Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization

Rabin A. Sugumar and Santosh G. Abraham
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
email: rabin@eecs.umich.edu, sga@eecs.umich.edu

ABSTRACT

Cache miss characterization models such as the three Cs model are useful in developing schemes to reduce cache misses and their penalty. In this paper we propose the OPT model that uses cache simulation under optimal (OPT) replacement to obtain a finer and more accurate characterization of misses than the three Cs model. However, current methods for optimal cache simulation are slow and difficult to use.

In this paper we present three new techniques for optimal cache simulation. First, we propose a limited lookahead strategy with error fixing, which allows one pass simulation of multiple optimal caches. Second, we propose a scheme to group entries in the OPT stack, which allows efficient tree-based fully-associative cache simulation under OPT. Third, we propose a scheme for exploiting partial inclusion in set-associative cache simulation under OPT.

Simulators based on these algorithms were used to obtain cache miss characterizations using the OPT model for four SPEC benchmarks. The results indicate that miss ratio improvements of up to 125% in fully-associative caches, and up to 40% in two-way set-associative caches are possible with replacement policies more sophisticated than LRU.

1 Introduction

The increasing gap between CPU cycle times and memory access times has made caches important in computer systems. Cache misses contribute significantly to the cycles per instruction (CPI) figure of computers [16]. The effect that cache misses have on performance has led to research on methods to reduce cache misses or their penalty. Apart from changing the cache configuration, other strategies such as remapping basic blocks in memory [8, 15], blocking algorithms [4], using two or more levels of caching, miss caches [10] and shadow directories [20] have been proposed. Models which can explain or classify misses are useful both for evaluating the gain from various strategies and for insight on the kinds of strategies that would be useful. Some such models are those that have been proposed by Thiebaut and Stone [22], Agarwal [1], and Hill [6, 5]. In Thiebaut and Stone's model and in Agarwal's model expressions for miss rates are derived in terms of a few trace dependent parameters.

In Hill's three C's model, misses are classified into three categories: compulsory misses, which are misses that occur on first time references to lines, capacity misses, which are misses resulting from the limited capacity of a cache, and conflict misses, which are all the rest and occur owing to mapping constraints in practical caches.

In this paper we propose the OPT model, which provides a finer characterization of misses than the three Cs model. The OPT model differs from the three Cs model in two respects. First, it defines capacity misses as the non-compulsory misses from a fully-associative cache with OPT replacement rather than LRU replacement.[1] Since the performance of LRU is non-optimal,[2] the misses in a fully-associative LRU cache are not a true measure of the misses caused by limited capacity. For instance, when the miss ratio of a fully-associative LRU cache is greater than that of a direct mapped cache, the conflict miss component of the three Cs model is negative which is contradictory to the intuitive concept that conflicts increase the total number of misses. Second, the OPT model separately identifies mapping misses that occur as a result of the set mapping strategy and replacement misses that occur as a result of sub-optimal replacement inside a set. In the three Cs model this distinction is not made. The replacement strategy is thus included in the OPT model. The miss characterizations are for a fixed line size in the OPT model as in the three Cs model.

The OPT model is practical only if caches can be simulated efficiently under optimal replacement. There are two constraints in simulating optimal caches. First, the current optimal cache simulation algorithm that can simulate a range of caches requires two passes over the address trace, where the first pass is a reverse pass. The reverse pass is difficult even when the trace is stored. In order to avoid storing long traces the trace is often simulated as it is generated. The reverse pass is not compatible with such on-the-fly simulation. Second, the current algorithms for simulating optimal caches are not efficient in contrast to those for simulating LRU caches. In this paper we address both issues. We describe an algorithm which simulates optimal caches without a reverse pass but by using a limited lookahead, and then we present schemes to do optimal cache simulation more efficiently. We have implemented the algorithms and we present miss characterizations for four SPEC benchmarks using our simulators.

The following section contains a brief overview of earlier relevant work on cache simulation. The limited lookahead algorithm is described in Section 3. In Section 4, a scheme for a tree-based simulation is described and an extension of the limited lookahead algorithm for set-associative caches is presented. Results of empirical performance evaluations of the various algorithms are presented in Section 5. The OPT model for characterizing misses is described, and a miss characterization for four of the SPEC benchmarks is presented in Section 6. Section 7 concludes the paper.

[1] More precisely, Hill [6] defines capacity misses as the non-compulsory misses from a fully-associative cache using the same replacement strategy as that in the cache being characterized. Empirical characterizations have used LRU [6], and practical caches usually use LRU.

[2] A common access pattern for which LRU is non-optimal is a sequential loop [19].

2 Related Work

In optimal replacement [12, 2], on a miss, the address that is displaced from the cache is that address which is accessed farthest in the future. The optimal replacement algorithm therefore requires future information, and is not a practical replacement strategy. The algorithm minimizes bus traffic over the class of all replacement algorithms when write-backs are ignored. It minimizes the cache miss ratio as well over the class of demand fetching algorithms.

Belady [2] describes an algorithm for simulating a single cache under optimal replacement, where the decision on which address is replaced is delayed till there is no uncertainty. Recently, Mcfarling [13] has developed a simulator for single caches under optimal replacement. In 1970, Mattson *et al* [12] developed the notion of single-pass simulation for caches where multiple cache configurations are simulated together efficiently. They describe an algorithm for simulating a range of fully-associative cache sizes with a fixed line size. This simulation algorithm is applicable when the replacement algorithm is one of a class (stack algorithms) which ensure that at any instant a cache includes the contents of all smaller caches (inclusion property). When knowledge about future references is available, optimal replacement is also a stack algorithm. Mattson *et al* proposed using a reverse pass to gather future information. Single-pass simulation as described by Mattson *et al* is superior to single cache simulation algorithms when a range of cache sizes need to be evaluated. However, the need for a reverse pass makes the single-pass simulation difficult. In this paper we propose a limited lookahead scheme to do single-pass optimal cache simulation without the reverse pass.

The sequential list-based search of the stack is a bottleneck in stack simulation. This problem has been addressed for LRU replacement [3, 17, 23, 11]. These algorithms make use of the fact that the stack update is simple in LRU; the referenced entry is just moved from its current position to the top of the stack. Under OPT replacement, however, the relative positions of many entries other than the referenced entry can potentially change in the stack. In this paper we introduce a scheme for grouping stack entries, so that the relative position of at most one of the entries in a group changes. This grouping enables efficient tree or multi-level list implementations for optimal replacement too.

Single-pass set-associative cache simulation where the number of sets and the associativity are varied has been considered before for LRU replacement. Three algorithms have been proposed: All-associativity simulation [12, 7], generalized binomial forest simulation [21] and generalized forest simulation [7]. All-associativity simulation and generalized binomial forest simulation are not applicable with optimal replacement, but a restricted version of generalized forest simulation is applicable, which we describe.

3 OPT Simulation with Finite Lookahead

We first describe the overall simulation system and then go into details of the unknown-handler routine which fixes errors in the stack.

3.1 The Simulation System

The simulation consists of two phases that are executed alternately: a lookahead phase and a stack processing phase. These two phases communicate through a buffer consisting of a list of (Address, Time of next reference) tuples.

