Static Analysis for Fast and Accurate Design Space Exploration of Caches

Yun Liang, Tulika Mitra
Department of Computer Science
National University of Singapore
{liangyun,tulika}@comp.nus.edu.sg

ABSTRACT

Application-specific system-on-chip platforms create the opportunity to customize the cache configuration for optimal performance with minimal chip estate. Simulation, in particular trace-driven simulation, is widely used to estimate cache hit rates. However, simulation is too slow to be deployed in the design space exploration, specially when it involves hundreds of design points and huge traces or long program execution. In this paper, we propose a novel static analysis technique for rapid and accurate design space exploration of instruction caches. Given the program control flow graph (CFG) annotated only with basic block and control flow edge execution counts, our analysis estimates the hit rates for multiple cache configurations in one pass. We achieve this by modeling the cache states at each node of the CFG in probabilistic manner and exploiting the structural similarities among related cache configurations. Experimental results indicate that our analysis is 24–3,855 times faster compared to the fastest known cache simulator while maintaining high accuracy (0.7% average error), in predicting hit rates for popular embedded benchmarks.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Algorithms, Performance, Design

Keywords

Cache, Design Space Exploration, Probabilistic cache states

1. INTRODUCTION

The fixed functionality nature of embedded systems opens up the opportunity to design a customized system-on-chip (SoC) platform for a particular application or an application domain. The memory subsystem plays a critical role in the design of such customized SoC both in terms of performance and energy consumption. Thus careful tuning of the memory subsystems, in particular the cache parameters, is of paramount importance in meeting the design constraints of a specific embedded application. The cache design parameters include the size of the cache, the line size, the degree of associativity, the replacement policy, and many others. This entire design space has to be explored to identify the cache configuration that optimizes certain objectives, such as performance, energy consumption, or a combination of the two.

The design space exploration of caches is a well studied problem. The exploration process requires cache hit/miss rates for all possible design points. The most popular approach to compute the hit rate for a particular cache configuration is to employ trace-driven simulation or functional simulation. Unfortunately, simulation based approaches are too slow and huge trace sizes put practical limit on both the size of the application and its input. In this paper, we explore static analysis method as an alternative to simulation for fast and accurate estimation of cache hit rates.

Recently, we have introduced the concept of probabilistic cache states [10], which captures the set of possible cache states at a program point along with their probabilities. We have also proposed a static analysis method [10] that models the cache behavior to estimate the expected (average) execution time of a program over all possible program inputs. The notion of probabilistic cache states is quite general. It can be easily adapted to construct a fast and accurate static analysis method that estimates cache hit rate of a program for a particular configuration. Unfortunately, when employed in the context of design space exploration, the runtime of this static cache analysis approach is not competitive compared to state-of-the-art cache simulators such as Cheetah [13]. This is because fast cache simulators employ single-pass simulation that estimates the hit rates for a large number of cache configurations in one pass. In contrast, static cache analysis has to estimate the hit rate for each cache configuration individually leading to overall slower design space exploration.

We observe that if a static analysis approach can model multiple cache configurations in one pass, we get a very powerful tool for design space exploration. In this paper, we extend the concept of probabilistic cache states to achieve this goal. We borrow the data structure, called Generalized Binomial Tree (GBT), proposed by Sugumar and Abraham [13] to exploit the inclusion property among related cache configurations. GBT enables us to capture the cache states corresponding to a number of related configurations in one succinct representation. However, as a program point can be reached from different contexts, we may have a number of GBTs, each associated with the probability of the corresponding context.

In this paper, we propose *probabilistic GBT* to capture the cache states corresponding to all cache configurations and all contexts at any program point. We also define operators for update and concatenation of probabilistic GBTs. These operators are employed

CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.

in our static program analysis to obtain the probabilistic GBTs at every program point in an efficient manner. Given a probabilistic GBT, we can easily estimate the hit rate of a memory access for all possible cache configurations. However, maintaining these *probabilistic GBTs* and operating on them can become space and time inefficient as the number of contexts increases. Therefore, we propose a number of optimizations for space and time efficiency.

In summary, we propose a static analysis method for rapid and accurate design space exploration of instruction caches. Our analysis method can estimate the hit rates for all cache configurations with varying number of sets and associativity in one pass as long as the cache line size remains constant. The input to our analysis is simply the basic block and control flow edge execution count profiles, which is significantly more compact compared to memory address traces required by trace-driven simulators. Our experimental evaluation for a number of embedded benchmarks reveal that our estimation is highly accurate (0.7% average error) and our single-pass static cache analysis is 24–3,855 times faster compared to the fastest known single-pass cache simulator Cheetah.

