since there is

Workload	No. of Comps.[12]		Exec. Time	
	Forest	BT	Forest	BT
gcc	1.5791	1.2087	257.2	241.7
espresso	1.2155	1.0969	190.0	196.1
spice2g6	1.7033	1.4492	242.5	243.6
intmc	1.1362	1.0834	192.2	183.8
mean	1.4085	1.2096	220.48	216.3

Table 2: Performance of Constant Line Size Direct Mapped Cache algorithms. Caches $C^4_{7+i}, i = 0, \ldots 7$.

some overhead included in such packages and a fair comparison would be difficult. We present numbers from Tycho for comparison. In all cases the first 100 million references were simulated.

5.2 Experimental Results

Table 2 gives the number of comparisons and simulation times for direct mapped caches. It is seen that the execution times of forest simulation and the new binomial tree algorithm are close. This is primarily because of the high percentage of hits that occur at the top level cache. The simulation time is dominated by the comparisons at the top level cache. The binomial tree method makes fewer comparisons but the saving in comparisons is often not enough to offset the overhead of obtaining right-matches and exchanging tags instead of just replacing tags as in forest simulation.

Table 3 gives the number of comparisons per address and the simulation times for the constant cache size binomial forest algorithm and the naive algorithm. As noted earlier the naive algorithm examines all the caches for each address. The new algorithm consistently outperforms the naive algorithm by a factor of about two. It is seen that the number of comparisons per trace address for the binomial forest algorithm is around two. This indicates good spatial locality in the trace, primarily due to sequential accesses within basic blocks. For the naive algorithm, of course, the number of comparisons is constant at six.

The set associative cache simulation times for two different ranges of caches are given

[12] *No. of Comps.* denotes the mean number of comparisons per trace address in this and subsequent tables.

Workload	No. of Comps.		Exec. Time	
	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

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

in Table 4 and Table 5. The degree of associativity is eight in the former and four in the latter. The number of sets is also different. From the tables we see that associativity does not change the relative performance of the algorithms much for this set of traces. The number of comparisons per trace address is also given for Table 5. The all-associativity simulation times are without cleaning. Runs done with cleaning enabled show that, if the memory limit is set appropriately, the time with cleaning is occasionally better than the time without. This is because cleaning removes unnecessary nodes from the stacks and the number of comparisons that need to be done is less. But the difference is not significant.

The algorithm described in this paper outperforms all-associativity simulation by close to a factor of five for the *gcc* trace. *Gcc* references many distinct addresses and the resulting big stack is the cause for the poor performance of all-associativity. On other traces the performance difference is a factor of 1.5-2. On the average GFS performs better than all-associativity simulation. The GBT algorithm outperforms GFS by about a factor of 1.4 on the average. We also did simulation runs using Tycho, which is a software package for doing all-associativity simulation [5]. As noted before, there is overhead associated with such packages. As an example, for simulating the first 100 million references of *spice2g6*, for the caches in Table 5, Tycho took 6993.0*secs* of user time, on a DEC3100.

6 Conclusion

The increasing difference in the speed of the CPU and memory have made cache memories more important. This has resulted in larger caches and greater interest in cache design.

Workload	GFS	All Ass.	GBT
gcc	810.5	2355.6	565.7
espresso	429.1	455.0	296.9
spice2g6	523.4	885.1	453.1
intmc	344.9	342.0	226.9
mean	527.0	1009.4	385.7

Table 4: Performance of Constant Line Size Set Associative Cache algorithms. Caches $C_i^3(j), i = 7, \ldots, 11, j = 1, \ldots, 8$.

	No. of Comps.			Exec. Time		
Workload	GFS	All Ass.	GBT	GFS	All Ass.	GBT
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

Table 5: Performance of Constant Line Size Set Associative Cache algorithms. Caches $C_i^3(j), i = 7, \ldots, 13, j = 1, \ldots, 4$.