For each address it reads in from the trace, the lookahead phase enters a tuple at the tail of the list with the "Time of next reference" attribute initially set to "unknown". Then, the lookahead phase uses a hash table to check if the address has been referenced earlier. If the previous reference has not yet been processed by the stack processing phase, the position of the previous reference in the buffer is also determined from the hash table and the "Time of next reference" attribute of the previous reference to the same address is changed from "unknown" to the current time. Otherwise, the previous reference is an *unknown* in the stack whose actual time of next reference and hence priority is not known. The true priority is now available for this reference, and the lookahead phase calls a special routine, the *Unknown-Handler*, that converts the unknown to a known (i.e., an address whose priority is known), places the address in its true position in the stack, and rearranges the stack so that subsequent unknown conversions can be handled correctly.

The stack processing phase removes entries from the head of the list using the "Time of next reference" attribute to assign priorities to each address. Addresses whose "Time of next reference" is unknown are treated as having lower priority than all knowns. Stack processing is done as described in [12] where the contents of all the caches simulated are maintained in a single stack. The j^{th} line from the top in the stack (the line at depth j) is that line which is not in the cache with $j-1$ lines, but is in the cache with j lines. In the simulation, for each trace reference, the stack is searched for the referenced line, and the stack depth at which it is found is recorded. The reference is a hit in caches with at least as many lines as the stack depth, and is a miss in smaller caches. The stack is then updated to reflect the new state of the caches. During the update, the line at the top of the stack is displaced by the referenced line. For each depth j up to the hit depth, the priority of the deleted line, y, from the previous depth is compared against the priority of the line, z, at the current depth, and the line of lower priority becomes the deleted line from the current depth.[3] We say an *interaction* occurs between y and z in such a case, and that the deleted entry has been *displaced* by the other.

The hit depth determined by the stack processing phase is the same as it would have been had infinite future information been available. Consider unknown-known interactions; the unknown is always displaced because it is treated as having a lower priority than knowns. But unknowns are of lower priority than knowns in reality, i.e., even when infinite future information is available; there is hence no error in unknown-known interactions. Since there is no error in known-known interactions either, addresses that enter the stack as knowns are in their true position, i.e., their position had infinite future information been available. However, potential errors are made when unknowns interact, since no information is available on their relative priorities, and unknown addresses may not necessarily be at

[3] If cache bypass is allowed, the referenced address is treated as the address deleted from depth 0. Optimal replacement with bypass is discussed in [14].

their true stack position. Once an unknown becomes known it is moved to its true position by the unknown-handler and we describe how this is done in the next section. But assuming that the unknown-handler works correctly, we can say that all knowns are in their true position in the stack. By the time an address reference is processed in the stack processing phase, the address is a known in the stack (unless it is a first reference in which case the address is not in the stack at all), and therefore the stack processing phase finds it in its true position.

3.2 Unknown Handling

In this section we describe the unknown-handler. We start with a formal description of the action of the unknown-handler, and then formulate the notion of an interaction graph which is used to describe the unknown-handler. We then describe and prove a simple practical implementation, where by the proper assignment of dummy priorities to unknowns partial information about the interaction graph is automatically encoded in the stack.

Let $S(t,\tau)$ denote the set of stacks for which lookahead has been done till time τ and stack processing has been done till time t ($< \tau$). In the stacks in $S(t,\tau)$, the unknowns are those addresses that have been referenced before t, but not in the interval (t, τ). The positions of knowns are identical in the stacks in $S(t,\tau)$, but that of one or more unknowns is different. Let the next reference to an address that is an unknown occur at time $t+\delta$. When the unknown-handler is called on this reference, it takes the stack $s(t,\tau)$ ($\in S(t,\tau)$) formed in the course of the simulation, and forms a stack $s(t,\tau+\delta)$ ($\in S(t,\tau+\delta)$), with the unknown that became known in its true position.

With any stack, $s(t,\tau)$, an *interaction graph* is associated. The interaction graph is a directed graph which has information on unknown-unknown interactions for all the unknowns in the stack. The vertices of the interaction graph correspond to the unknowns in the stack, and each edge represents an interaction between the two unknowns it connects, with the arrow pointing to the unknown that displaced the other. With each edge a number is associated; these numbers are used to determine the order in which the edges were created. At the start of the simulation the interaction graph is empty, and an interaction counter is set to zero. In the course of the simulation, when an unknown enters the stack, a new vertex is created for it in the graph. When an unknown interacts with another unknown, the two unknowns are connected with an edge pointing away from the displaced unknown,[4] the interaction counter is incremented, and the new edge is numbered with the current value of the interaction counter. When an unknown becomes known, the information on unknown-unknown interactions maintained in the interaction graph is used to undo the effect of any wrong decisions, both in the stack and in the interaction graph, and the node of the unknown is deleted from the graph. The procedure for doing this is described below.

When an unknown (U say) becomes known, we know that its true priority is greater than the true priority all other unknowns. Therefore, any interaction on which it has

[4]The decision on which unknown is displaced may be made arbitrarily

been displaced by an unknown in the interaction graph is a wrong interaction. These wrong interactions are fixed as follows. The first wrong interaction is identified — it is the interaction corresponding to the edge pointing away from U with the lowest number. Let that edge be v_{UX}, X being the unknown at the other end of the arc. The corresponding interaction is then reversed by: 1. Flipping the edge representing the interaction being reversed, 2. Swapping later interactions of the two unknowns, and 3. Swapping the stack positions of the two unknowns. As a result of the reversal, whatever was done with U after the wrong interaction, is now associated with X and vice-versa. In the interaction graph after the reversal the first interaction on which U was displaced (if any) is identified similarly (this interaction was originally associated with X), and is reversed as before. This procedure is repeated until a state is reached where U has not been displaced by any other unknown, and at that point U is in its true position in the stack. The formal procedure is given below.

Algorithm for rearranging the stack and the interaction graph when unknown U becomes known

While U has been displaced by some unknown
    Let V be the first unknown to have displaced U, and v_{UV} that arc from U to V
    For each edge v_{UX} (v_{XU}) ($X \neq V$) with number greater than v_{UV}
        Delete v_{UX} (v_{XU})
        Add edge v_{VX} (v_{XV}) with same number as v_{UX} (v_{XU})
    For each edge v_{VX} (v_{XV}) ($X \neq U$) with number greater than v_{UV}
        Delete v_{VX} (v_{XV})
        Add edge v_{UX} (v_{XU}) with same number as v_{VX} (v_{XV})
    Flip all edges between U and V with a number greater than or equal to that of v_{UV}
    Swap the positions of U and V in the stack
End while
Delete U and all edges adjacent to U

3.3 Unknown handling through a specific dummy priority assignment

The unknown handling scheme using the interaction graph, described in the previous section, works for arbitrary unknown-unknown interaction policies. However, maintaining and traversing the interaction graph is complex, and in this section we present a simpler scheme. In this scheme dummy priorities are assigned to unknowns. The stack processing routine treats unknowns and knowns similarly, taking replacement decisions based on their dummy or actual priorities. The dummy priority assignment forces an automatic encoding of information on which unknowns interacted, but not on the order in which they occurred. Unknown handling may be done even without this order information as we describe below.

The dummy priorities are assigned following the two rules below:
1. Each unknown is given a dummy priority lower than the priority of all knowns.
2. The dummy priorities are given in decreasing order, i.e., the dummy priority of an unknown is less than the dummy priority of all earlier unknowns.