2. ANALYSIS FRAMEWORK

The inputs to our analysis framework are the executable program code and its corresponding input. We can obtain the basic block and control flow edge counts through execution or quick functional simulation of an instrumented version of the program. The instrumentation can be done very efficiently by using edge profiling [2]. More importantly, the profiling needs to be done only once, as basic block and edge execution counts remain unchanged across different cache configurations.

Our analysis first constructs the loop-procedure hierarchy graph (LPHG) corresponding to the whole program [9]. The LPHG represents the procedure calls and loop nest relations in the program. Loop and procedure bodies are represented as directed acyclic graphs (DAG), where the nodes of a DAG are the basic blocks. If a loop (procedure) contains other loops within its body, then the inner loops are represented as dummy nodes in the DAG. For each loop L, it is annotated with its loop count N_L and its control flow graph is transformed such that every loop has a loop pre-header, post-loop, start, and end node (see Figure 6).

Given a basic block B and an edge $B' \rightarrow B$, we use N_B and $N_{B' \rightarrow B}$ to denote their execution counts, respectively. For control flow edge $B' \rightarrow B$, the edge frequency $f(B' \rightarrow B)$ is defined as the probability that B is reached from B', that is, $f(B' \rightarrow B) = \frac{N_{B' \rightarrow B}}{N_B}$. By definition, $\sum_{e \in In(B)} f(e) = 1$, where $In(B)$ represents all the incoming edges of B.

Cache Hit Rate. Let us use $\mathbf{B}$ to represent the set of the basic blocks of the program and R_{hit} to represent the cache hit rate of the program. Let I_B be the number of instructions and M_B be the set of memory blocks of B. Then, R_{hit} can be computed as

$$R_{hit} = \frac{\sum_{B \in \mathbf{B}} \sum_{m \in M_B} N_B \times H_m}{\sum_{B \in \mathbf{B}} N_B \times I_B} \quad (1)$$

where H_m is the cache hit rate of memory block $m \in M_B$. N_B and I_B are constants across different cache configurations and are available through profiling. However, H_m is unknown and may change across different cache configurations. In the following, we will illustrate how to estimate H_m for all cache configurations through our static cache modeling.

3. CACHE MODELING

We rely on **General Binomial Forest (GBF)** data structure to estimate H_m for multiple cache configurations simultaneously. GBF

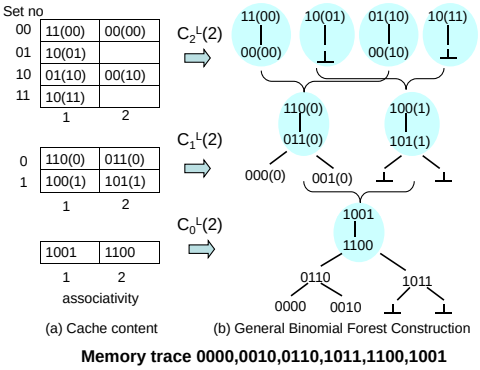

Figure 1: Cache content and construction of generalized binomial forest. Memory blocks are represented by tags and set number, for example, for memory block $11(00)$, 00 denotes the set and 11 is the tag.

was originally proposed for simulating multiple cache configurations in one pass [13]. In this section, we provide a brief background on GBF and then proceed to present our probabilistic GBF. We use the probabilistic GBF for static cache analysis in Section 4.

3.1 GBF Background

Let us explain the GBF data structure with an example. In this paper, we consider LRU as the cache replacement policy. Figure 1(a) shows, for the same memory address trace, the contents of six caches with number of sets = $1, 2, 4$ and associativity = $1, 2$. From the example, we observe that for the caches with the same associativity, the memory blocks in the cache with $2(1)$ sets are included in the cache with $4(2)$ sets. For the caches with the same number of sets, the memory blocks in the cache with associativity 1 are included in the cache with associativity 2.

GBF exploits the aforementioned inclusion property that holds between cache configurations. Let us denote a set-associative cache with 2^S sets, line size L, and associativity N as $C_S^L(N)$. A GBF can represent a set of cache configurations $\{C_S^L(n) | S_{min} \leq S \leq S_{max}; n \leq N\}$, where $2^{S_{min}}$ ($2^{S_{max}}$) is the minimum (maximum) number of sets among the group of cache configurations and N is the maximal associativity.