In this paper, techniques for single-pass simulation of classes of direct mapped and set associative caches are presented assuming bit selection and LRU. Three classes of caches are considered. Direct mapped caches of constant LS and varying NS, direct mapped caches of constant CS and varying NS and set associative caches of constant LS and varying NS and DA. It is shown that the caches can be represented using binomial trees or generalized binomial trees. Algorithms are developed for the simulation of entire classes based on these data structures. Analytical and empirical comparisons are made between the new algorithms and algorithms such as forest simulation and all-associativity simulation published earlier in literature. Analytical comparisons show that the new algorithms always require fewer comparisons. The actual difference in comparisons depends on trace characteristics. But the cost of each comparison is higher for the new algorithms. Empirical evaluations reveal that the constant line size direct mapped cache algorithm performs about the same as the forest simulation algorithm. The constant cache size algorithm achieves about a factor of two performance improvement over the naive algorithm, the only other algorithm known for this class. The constant line size set associative cache algorithm performs about a factor of 1.5 times better than all-associativity simulation and GFS for most traces. But for traces with a large working set size the performance of all-associativity simulation is poor.

Apart from the utility of the algorithms for cache simulation the binomial tree data structures present a new way of looking at classes of caches. There is the notion of sets dominating other sets which decides what smaller caches contain. A duality between spatial and temporal locality is demonstrated. It is argued that the classes of caches considered correspond to different trade-offs: spatial locality – temporal locality and locality – cost.

Considerable research has been done on developing models for program behavior [14, 16, 15, 1]. Recent work has focused on access patterns of workloads in order to develop effective memory hierarchies. Many analytic models do not consider the constraints such as bit selection, under which caches and memory hierarchies are designed. In this context, the present work provides an interesting representation of a range of design alternatives. Independent of its utility in actual simulation of caches, the binomial tree algorithms could form the basis for locality models that consider the constraints of cache design.

Some directions for future work are:

1. Generalizing the algorithms to simulate multi-level caches.
2. Using the binomial tree representations for exploring locality models.

References

[1] A. Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming.* PhD thesis, Stanford University, 1988. Available as Technical Report CSL-TR-87-332.

[2] A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. In *Proc. of 17th Intl. Symp. on Computer Architecture*, pages 270–279, 1990.

[3] M. R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM J. of Computing*, 7:298–319, 1978.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture — A Quantitive Approach.* Morgan Kaufmann Publishers Inc., 1990.

[5] M. D. Hill. *Man page of* tycho.

[6] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California, Berkeley, 1987. Available as Technical Report UCB/CSD 87/381.

[7] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. on Computers*, 38(12):1612–1630, December 1989.

[8] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, C-37(11):1925–1936, November 1988.

[9] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[10] S. A. Przybylski. *Performance Directed Memory hierarchy Design.* PhD thesis, Stanford University, 1988.

[11] T. R. Puzak. *Analysis of Cache Replacement Algorithms.* PhD thesis, University of Massachusetts, Amherst, 1985.

[12] D. R. Slutz and I. L. Traiger. One pass techniques for the evaluation of memory hierarchies. Technical Report RJ892, IBM, 1971.

[13] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, Sept 1982.

[14] J. R. Spirn. *Program Behaviour: Models and Measurement.* New York: Elsevier/North-Holland, 1977.

[15] H. S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, 2^{nd} edition, 1987.

[16] D. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Trans. on Computers*, 38(7):1012–1026, July 1989.

[17] J. Vuillemin. A data structure for manipulating priority queues. *Comm. of the ACM*, 21:309–315, 1978.

[18] W-H. Wang and J-L. Baer. Efficient trace-driven simulation methods for cache performance analysis. In *Proc. ACM SIGMETRICS Conf.*, pages 27–36, 1990.

Appendix

Generalized Binomial Trees

Definition 1: The following is a definition by construction of a GBT (Fig. 14). A GBT of degree zero $B_0(n)$ is a list of length n. A GBT of degree x $B_x(n)$ is constructed by putting together two GBTs of 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 from the root-lists 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 4: A GBT(n) of degree x has $n2^x$ nodes.