The first rule ensures that on a known-unknown interaction the unknown is always

displaced, and the second rule ensures that on an unknown-unknown interaction, the later unknown is always displaced. Whenever an unknown is lower in the stack than another unknown with a higher dummy priority number, we know that the two unknowns have interacted, since the former entered the stack later, and is now lower. In contrast, when an unknown is higher in the stack than another of higher dummy priority number, we know that the two have not interacted. Therefore, the stack automatically maintains information on which unknowns have interacted, but it does not have precise information on the order in which the interactions occurred. Therefore, there is no one-to-one mapping from the stack to the interaction graph. The unknown-handler needs to use the partial information in the stack to move the unknown that becomes known to its true position, and then rearrange the stack so that the next unknown may be fixed similarly. More formally, let $s'(t,\tau)$ denote the stack, where stack processing has been done till time t, and lookahead has been done till time τ $(> t)$ with the above dummy priority scheme for the unknowns. Note that $s'(t,\tau)$ is in $S(t,\tau)$. Assuming that $(\tau+\delta)$ is the first reference to an address which is an unknown in $s'(t,\tau)$, the unknown-handler takes the stack in state $s'(t,\tau)$ and moves it to state $s'(t,\tau+\delta)$. We argue below that the final stack is $s'(t,\tau)$ whatever interaction graph is assumed, after the unknown that becomes known is moved to its right position, and any wrong displacements that are caused by the move are fixed.

Define the move set, M, of unknowns above U in the stack as follows

$$
\begin{aligned}
M \;=\; & \{\text{Unknowns } u \text{ s.t} \\
& \quad 1.\ \text{The } DP(u) \geq DP(U) \text{ and } SD(u) < SD(U) \\
& \quad 2.\ \nexists \text{ an unknown } u' \text{ s.t. } DP(U) < DP(u') < DP(u) \text{ and} \\
& \quad SD(u') < SD(u)\}
\end{aligned}
$$

where $DP(u)$ denotes the dummy priority of u and $SD(u)$ denotes the stack depth of u.

Let $M = \{U_1, U_2, \ldots, U_n\}$ be the move set of U_1; U_i is above U_j in the stack if $i > j$. .

Lemma 1: The first unknown to displace U_i is one of U_{i+1} to U_n .

Proof :

By the definition of the move set the dummy priority of U_i is less than the dummy priority of U_{i+1} to U_n. Therefore U_i came after $U_{i+1}, \ldots, U_n$ and since it is now below those unknowns, it has been displaced by them. We now show that any other unknown in the stack that has displaced U_i has done so after U_i had been displaced by one of U_{i+1} to U_n. Consider any such unknown and let it be above U_k $(k > i)$ in the stack. The dummy priority of U_k is lower than the dummy priority of this unknown by the definition of the move set. Therefore, since U_k is higher than this unknown in the stack, U_k and this unknown have never interacted, and hence U_k has always been higher than this unknown from the time of entry of U_i into the stack. U_i would therefore have been displaced first by U_k before being displaced by this unknown. □

Lemma 2: The true position of U_1 is the position of U_n.

Proof :

The first unknown to displace U_1 is one of U_1 to U_n (by Lemma 1). Assuming it is some U_i, after the first reversal U_1 is in the position of U_i, and U_1 gets all the interactions of U_i after the U_1U_i interaction. This includes the first interaction on which U_i was displaced (otherwise U_1 would have interacted with that unknown first rather than U_i), and this is the first interaction to displace U_1 in the modified interaction graph (since U_1 itself does not have any interactions on which it was displaced). Therefore in the modified interaction graph U_1 was first displaced by one of U_{i+1} to U_n (by Lemma 1). In the next reversal U_1 goes to the position of U_j, $i+1 \leq j \leq n$, in the next to U_k, $j+1 \leq k \leq n$, and so on till it reaches U_n. By the definition of the move set, U_n has not been displaced by any unknown, and so that is the true position of U_1. □

As described above U_1 is moved to the position of U_n by successively reversing interactions. But each reversal potentially introduces wrong displacements (i.e., interactions where an unknown displaces an unknown of higher dummy priority number) in the stack, which have to be set right in order to use the stack to fix the next unknown. Consider the first reversal, i.e., the reversal of the U_1U_i interaction. U_i now gets all the interactions of U_1 after the U_1U_i interaction, which includes wrong displacements by unknowns u s.t $DP(U_i) > DP(u) > DP(U_1)$. These wrong displacements are fixed by successively reversing the first wrong displacements of U_i till there are no more wrong displacements. The procedure for doing this is similar to that for moving U_1 to U_n, and for similar reasons the other unknowns involved are in the set $\{U_2, \ldots, U_{i-1}\}$. U_i does not have any wrong displacements when it reaches the position of U_{i-1}. Each such reversal may introduce further wrong displacements for the other unknown and these wrong displacements are similarly fixed. Finally when all wrong displacements have been fixed, U_1 is in the position of U_i, U_i is in the position of U_{i-1} and so on and U_2 is in the position of U_1. In a similar fashion, assuming U_j is the first entry to displace U_1 in the modified interaction graph, when the U_1U_j interaction is reversed and all resulting wrong displacements are reversed, U_1 is in the position of U_j, U_j is in the position of U_{j-1} and so on and U_{i+1} is in the position of U_1. Finally when U_1 reaches the position of U_n, and all wrong displacements are fixed, U_n is in the original position of U_{n-1}, U_{n-1} is in the original position of U_{n-2} and so on with U_2 being in the original position of U_1. This would have happened whatever the actual interaction graph. Also since the interactions that gave $s'(t,\tau)$ have been redone to give a stack where the unknown that becomes known is not displaced by other unknowns, and there are no wrong displacements, the resulting stack is $s'(t,\tau+\delta)$.

The algorithm is presented below. It traverses the stack top down looking for entries that are in the move set of the unknown that becomes known. It inserts the unknown with its true priority at the location of the first entry in its move set. Other entries in the move set are shifted to the position of the next entry in the move set. The algorithm quits when it reaches the original position of the unknown.

Fig. 1 shows an example illustrating the actions of the unknown-handler. Here the

Unknown-Handler Algorithm:

Input: *ADDR* - Address of unknown that has become known
PRTY - New priority of address

Procedure:

```
DPA = dummy priority of address (obtained from a hash table or by a stack lookup)
DE→prty = PRTY
DE→addr = ADDR
entry_ptr = head of stack
while (entry_ptr→addr ≠ ADDR)
        if ((entry is unknown) && (entry_ptr→prty > DPA))
                if (entry_ptr→prty < DE→prty)
                        Replace entry with DE and make entry the new DE
        entry_ptr = Pointer to next stack entry
Replace entry with DE
```

priority numbers are integers from $-MAXINT$ to $MAXINT$. An address with a higher priority number is of greater priority. All knowns have priority numbers that are greater than zero and all unknowns have priority numbers that are less than zero.[5]. The figure shows the state of the stack just before address H becomes a known, and the resulting stack after rearrangement by the unknown-handler. H goes to the position of C which is the highest unknown with a dummy priority greater than or equal to H. C replaces D, and D moves into the original position of H.

The complexity of the unknown-handler algorithm above is O(Depth of unknown in stack).