A GBF consists of one or more **Generalized Binomial Trees (GBT)**. A GBT can be defined recursively as follows. A GBT of degree 0 is a list of length N and the elements in the list are ordered according to LRU policy (i.e., the top element is the most recently accessed address, while the bottom element is the least recently accessed address). A GBT of degree k is constructed by linking two GBTs of degree $k-1$ together, with the most recently accessed N references in either root lists of the two GBTs as the new root list. By definition, a GBT of degree k has $2^k \cdot N$ nodes.

Let us explain the construction of GBF based on the example shown in Figure 1. The GBF for the cache configuration $C_2^L(2)$ consists of 4 GBTs of degree 0 (one corresponding to each set). We use $\perp$ to denote an empty cache block. The GBF for the cache configuration $C_1^L(2)$ contains 2 GBTs of degree 1 (one corresponding to each set). The GBT for a set s in $C_1^L(2)$ is obtained by linking two GBTs of $C_2^L(2)$ that map to the set s. For example, the memory blocks in set 0 and 2 of $C_2^L(2)$ map to set 0 of $C_1^L(2)$. They are merged together with the most recently accessed 2 references as the new root. The merging is done similarly for set 1 in $C_1^L(2)$. This process is continued until the GBF for the cache configuration with the minimum number of sets $C_0^L(2)$ is constructed. Now the contents of all the cache configurations in the set $\{C_S^L(n) | 0 \leq S \leq 2; n \leq 2\}$ can be found in the GBF for the cache configuration $C_0^L(2)$. A detailed description of GBT as well as their search and update procedure can be found in [13].

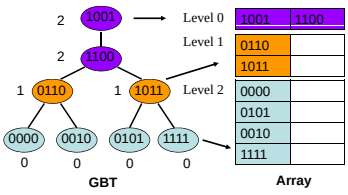

Figure 2: Mapping from GBT to array. The nodes in GBT are annotated with their ranks.

Array Implementation. We use an array based implementation of GBT [13]. Let us assume the degree of GBT as M. The GBT is implemented as a two-dimensional array with $2^{M+1}-1$ rows and N columns. The rows are divided into $M+1$ levels from 0 to M and level k has 2^k rows. As discussed before, a GBT of degree M has $2^M \cdot N$ nodes. Thus, array implementation has about a factor of two redundancy.

Figure 2 shows an example of the array implementation of GBT, where $M = 2$ and $N = 2$. Given a node t in the GBT, we use $des(t)$ to denote the number of descendants (inclusive) of node t. The rank of a node is defined as $log(\lceil \frac{des(t)}{N} \rceil)$. Memory block at a node of rank k maps to level $M-k$ and the row within the level is determined by the least significant $M-k$ bits of the memory block address. There are at most N memory blocks in the same row and they are arranged in the order in which they have been accessed (i.e., the leftmost memory block is the most recently used, while the rightmost memory block is the least recently used).

Given an incoming memory block address $address$, the search and update procedure of GBT starts from the top level and only one row in each level is checked. The row examined in level k is determined by the least significant k bits of $address$ and the tag matches are done with the memory blocks in that row. For example, in Figure 2, suppose we are searching for address 0101. We first examine 1001 and 1100 in level 0. Then, in level 1, the address 0101 maps to row 1 and so 1011 is examined. Finally, in level 2, the address 0101 maps to row 1 and it is found there.

Cache Hits Computation. A two dimension array hit is used for storing the cache hits for multiple cache configurations. Array hit will be updated if a memory block is cache hit, and the corresponding entries will be increased by 1. However, $hit[m][n]$ only stores the number of references that hit in cache configuration $C_m^L(n)$ but miss in smaller caches $C_m^L(n')$ where $n' < n$. According to the inclusion property related to associativity, the number of hits in $C_m^L(n)$ can be computed by summing up the hits of itself and those from smaller caches as $\sum_{i=1}^{n} hit[m][i]$.

3.2 Probabilistic GBT

We now describe the probabilistic cache modeling based on General Binomial Forest (GBF). The multiple cache configurations we support are constant line size, varying number of cache sets and degree of associativity. Based on the description in Section 3.1, we are interested in the set of configurations $\{C_S^L(n) | S_{min} \le S \le S_{max}; n \le N\}$, where $2^{S_{min}}$ ($2^{S_{max}}$) is the minimum (maximum) number of cache sets and N is the maximum associativity.