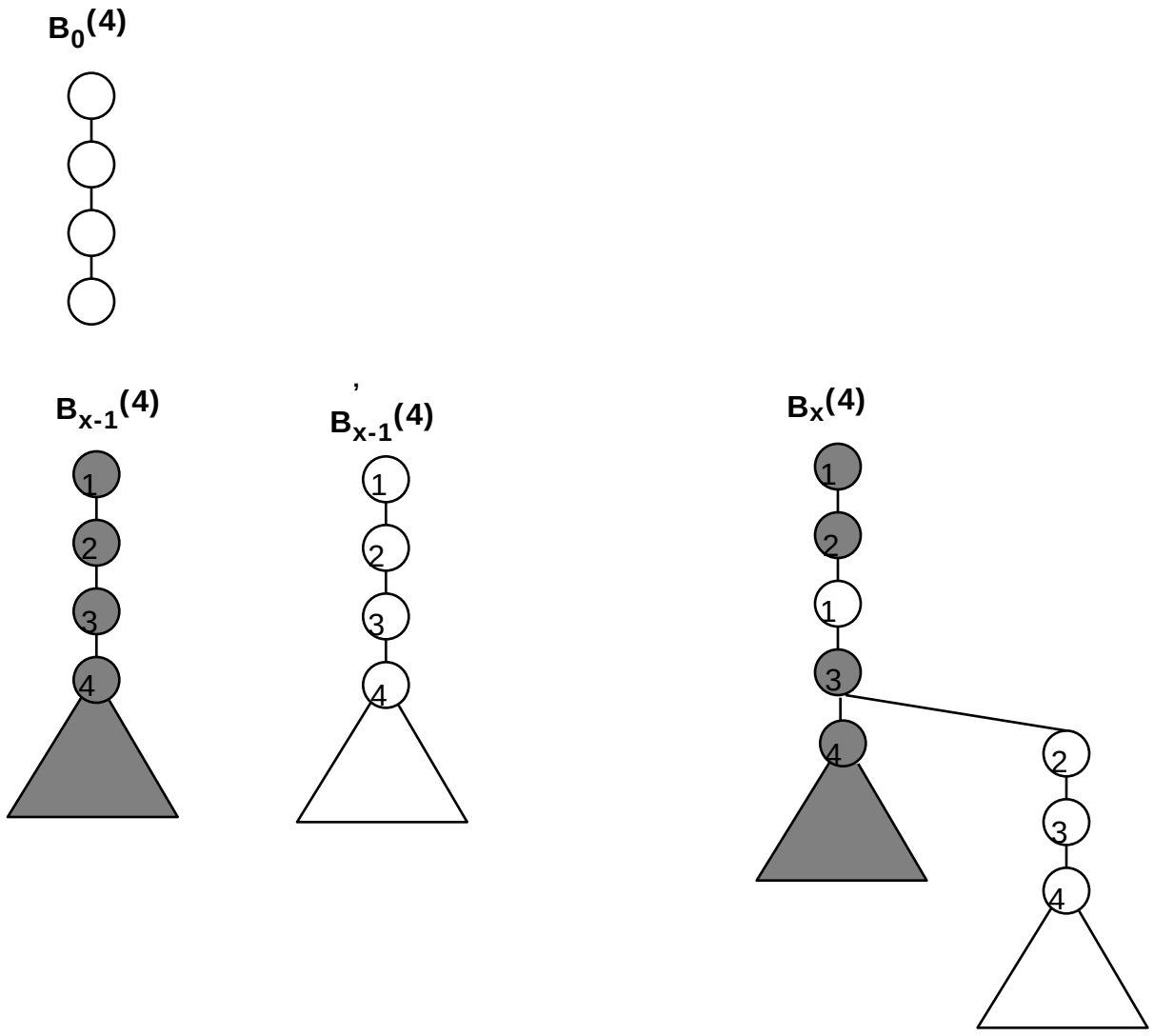


Figure 14: Definition of the generalized binomial tree

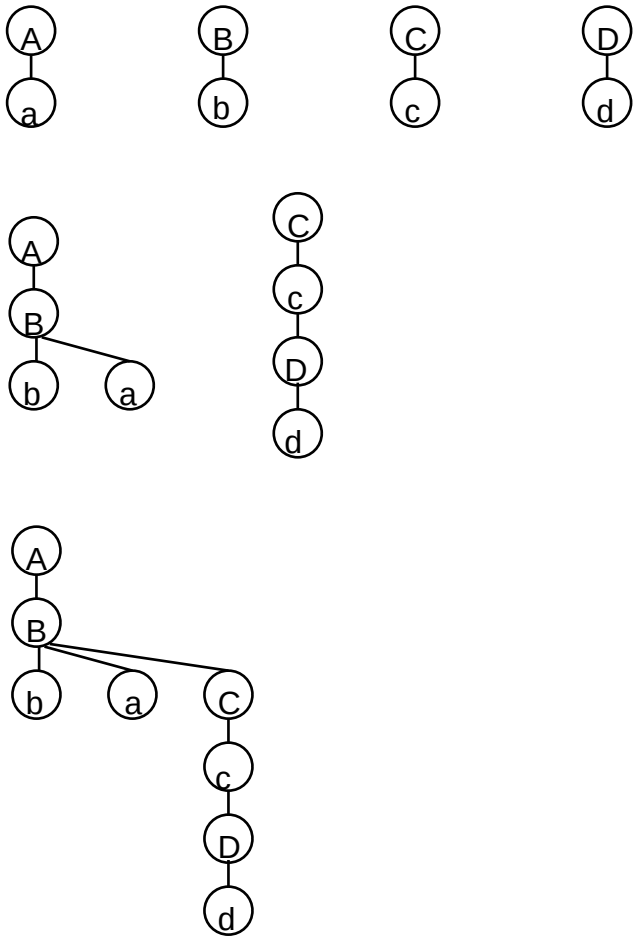


Figure 15: Example GBT to illustrate definitions

Proof :

By induction on the degree. □

Definition 2: The *rank* of a node in a GBT is defined to be $log(\lceil \frac{\text{Number of descendants (inclusive)}}{\text{n}} \rceil)$.

Definition 3: The tree rooted at a node is obtained by pruning all the ancestors of a node.

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

The following lemma proves that the definition of degree above is consistent with its usage earlier.

Lemma 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: $\log \frac{n2^x}{n} = x$. The rank of the last node is: $\log \lceil \frac{n2^x-(n-1)}{n} \rceil = \log \lceil 2^x - 1 + \frac{1}{n} \rceil = x$. Clearly, the rank of the other nodes in the root-list is also x. □

Lemma 6: The rank of any node in a GBT is a whole number.

Proof :

By induction. □

Definition 5: 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.

Lemma 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 a node cannot have two list-children. □

Lemma 8: If a node of rank k in a GBT does not have a list child it has tree-children of rank 0 to $k-1$. If it has a list-child of rank $k-r$, then it has tree-children of rank $k-j, j = 1, \ldots, r$.

Proof :

By induction. The statement is true for all nodes in a GBT of degree zero because the nodes have only list-children. Assuming the statement is true for GBT of degree i, we prove that it is true for a GBT of degree $i+1$. A GBT of degree $i+1$ is formed by combining two GBTs of degree i. All nodes not in the root-list of the degree $i+1$ 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 $i+1$ GBT, excepting the last node, have only list children; the lemma is hence true for these nodes. The rank of the last node in the root-list increases to $i+1$. If node has a list-child of rank $i-r$ (or does not have a list-child), by the induction assumption the node has tree-children of rank $i-r$ to $i-1$ (or 0 to $i-1$); the new child it got as a result of the combining is of rank i. The lemma thus holds for the last node of the root-list too. The proof follows by induction. □