The memory requirement of the simulation algorithm above cannot be bounded. This is most easily seen by considering a sequential access pattern, where no address is referenced twice. All the lines in the stack are unknowns, and none of the lines can be removed, since it is not known which of them is going to be first referenced again. This appears to be a fundamental problem with OPT simulation, and is present in both earlier techniques for OPT simulation, viz., Belady's single cache algorithm, and Mattson *et al*'s two-pass algorithm. In practical traces, when the cache size is limited, some of the unknowns may be identified as unnecessary. Consider the unknown (U_{max}) of highest dummy priority up to the largest depth of interest, C, in the stack. Any unknown of higher dummy priority than that of U_{max} has left the largest cache of interest and may be deleted. Conversely any unknown of lower dummy priority will move up above depth C if it becomes a known now, and needs to be kept around. A cleaning step is done periodically to remove the unnecessary unknowns.

[5]For instance, the priority number of a known may be (MAXINT − time of next reference), while the priority number of an unknown may be −(number of unknowns seen earlier)

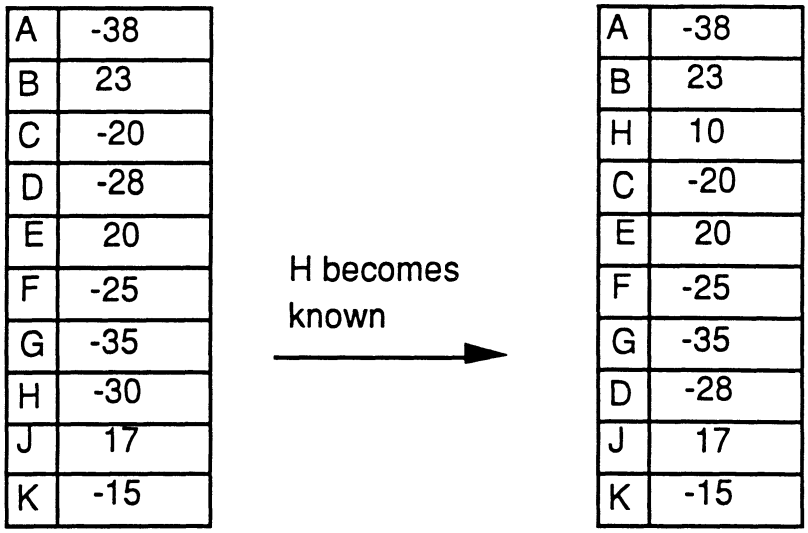

Figure 1: Example for Unknown-Handler

4 Efficient Algorithms for OPT Simulation

In this section we describe two techniques for making OPT simulation faster, in the context of the limited lookahead scheme described above. First, we describe a technique for breaking the OPT stack into groups, such that during the update process at most one entry may enter or leave a group. This grouping limits the number of stack entries that need to be examined during the update process. It speeds up list implementations and also makes possible tree implementations of OPT stack simulation. Second, we describe a technique for doing single-pass set-associative cache simulation under OPT which exploits partial inclusion.

4.1 Grouping Algorithm

In Mattson's algorithm the stack is implemented as a list and the execution time is dominated by the list lookup. The limited lookahead scheme described in the previous section enables a single-pass simulation, but the complexity of stack processing remains unchanged. An approach to speeding up the computation is to convert the list lookup into a tree lookup. A technique for doing the update by examining only a small fraction of the stack entries is the key to developing a tree based simulation algorithm for any replacement scheme. In this section we describe one approach to doing this for OPT.

A *group* is defined to be a contiguous region of the stack in which entries are in correct priority order.

The OPT stack is partitioned into groups based on this definition. Note that there is flexibility in partitioning the stack, in that a section of the stack which forms a group may be split into two or more smaller groups without violating the definition. When cache bypass is not allowed, it is convenient to regard the top entry in the stack (the line just

Algorithm for Stack Processing with Groups - Optimal Replacement

```
For each (Address, Time of next reference)-tuple output by the lookahead phase
Assign priority to address based on the time of next reference
If (Address == Address at top of stack)
        Change priority of top of stack and record hit
Else
        Delete first stack entry and make it DE
        Insert an entry for the referenced address at the top of the stack
        For each group in the group list starting at the top
                If (address == address at top of group)
                        Record depth of hit
                        Delete top of group
                        If this is the first group
                                Create a new group consisting of DE
                        Else
                                Add DE to end of previous group
                        Break
                Else
                        If (priority of DE > priority of last entry in group)
                                Insert current DE in group
                                Delete Last entry in group and make it DE
        Add DE to end of last group
```

referenced) as a special case, not belonging to any group. If cache bypass were allowed the top entry would be the first entry in the first group, but otherwise the algorithm is identical. In the descriptions below we assume that cache bypass is not allowed.

Lemma 3: When optimal replacement is followed, hits always occur at the top entry in the stack or at the first entry of a group.

Proof :

In OPT a line of higher priority will always be referenced before lines of lower priority. Since, by definition, the line at the top of a group is of higher priority than the lines inside the group, the lemma follows. □

The algorithm for doing stack processing with groups is shown. In the algorithm, for each trace address, the top of the stack is first examined, and then the head of each group is examined till the address is found or the end of the group list is reached. When the address is not found up to a certain depth in the stack, some entry has to be displaced from the section of the stack up to that depth to make space for the address. This entry is called DE (for displaced entry) in the algorithm above. When the address is not found at a group, the entry displaced from the section of the stack including the group is either the DE of the section of the stack above this group, or the last entry of the group, whichever is of lower priority. When the last entry of the group is of lower priority, the current DE is inserted into its right position in the group (this can be anywhere in the group), and the last entry of

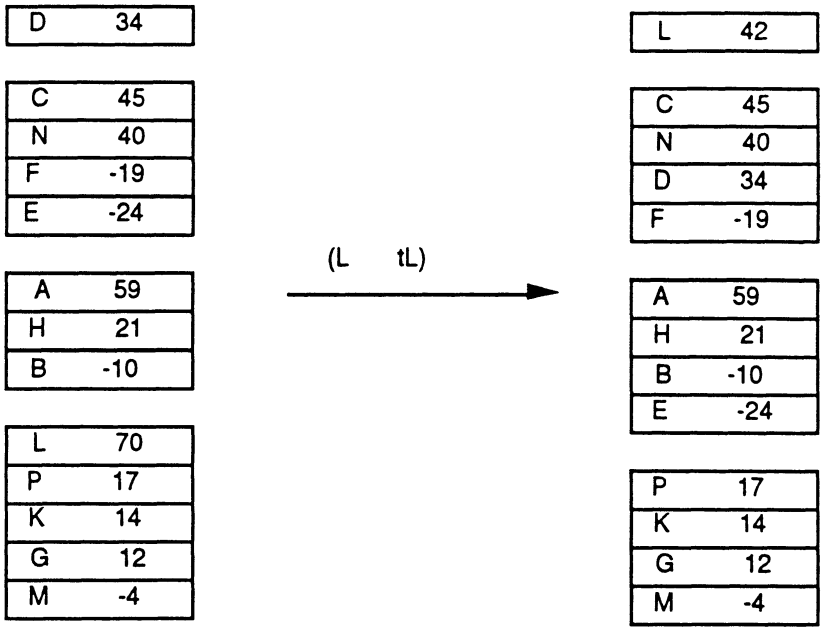

Figure 2: Grouping algorithm – Example

the group is made the DE; otherwise the DE is left unchanged. When the address is found at a group, it is deleted, and the current DE is inserted at the position of the address. Since the priority of the current DE is less than the priority of all the addresses in the previous group, it can be made a part of the previous group. However, when the hit occurs at the first group (the second entry in the stack), there is no previous group and DE is inserted into the second stack position and made a new group. This is the only event that results in the creation of a new group. When a group becomes empty it is deleted from the group list. When entries are deleted from groups, it is possible that the group can be combined with the previous or next group. Such combining is not shown in the algorithm above for simplicity.