Assumptions. For the set of cache configurations above, we will have $2^{S_{min}}$ GBTs with degree $S_{max} - S_{min}$ in the GBF. However, one memory block maps to only one GBT based on its index in $C_{S_{min}}^L(N)$. Thus, there is no interference between different GBTs. Thus, we assume $S_{min} = 0$. In other words, there is only one GBT of degree S_{max} in the GBF. For the configurations with more than one GBTs, each GBT can be modeled independently.

More concretely, in the following, we consider a GBT of degree $M(S_{max})$ and root list length as N. To indicate the absence of any memory block in a cache line, we introduce a new element $\perp$. We use Ω to denote the set of all the possible GBTs of the program. We also introduce a special empty GBT $c_\perp$.

At any program point, the GBT is determined by the program path taken before reaching this program point. Usually a program point can be reached via multiple program paths leading to a number of possible GBTs at that point. Thus, we introduce the notion of probabilistic GBT.

DEFINITION 1 (**Probabilistic GBT**). *A probabilistic GBT $\mathcal{C}$ is a 2-tuple: $\langle C, X \rangle$, where $C \in 2^\Omega$ is a set of GBTs and X is a random variable. The sample space of the random variable X is Ω. Given a GBT c, we define $Pr[X = c]$ as the probability of c in C. If $c \notin C$, then $Pr[X = c] = 0$. By definition, $\left(\sum_{c \in \Omega} Pr[X = c]\right) = 1$. Finally, we define a special probabilistic GBT $\mathcal{C}_\perp$ denoting the empty probabilistic GBT. That is $\mathcal{C}_\perp = \langle \{c_\perp\}, X \rangle$, where $Pr[X = c_\perp] = 1$.*

We use $\lhd$ to denote GBT search and update operator. Given a memory block m and a GBT c, $c \lhd m$ returns the GBT after accessing m. Meanwhile, we define new operator $\trianglelefteq$ as the search and update operator of probabilistic GBT. Given a memory block m and a probabilistic GBT $\mathcal{C} = \langle C, X \rangle$, $\trianglelefteq$ will update each GBT $c \in C$ and $\mathcal{C} \trianglelefteq m$ returns the updated probabilistic GBT.

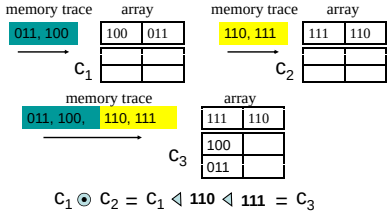

Figure 3: Concatenation for GBTs where $M = 1$ and $N = 2$.

3.2.1 Concatenation of Probabilistic GBTs

In this subsection, we introduce the concatenation of probabilistic GBTs, which will be used later. We first define the operator $\odot$ for the concatenation of two GBTs in Algorithm 1.

Algorithm 1: Implementation of $\odot$ operation

```
input  : GBT c1 and c2
output : c = c1 ⊙ c2
c = c1;
for lev ← M to 0 do
    Let T be the two dimension array at level lev in c2;
    foreach row ∈ T do
        for col ← N to 1 do
            if T[row][col] ≠⊥ then
                c = c ◁ T[row][col] ;
return c;
```

In the array based implementation of GBT, c_2 is a multilevel two-dimensional array. The concatenation is done by using the memory blocks in c_2 from the bottom level to top level and from right to left to update c_1. In other words, the update is done from the least recently used to most recently used memory blocks of c_2. An example of GBT concatenation is shown in Figure 3. Let us assume the GBT after the first and second memory traces are c_1 and c_2, respectively. Then the GBT after accesses corresponding to the two memory traces sequentially is $c_1 \odot c_2$. Next, we extend the concatenation operation to probabilistic GBTs.

DEFINITION 2 (**Concatenation of Probabilistic GBTs**). *Given probabilistic GBTs $\mathcal{C}_1 = \langle C_1, X_1 \rangle$ and $\mathcal{C}_2 = \langle C_2, X_2 \rangle$*

$$\mathcal{C}_1 \bigodot \mathcal{C}_2 = \mathcal{C} \text{ where } \mathcal{C} = \langle C, X \rangle$$
$$C = \{c | c = c_1 \odot c_2, c_1 \in C_1, c_2 \in C_2\}$$
$$Pr[X = c] = \sum_{c_1 \in C_1, c_2 \in C_2, c = c_1 \odot c_2} (Pr[X_1 = c_1] \times Pr[X_2 = c_2])$$

Let us assume the execution of two program fragments sequentially each starting with an empty GBT. The probabilistic GBT after the execution of the first and second program fragments are $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Then the probabilistic GBT after execution of the two program fragments sequentially is $\mathcal{C}_1 \bigodot \mathcal{C}_2$.