It follows from the lemma above that the list-child is the child of lowest rank.

Definition 6: A subtree of degree r at a node is the tree left after pruning tree-children of rank r and greater, and ancestors. The smallest subtree at a node is the tree left after pruning all tree-children and ancestors.

Lemma 9: Degree of a subtree = $\log\lceil \frac{\text{Number of nodes in subtree}}{\text{n}} \rceil$

Proof :

By induction. □

From the above lemma it follows that the definition of degree for a subtree is the same as the definition of degree for a tree.

Lemma 10: If a node of rank k in a GBT does not have a list-child, it has subtrees of degree 0 through $k-1$. If the node has a list-child of rank $k-r$, it has subtrees of degree $k-j, j = 0, \ldots, r$.

Proof :

By induction along the lines of the proof for Lemma 8. □

We illustrate the definitions using Fig. 15. The figure shows how a GBT(2) of degree 2 is 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. 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. The subtrees rooted at B are $\{B, b\}$ of degree zero, $\{B, b, a\}$ 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.

Intuitively, a GBT is a result of merging lists and binomial trees. Accordingly nodes can have two kinds of children: tree-children and list-children. A node can have only one list-child however, since whenever a node gets a new list-child, it has to be one of the top $n-1$ nodes and it loses all of its other children. List-children are not separated from a node in the definition of subtrees. Since a binomial tree (GBT(1)) does not have list-children, the above definition of subtrees may be shown to be the same as the earlier definition.