An example is shown in Fig. 2. The state of the stack just before the arrival of address L is shown on the left. The priority assignment is the same as that used in Fig 1, with higher numbers denoting greater priorities and with unknowns having negative dummy priority numbers. On the arrival of address L, address D at the top of the stack is first examined. It is then removed from the top of the stack and L is put in at the top. D is the DE when the simulation comes to the first group. The first entry in the first group is then examined (Address C), but there is no match. Since the priority of D, the current DE, is greater than the priority of the last entry of the group (Address E), E is made the DE and D is inserted into the first group. The first entry in the second group (Address A) is then examined, but again there is no match. Since the priority of E is less than that of the last entry in the group (Address B), E continues to be the DE. L is the first entry of the next group and a hit occurs at depth 9 in the stack. E is inserted in place of L and is made a part of the previous group, i.e., the second group. The resulting stack is shown on the right in Fig. 2.

The complexity of the algorithm depends on the number of groups passed and the cost

of inserts and deletes. When the groups are maintained as lists, the insert operation could be O(Number of stack entries), but since inserts need to be done only for some of the groups passed, it is less expensive than a complete search of the stack up to the hit depth as done in the list implementation. Therefore, using groups would speed up a list implementation. By maintaining the groups as trees the algorithm may be speeded up even further. The complexity of the algorithm depends on the number of groups passed on the average, and the tree operations (when the groups are maintained as trees) required at each group. Roughly the complexity is

$$O(\text{Mean groups passed} \times \log(\text{Mean tree weight}))$$

An unknown handling scheme similar to the one described in the previous section is applicable here too. The tree based group implementation helps speed up unknown handling too. We omit the details for the sake of brevity.

4.2 Set-Associative Cache Simulation under OPT

In set-associative caches, the cache is divided into sets and each address maps to a particular set. The maximum number of lines that each set may contain is called the degree of associativity (associativity, in short) of the cache, and the number of sets in a cache is called the *degree of mapping.* For a given replacement policy, the space of set-associative cache configurations is three dimensional, the parameters that may be independently varied being the cache size, the line size and the degree of associativity. Generalized forest simulation (GFS) is a single-pass cache simulation algorithm where two parameters, viz. degree of mapping and degree of associativity, can be varied over a range, and LRU replacement and bit-selection are used [7].

Generalized forest simulation is based on the following inclusion properties:
1. The contents of a cache with 2^m sets and associativity k is contained in a cache with 2^m sets and associativity greater than k.
2. The contents of a cache with 2^m sets and associativity k, is contained in a cache with 2^{m+1} sets and associativity k.

In GFS, for each degree of mapping, 2^m, considered, a stack is maintained for each of the 2^m sets. Each of the stacks is processed as described earlier for fully-associative cache simulation. Once an address is found in a stack at a depth d, the address is known to miss only in caches with associativity less than d, for that mapping degree because of inclusion property 1. Therefore, the rest of the stack does not have to be examined. Similarly, when an address hits at the top of the stack for a mapping degree, it will hit at the top of the stack for larger mapping degrees too, because of the second inclusion property, and those mapping degrees do not have to be examined.

Under optimal replacement only the first inclusion property holds. Fig. 3 shows an example where the second inclusion property does not hold for optimal replacement. The example shows two direct mapped caches, Cache I, having one set and Cache II, having two sets. Cache bypass is allowed. Upper case and lower case addresses map to different

Trace (Address, Priority) :		(A, 10)	(a, 15)	(a, 4)	(B, 8)	(A, 5)	(B, 3)
Cache I (Degree of Mapping = 0 Degree of Associativity = 1)	Invalid	A MISS	a MISS	a HIT	B MISS	B MISS	B HIT
Cache II (Degree of Mapping = 1 Degree of Associativity = 1)	Invalid Invalid	A a MISS	A a MISS	A a HIT	A a MISS	A a HIT	A a MISS

Figure 3: Inclusion does not hold across mapping degrees with optimal replacement

sets in Cache II. Inclusion does not hold after the fourth address is processed, with Cache I containing B, which is not in Cache II, and the last trace address hits in Cache I and misses in Cache II. Therefore, the GFS algorithm cannot be directly applied.

A large fraction of the references hit at the top level for all mapping degrees. The following scheme permits detection and exploitation of such behavior, and thus greatly decreases the number of stacks that have to be examined. With each stack, a flag is maintained, which is set when a reference hits at the top entry in this stack and in the stacks of all higher mapping degrees. Subsequently, when an address hits at the top entry in a stack whose flag is set, the stacks of greater mapping degrees need not be examined for this reference, since it is known that the same address is at the top in those stacks too. When a reference misses at the top entry in a stack whose flag is set, the flag at this stack is reset. Any change in priority of the top-of-stack is propagated to the corresponding stack in the next mapping degree, and simulation continues for the current reference.

Since each stack is essentially an independent fully-associative stack, unknown handling may be done as described earlier.

5 Empirical Performance Evaluations

The tree- and list-based algorithms were implemented and their performance was evaluated on traces from the SPEC benchmarks. The traces were generated using the *pixie* utility. The pixified executable generated the trace, a pre-processor converted the trace into another format, and the cache simulator read this converted trace. All three programs were piped in sequence and executed on a DEC 5000. The trace generation and pre-processing times are not included in the times reported below, but these times are less than 10% of the simulation time typically.

Benchmark	Dist. addr	OPT (list) Time	OPT (list) Entries	LRU (list) Time	LRU (list) Entries	OPT (tree) Time	OPT (tree) Groups	LRU (tr.) Time
spice i&d	31362	20625.5	113.6	29272.9	201.0	3737.7	2.52	1481.6
gcc i&d	>65536	62764	222.0	39737.2	263.5	5599.1	3.31	2015.1
espresso i&d	13185	5102.1	21.87	3830.2	30.26	3342.3	2.16	1396.7
doduc i&d	11995	16240.9	86.62	20056.0	159.5	5322.7	3.52	1647.6

Table 1: Execution times and performance metrics of simulation algorithms

The execution times and critical performance metrics of the list-based OPT (OPT list) algorithm, the tree based OPT (OPT tree) algorithm, an LRU list algorithm, and an LRU tree algorithm on traces generated using four SPEC benchmarks are given in Table 1. For each benchmark, the initial unified trace segment of length 100 million was used in the simulation. All simulations were performed with a cache line size of 16 bytes. In the table, *Dist. addr* represents the number of distinct addresses in the trace, *Entries* stands for the mean number of stack entries passed in the list algorithms, and is also the mean stack depth at which an address is found, and *Groups* is the mean number of groups that are inspected per address reference. All execution times are in seconds and all other metrics are unitless.