3.2.2 Merging GBTs in a Probabilistic GBT

A program path can be specified by the basic block sequence. Although multiple paths could reach a program point, they probably traverse some common basic block subsequence. Thus, the set of GBTs in a probabilistic GBT can include some identical memory blocks. By merging the similar GBTs together, we can reduce the space requirement of probabilistic GBTs. More importantly, the search and update of probabilistic GBTs will be much faster.

In the array based implementation, GBT is divided into $M+1$ levels. We merge the GBTs level by level from top to bottom. More concretely, given two GBTs, if the content of the top k ($k \leq M+1$) levels are identical, then they are merged together to have only one copy of the top k levels as shown in Figure 4(a). Also as the GBTs are merged together, the probabilities are now associated with each level rather than with the GBTs.

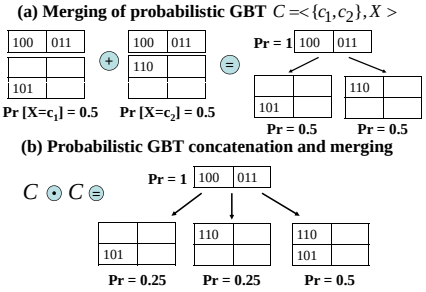

Figure 4: Probabilistic GBT merging and concatenation.

It is possible to perform merging at finer granularity, for example, using rows rather than levels. However, the complexity of the merging process increases considerably leading to slower implementation. It is also possible that two GBTs are different at the top levels, but they are identical at the bottom levels. We choose *not* to perform merging for such GBTs. This is because, as the probabilistic GBT is updated, the contents from the upper levels move to the lower levels. Thus the commonality among the GBTs are lost and they have to be split again. It is far more efficient to merge GBTs only if they are identical at the top levels.

The implementation of a merged GBT can be viewed as a tree with the sub-arrays (levels) of the original GBTs as nodes (see Figure 4(a)). The sub-array corresponding to the common top levels $0-k$ is the root node of this tree. Level k, however, has multiple children at level $k+1$. Now the search and update of probabilistic GBTs become more efficient. Consider a memory block m that is present somewhere in the top k levels. Without merging, m will be searched in all the original GBTs; now it will be searched only once in the merged GBT. For example, in Figure 4(a), before merging, the reference to memory block 100 is searched in both c_1 and c_2. With merged GBT, it is only searched once. In Figure 4(b), we show the merged probabilistic GBT after concatenation operation.

3.2.3 Bounding the size of Probabilistic GBT

We observe that, in a probabilistic GBT, some of the constituent GBTs have very low probabilities. That is, these GBTs correspond to rare program paths. Based on this observation, we prune some of the GBTs for space and time efficiency.

We define the metric $dist$ for pruning. Consider a merged GBT with two nodes at level k. Each node is a two dimension array

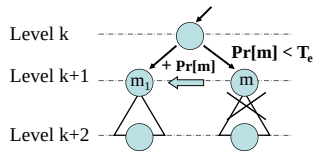

Figure 5: Pruning in probabilistic GBT.

with 2^k rows and N columns. Given two such nodes n_1, n_2 at the same level, we define $d(n_1, n_2)$ as the measure of the distance between them. It is defined as a function of the number of different memory blocks between them. But higher priority is given to the more recently used memory blocks as shown in Equation 2.

$$dist(n_1, n_2) = \sum_{\forall i,j} \begin{cases} N-j+1, & \text{if } n_1[i][j] \neq n_2[i][j] \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

We apply two merging strategies. First, if the probability of a node n is too small ($< T_e$), then the subtree rooted at n is pruned. But its probability is added to the subtree rooted at the closest sibling of n (the closest is defined by the $dist$ metric). Second, if the number of children of a node exceeds a pre-defined limit Z, then Z children with highest probability are kept and the subtrees rooted at the rest of the children are pruned. As before, the probability of each pruned child is added to its closest surviving sibling defined by the $dist$ metric. The pruning process continues from top to bottom. As shown in Figure 5, the subtree rooted at m (including m) is pruned because its probability is too small. However, its probability is added to the subtree rooted at m_1, which is the closest sibling of m. Similar pruning strategy can be applied across independent or merged GBTs in a probabilistic GBT. In practice, we set T_e to 10^{-6} and Z to 4.