The tree-based algorithms use splay trees [18] which are self-balancing and well-suited for cache simulation. In the LRU tree algorithm, the time of previous reference of an address is obtained by a hash lookup. The hit depth is then determined by a tree lookup that uses this time as the key[17, 23]. In the OPT tree implementation, a list of groups is maintained numbered in order. A descriptor is associated with each group that contains the first and last addresses of the group, and the number of entries in the group. All the addresses in the stack are maintained in in-order form in one splay tree, with the current group number of the address as the primary key, and the priority of the address as the secondary key. Trace addresses are located by successively examining the first addresses of groups. The tree is used for insertions, deletions and other stack operations. Unknown handling is done using the dummy priority scheme described in Section 3.

1. *OPT tree Vs. OPT list:* The OPT tree algorithm runs about 1.5 to 10 times faster than the OPT list algorithm. The speedup obtained using the OPT tree algorithm has a strong correlation with the mean stack depth for a trace.
2. *OPT tree Vs LRU tree:* The OPT tree algorithm runs about 2.5 to 3.0 times slower than the LRU tree algorithm. The OPT tree algorithm is slower primarily because more than one tree operation might be required per address. In addition pre-processing and unknown handling steps are required in OPT.
3. *Effect of grouping:* The mean number of groups examined is quite low and ranges from 1.56 to 3.44. This range of values appears to be typical even in longer simulations. More experimentation and analytical modeling are necessary to confirm and explain this behavior.
4. *Unknown handling:* When the lookahead phase is at least 100,000 addresses ahead of the stack processing phase, the unknown-handler is called about 1.5 to 13 times for every 100,000 addresses. This extent of lookahead requires about 1.6 Mbytes in the current implementation. The variation in run time and the number of calls to the unknown-handler

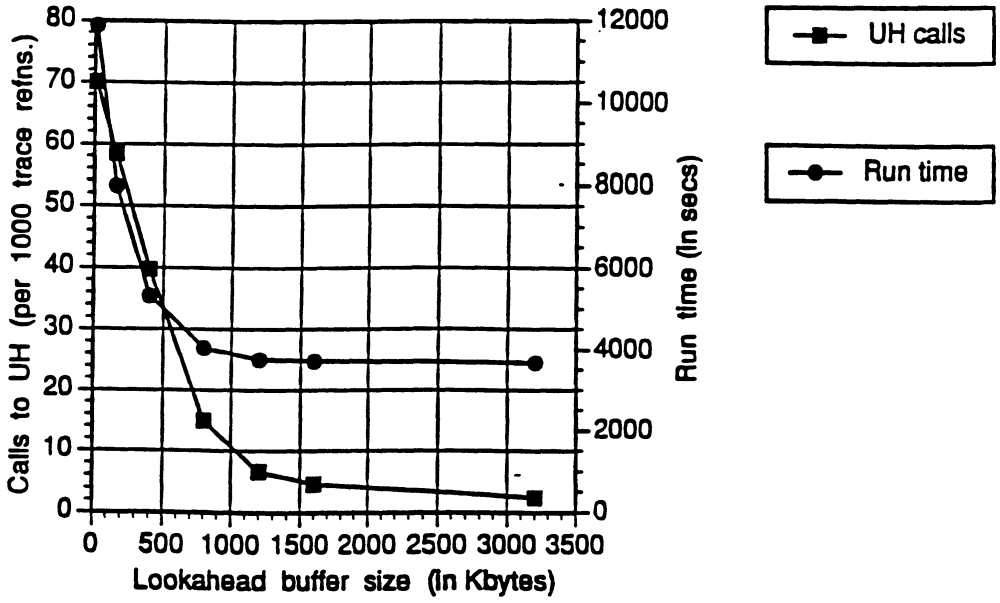

Figure 4: Variations in calls to UH and run time with lookahead buffer space

as the lookahead buffer size (and hence lookahead distance) is varied is shown in Fig. 4 for `spice`. The number of calls to the unknown-handler and the effect of unknown handling on run-time become negligible with a lookahead buffer size of around 1 Mbyte.

5. *LRU list Vs OPT list:* The OPT list algorithm is about a factor of 1.5 faster than LRU list for the `spice` and `doduc` benchmarks because the mean stack depth in OPT is significantly lower than in LRU. LRU list is faster than OPT list for the other two benchmarks. The stack can be grouped in the OPT list implementation and this optimization is estimated to improve the OPT list execution time by a factor of 1.5 to 2.

6 Characterization of Misses

In this section we describe and contrast our miss characterization scheme, OPT model, with the 3Cs model, a previously proposed scheme that is similar in many respects. We describe qualitatively why our miss characterization provides better insight into possible approaches that can be used to reduce cache misses and present a miss characterization averaged over four SPEC benchmarks.

Let M(L, C, k, repl) represent the number of cache misses in a cache where L, C, k, repl are the line size, cache size, associativity, and replacement strategy. Table 2 shows the characterization of cache misses, M(L, C, k, LRU) using the OPT and 3Cs model. In the table, a fully-associative cache is represented using an associativity of C/L. In order to illustrate the characterization, consider the mapping misses in the third row. The expression (M(L, C, k, OPT) - M(L, C, C/L, OPT)) represents the number of extra misses in a cache with a small associativity k as compared to a fully-associative cache of the same cache size

OPT Model		3 Cs Model	
Miss Type	Number of Misses	Number of Misses	Miss Type
Compulsory	M (L, ∞, ∞, any)	M (L, ∞, ∞, any)	Compulsory
Capacity	M (L, C, C/L, OPT) - M (L, ∞, ∞, any)	M (L, C, C/L, LRU) - M (L, ∞, ∞, any)	Capacity
Mapping	M (L, C, k, OPT) - M (L, C, C/L, OPT)	M (L, C, k, LRU) - M (L, C, C/L, LRU)	Conflict
Replacement	M (L, C, k, LRU) - M (L, C, k, OPT)		

Table 2: OPT model and the three Cs model

and line size, both using the OPT replacement strategy. Since these misses arise from the mapping strategy and are affected by the choice of the mapping strategy (e.g. hash-based mapping instead of bit-selection mapping) they are characterized as mapping misses.

The OPT model is an extension of the 3Cs model and gives a finer and more accurate characterization of cache misses. Misses are characterized into four categories instead of three. The compulsory misses are the same in both categories. In the OPT model, the capacity misses are the extra misses occurring in a fully-associative cache of finite size C simulated using the OPT replacement strategy, and therefore accurately model the misses due to the finite capacity of the cache. In this characterization, the 3Cs model uses a fully-associative LRU cache to obtain the capacity misses. Thus, the capacity misses obtained under the 3Cs model exceed the actual capacity misses by M(L, C, C/L, OPT) - M (L, C, C/L, LRU). This overcount in the 3Cs model can be quite significant, as indicated by the results at the end of this section. In the OPT model, conflict misses are more finely characterized into mapping misses and replacement misses. Mapping misses arise because of the small degree of associativity of CPU caches, whereas the replacement misses are due to the sub-optimal replacement strategy. A finer characterization enables a designer to narrow down the set of approaches to reduce cache misses. For instance, if the mapping misses are significant, increasing the associativity can improve the miss ratio while modifying the replacement strategy will have little effect on the overall miss ratio.