3.2.4 Cache Hit Rate of a Memory Block

Recall that in Section 3.1, if a memory block m results in a cache hit, the corresponding entries in the array hit are incremented by 1. However, in our probabilistic cache modeling, we get a cache hit probability by looking up the probabilistic GBT. The hit probability is simply the sum of the probabilities of all the nodes where m can be found in the probabilistic GBT. Now we add this hit probability to the hit array.

For memory block m, we can get its hit rate H_m for different cache configurations if the probabilistic GBT at that program point is known. Then the cache hit rate of the whole program can be derived from Equation 1. Now we present our static analysis method to derive the probabilistic GBTs at every program point.

4. STATIC CACHE ANALYSIS

In this section, we first describe cache analysis for a loop in isolation. Subsequently, we will extend this analysis to the whole program. For loops, we consider its control flow graph as a *directed acyclic graph* (DAG). We first perform the analysis on the DAG for a single iteration, followed by modeling across iterations.

4.1 Analysis of DAG

Let $\mathcal{C}_B^{in}$ and $\mathcal{C}_B^{out}$ be the incoming and outgoing probabilistic GBTs of a basic block B. Similarly, $\mathcal{C}_L^{in}$ and $\mathcal{C}_L^{out}$ denote the incoming and outgoing probabilistic GBTs of a loop L. Let $start$ and end be the unique start and end basic blocks of the DAG corresponding to the loop body. Then $\mathcal{C}_L^{in} = \mathcal{C}_{start}^{in}$ and $\mathcal{C}_L^{out} = \mathcal{C}_{end}^{out}$. As we are analyzing the loop in isolation at this point, $\mathcal{C}_L^{in} = \mathcal{C}_\perp$.

Let $gen_B = \langle m_1, \ldots, m_k \rangle$ be the sequence of memory blocks accessed within a basic block B. Then $\mathcal{C}_B^{out}$ can be computed as

$$\mathcal{C}_B^{out} = \mathcal{C}_B^{in} \trianglelefteq m_1 \trianglelefteq \ldots \trianglelefteq m_k \quad (3)$$

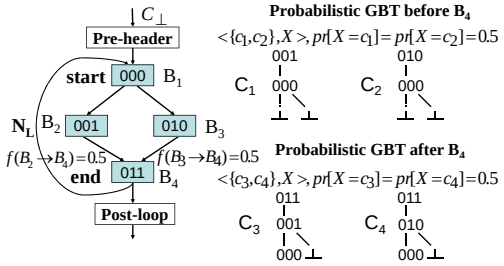

Figure 6: A loop with pre-header and post-loop node. Probabilistic GBT before/after B_4 are shown for the first iteration of the loop. For each basic block, its memory blocks are displayed.

The incoming probabilistic GBT of B is obtained from the outgoing probabilistic GBTs of its predecessors. We rely on following new operator to do the combination.

DEFINITION 3 (**Probabilistic GBTs Combination**). *We define $\bigoplus$ as the combination operator for probabilistic GBTs. It takes in n probabilistic GBTs $\mathcal{C}_i = \langle C_i, X_i \rangle$ and a corresponding weight function w as input s.t. $\sum_{i=1}^{n} w(\mathcal{C}_i) = 1$. It produces a combined probabilistic GBT $\mathcal{C}$ as follows.*

$$\bigoplus(\mathcal{C}_1, \ldots, \mathcal{C}_n, w) = \mathcal{C} \text{ where } \mathcal{C} = \langle C, X \rangle, C = \bigcup_{i=1}^{n} C_i,$$
$$Pr[X = c | c \in C] = \sum_{\forall i, c \in C_i} Pr[X_i = c] \times w(\mathcal{C}_i)$$

In other words, the set of GBTs in C is the union of all the GBTs in $C_1, \ldots, C_n$. The probability of a GBT $c \in C$ is a weighted summation of the probabilities of c in the input probabilistic GBTs. Let $in(B) = \{B', B'', \ldots\}$ be the set of predecessors of B. Then the incoming probabilistic GBT of B can be derived as

$$\mathcal{C}_B^{in} = \bigoplus(\mathcal{C}_{B'}^{out}, \mathcal{C}_{B''}^{out}, \ldots, w) \tag{4}$$

where the weight function w is defined as $w(\mathcal{C}_{B'}^{out}) = f(B' \rightarrow B)$. Starting with $\mathcal{C}_\perp$, Figure 6 shows an example of probabilistic GBT combination at basic block B_4 and the probabilistic GBT after B_4 in the first iteration of the loop, where $M = 1$ and $N = 2$.