The OPT cache simulators were used to obtain the miss ratio characterization using the OPT model for direct mapped caches of varying sizes for instruction, data and instruction & data (unified) traces for four of the SPEC benchmarks: `spice2g6`, `gcc`, `espresso`, `doduc`. All simulations were run to completion or for one billion addresses. Read and write accesses were handled identically. The line size was fixed at 16 bytes. We assume cache bypassing is available and therefore there are replacement misses even in direct-mapped caches. Fig. 5

shows the mean cache miss components averaged over the four benchmarks for a range of direct mapped caches using the OPT model for the three types of traces. Superimposed on this figure is the miss ratio of a fully-associative cache with LRU replacement. This miss ratio represents the sum of the compulsory and capacity misses in the 3Cs model. Fig. 6 shows a similar cache miss characterization for a range of two-way set-associative caches using the OPT model.

The main observation is that the capacity miss ratio under the 3Cs model is between 25% and 125% higher than the true capacity miss ratio obtained by the OPT model and is on the average about 40% higher. In other words, our results indicate that LRU replacement in fully-associative caches is considerably worse than optimal replacement. The anomalous and contradictory characterization occasionally obtained using the 3Cs model (under LRU simulation) is illustrated by the results obtained with a cache size of 256 KB on data traces. In this case, the compulsory and capacity miss components under the 3Cs model is greater than the actual miss ratio in a direct-mapped cache. Therefore, under the 3Cs model, the conflict miss ratio is negative. The OPT model does not lead to such contradictory characterization. The replacement miss component is about 20% of the miss-ratio on the average for the two-way set-associative caches, and is about the same magnitude as the mapping component, indicating that the miss-ratio may be reduced by about 20% using replacement schemes more sophisticated than LRU for two-way set-associative caches.

Some other comments and caveats about the miss components follow. Firstly, the compulsory miss component is negligible in most cases and much smaller than that reported for instance in [5]. Since we simulate much longer (or complete) traces, the cold start effects that contribute toward the compulsory miss component are amortized and are negligible. Furthermore, we simulate a single program at a time and do not account for multiprogramming effects. Multiprogramming tends to increase the number of compulsory misses. Secondly, mapping misses form the largest component of the misses for both instruction and unified traces, especially for the direct mapped caches. Software remapping of basic blocks of the programs was not performed, and is likely to make the mapping component less significant. Finally, the replacement miss component is small for direct mapped caches since there is little flexibility in replacement decisions; the only decision is whether to install or bypass on a cache miss.

7 Conclusion

This paper proposes a method for simulating multiple cache configurations in a single pass under optimal replacement. The method uses limited lookahead, and fixes potential errors later on as more information becomes available. With sufficient lookahead, the execution time of a fully-associative cache simulation with a list implementation is less than that for an LRU fully-associative cache simulation with a list implementation.

Schemes for making fully-associative cache simulation and set-associative cache simulation faster with OPT replacement are also proposed. First, we describe a scheme for partitioning the OPT stack into groups such that at most one entry enters or leaves a group