4.2 Extension to Loop Iterations

In the previous subsection, we assume $\mathcal{C}_L^{in} = \mathcal{C}_\perp$. However, for a loop iterating multiple times, the input GBT at the $start$ node of the loop body is different for each iteration. More concretely, let us add the subscript $\langle n \rangle$ for the n^{th} iteration of the loop. Then $\mathcal{C}_{start\langle n \rangle}^{in} = \mathcal{C}_{end\langle n-1 \rangle}^{out}$ for $n > 1$. However, in order to compute $\mathcal{C}_{start\langle 1 \rangle}^{in}, \ldots, \mathcal{C}_{start\langle N \rangle}^{in}$, where $N = N_L$ is the loop count, we *do not* need to traverse the DAG N times. Instead, we can rely on the $\bigodot$ operator. First, we note that $\mathcal{C}_{start\langle 1 \rangle}^{in} = \mathcal{C}_L^{in} = \mathcal{C}_\perp$. Then for iteration $n > 1$

$$\begin{array}{ll} \mathcal{C}_{start\langle n \rangle}^{in} & = \mathcal{C}_{end\langle n-1 \rangle}^{out} \\ \mathcal{C}_{end\langle n \rangle}^{out} & = \mathcal{C}_{start\langle n \rangle}^{in} \bigodot \mathcal{C}_{end\langle 1 \rangle}^{out} \end{array} \tag{5}$$

The final probabilistic GBT after N iterations starting with $\mathcal{C}_L^{in} = \mathcal{C}_\perp$, is denoted as $\mathcal{C}_L^{gen}$ where

$$\mathcal{C}_L^{gen} = \mathcal{C}_{end\langle N \rangle}^{out} \tag{6}$$

The cache hit rate of a memory block is dependent on the input probabilistic GBT $\mathcal{C}_B^{in}$ of the corresponding basic block B, which in turn is dependent on $\mathcal{C}_{start\langle n \rangle}^{in}$ of the loop L. Computing the cache hit rate for each memory block in each iteration is equivalent to complete loop unrolling. Instead, we observe that we only need to compute an "average" probabilistic GBT $\mathcal{C}_L^{avg}$ at the $start$ node of the loop body. This captures the input GBT of the loop over N iterations. That is, $\mathcal{C}_L^{avg}$ is defined as

$$\mathcal{C}_L^{avg} = \oplus(\mathcal{C}_{start\langle 1 \rangle}^{in}, \ldots, \mathcal{C}_{start\langle N \rangle}^{in}, w) \tag{7}$$

where $w(\mathcal{C}_{start\langle n \rangle}^{in}) = \frac{1}{N}$. Now, in Section 4.1, we simply replace $\mathcal{C}_{start}^{in} = \mathcal{C}_\perp$ with $\mathcal{C}_{start}^{in} = \mathcal{C}_L^{avg}$. The rest of the analysis for the DAG remains unchanged.

More importantly, the operator $\bigodot$ need not be invoked N_L times as the probabilistic GBTs across iterations may converge. After convergence point, the size and content of the probabilistic GBT as well as the probability of each GBT in the probabilistic GBT do not change. In practice, we relax the convergence constraint. If the difference of probabilities between every pair of identical GBTs in $\mathcal{C}_{end\langle n \rangle}^{out}$ and $\mathcal{C}_{end\langle n+1 \rangle}^{out}$ are within T_e, we declare convergence. Experimental results confirm that convergence is reached quickly for most of the loops in all the benchmark programs. In the worst case, concatenation operations is terminated at a pre-defined threshold of $MaxN$ iterations. The average probabilistic GBT across these $MaxN$ iterations is used as an approximation of the average probabilistic GBT across N_L iterations. In practice, we set $MaxN$ to 100 and T_e to 10^{-6}.