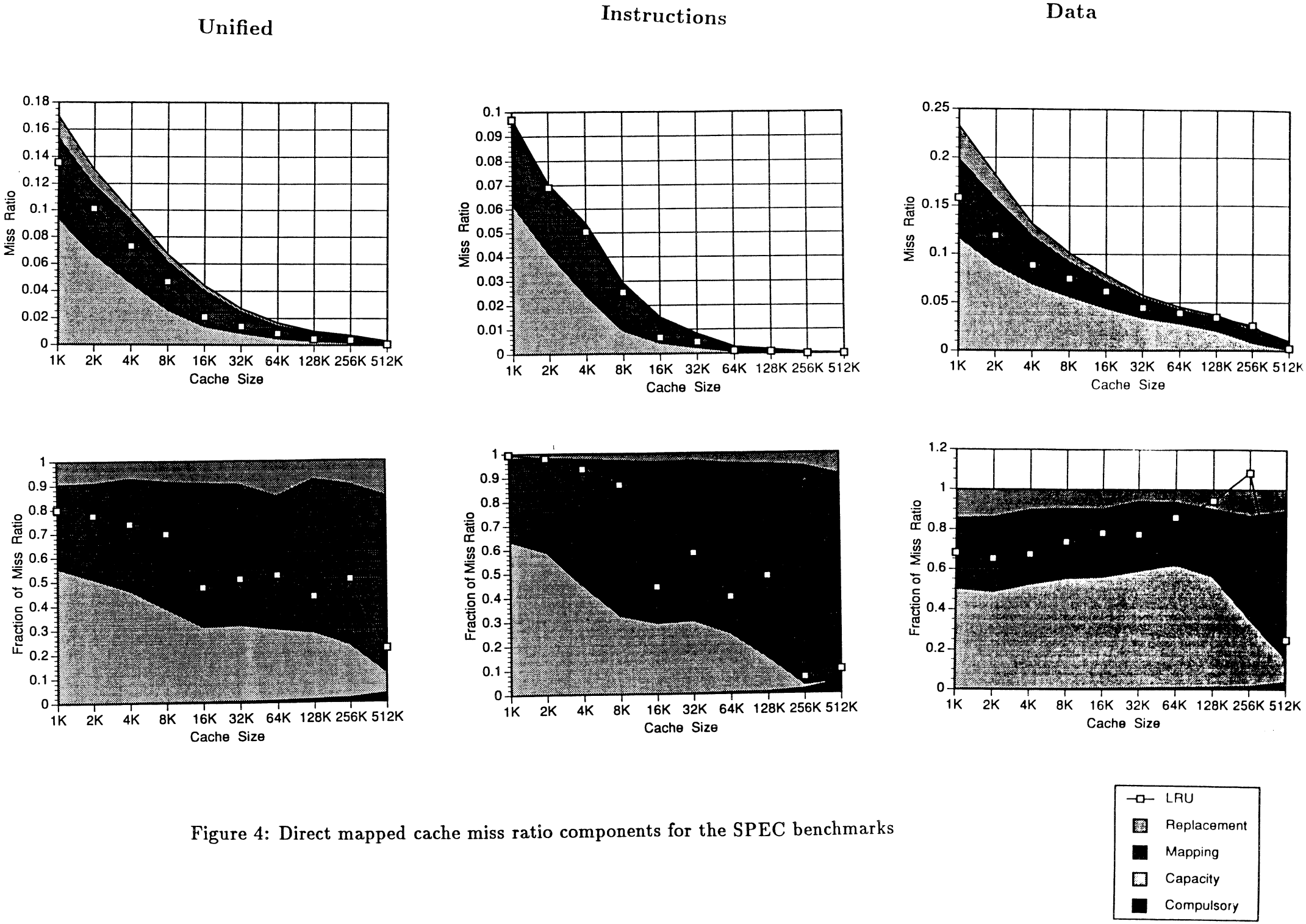

Figure 4: Direct mapped cache miss ratio components for the SPEC benchmarks

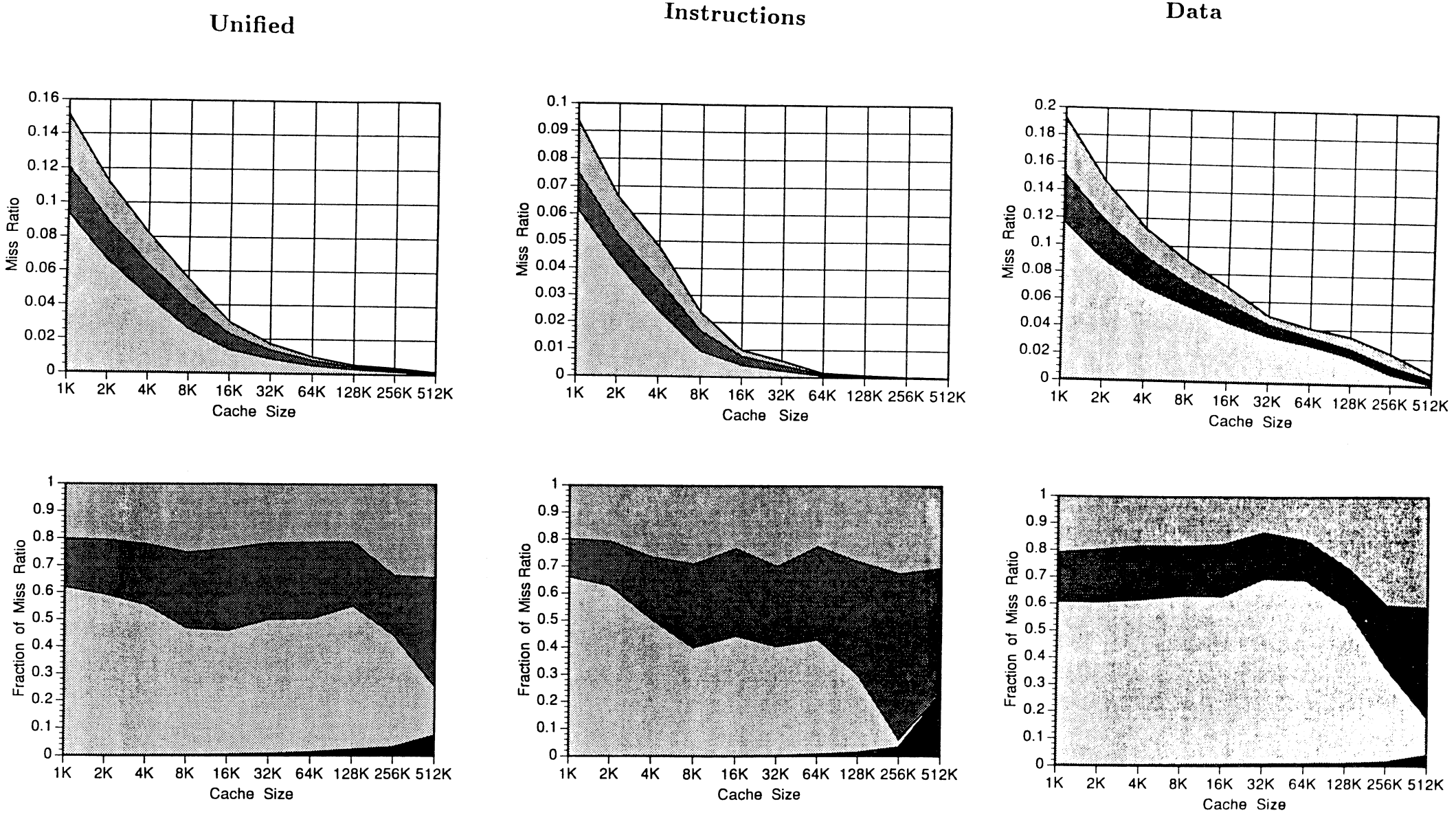

Figure 5: Two way set associative cache miss ratio components for the SPEC benchmarks

during the update procedure. We demonstrate empirically that the mean number of groups passed while updating the stack is low (1 - 4 for the benchmarks run). This grouping scheme enables an efficient tree-based algorithm for fully-associative cache simulation under OPT replacement. The tree-based OPT simulation algorithm is about 2-3 times slower than the tree-based LRU algorithm. Second, we observe that for set-associative cache simulation under OPT replacement, inclusion holds usually but not always across mapping degrees. We describe a scheme which exploits inclusion when it holds, and falls back on a complete simulation when it doesn't.

We propose a new OPT model for characterizing cache misses that extends the 3Cs model. The OPT model categorizes misses into compulsory, capacity, mapping and replacement misses and is a more accurate and finer characterization of cache misses. The OPT model is more accurate because the capacity misses represent a true lower bound on misses in any cache of a given size unlike the 3Cs model. The OPT model further characterizes conflict misses under the 3Cs model into mapping or replacement misses. Cache miss characterization under the OPT model using existing simulation algorithms would be extremely time consuming and in many cases infeasible because of disk space constraints, but using the efficient algorithms developed in this paper the characterization can be performed using reasonable simulation resources.

All the simulation algorithms described in this paper have been implemented, and the implementations were used to simulate traces from four SPEC benchmarks. Miss characterizations for instruction, data, and unified traces using the OPT and 3Cs model are presented. The capacity miss component as estimated by the OPT model is smaller by about 40% on the average. The replacement miss component is about 20% on the average for two-way set-associative caches.

Directions for future work include extending the model to include line size, and performing optimal simulation with writes. Considering multi-programming effects is another direction for future work. A promising approach to doing single-pass simulation with multiprogramming is presented in [9].

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] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM J. of Research and Development*, 5(2):78–101, 1966.

[3] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM J. of Research and Development*, pages 353–357, July 1975.

[4] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *J. of Para. and Dist. Comp.*, 5:587–616, 1988.

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

[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] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proc. of 16th Intl. Symp. on Computer Architecture*, pages 242–251, 1989.

[9] W. W. Hwu and T. M. Conte. The susceptibility of programs to context switching. Technical Report CRHC-91-14, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, April 1991.

[10] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of 17th Intl. Symp. on Computer Architecture*, pages 364–373, 1990.

[11] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *Proc. ACM SIGMETRICS Conf.*, pages 212–213, 1991.

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

[13] S. McFarling. Private commmunication.

[14] S. McFarling. *Program Analysis and Optimization for Machines with Instruction Cache.* PhD thesis, Stanford University, 1988. Technical Report No. CSL-TR-91-493.

[15] S. McFarling. Program optimization for instruction caches. In *Proceedings of ASPLOS III*, 1989.

[16] T. N. Mudge, R. B. Brown, W. P. Birmingham, J. A. Dykstra, A. I. Kayssi, R. J. Lomax, O. A. Olukotun, K. A. Sakallah, and R. A. Milano. The design of a microsupercomputer. *IEEE Computer*, pages 57–64, Jan 1991.

[17] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.

[18] D. D. Sleator and R. E. Tarjan. Self adjusting binary search trees. *J. of the ACM*, 32(3):652–686, 1985.

[19] J. E. Smith and J. R. Goodman. Instruction cache replacement policies and organizations. *IEEE Transactions on Computers*, C-34(3):234–241, March 1985.

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

[21] R. A. Sugumar and S. G. Abraham. Efficient simulation of multiple cache configurations using binomial trees. Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991.

[22] 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.

[23] J. G. Thompson. *Efficient analysis of Caching Systems.* PhD thesis, University of California, Berkeley, 1987.