4.3 Analysis of Whole Program

We first traverse the LPHG in bottom-up fashion, i.e., we start with the innermost loops/procedures and compute $\mathcal{C}_L^{gen}$ and $\mathcal{C}_L^{avg}$ for all such loops/procedures. Next, we replace the innermost loops or procedures with "dummy" nodes in the DAG of the enclosing loop or procedure. While traversing the DAG of the enclosing loop or procedure, special care is taken for the dummy nodes. Let $\mathcal{C}_L^{in}$ be the input GBT for dummy node L during traversal of the DAG. Then we treat the dummy node as a black box and compute the output GBT of the dummy node as $\mathcal{C}_L^{out} = \mathcal{C}_L^{in} \bigodot \mathcal{C}_L^{gen}$. At the end of this bottom-up traversal process, we reach the root node (`main` procedure). Then, we perform a top-down traversal to compute the probabilistic GBT at each basic block in the context of the whole program. Suppose L is a dummy node during this top-down traversal with input probabilistic GBT $\mathcal{C}_L^{in}$ and start node $start$. Then we traverse the DAG of L with $\mathcal{C}_{start}^{in} = \mathcal{C}_L^{in} \bigodot \mathcal{C}_L^{avg}$ and compute the probabilistic GBT at each node of the DAG. This top-down process continues till we traverse all the loops/procedures. At this point, we have computed the "average" probabilistic GBT for each basic block in the context of the whole program. Now the cache hit rate for each memory block across multiple cache configurations can be computed.

5. EXPERIMENTAL RESULTS

We evaluate the accuracy and efficiency of our static cache analysis by comparing it with cache simulator Cheetah [13]. Cheetah is the fastest known cache simulator, which can simulate multiple cache configurations in a single pass.

We select 10 programs from MiBench [5]. We fix a line size for each benchmark, but vary the number of cache sets from 4 to 64 and associativity from 1 to 8. That is, a total of 20 cache configurations are estimated and simulated. The line size for each benchmark is selected such that the cache hit rate has a wide coverage. The benchmarks, corresponding line size, and trace size are shown in Table 1. For trace-driven simulation, trace size can be quite large even for small programs as shown in column *Trace*. We use SimpleScalar toolset [1] for the experiments. We instrument its functional simulator to collect execution count of basic blocks and control flow edges. The time spent in our instrumentation during the functional simulation is shown in column *Prof*. Our estimator first disassembles the executable to construct CFG and LPHG,

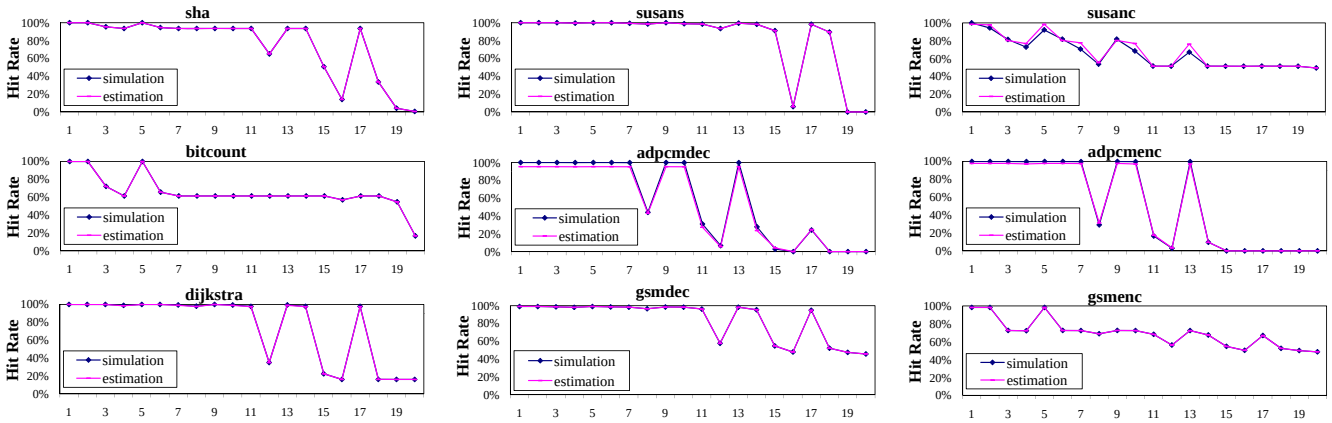

Figure 7: Comparison of cache hit rate for cheetah (simulation) and our method (estimation) across 20 cache configurations (horizontal axis).

Benchmark	Line (Byte)	Trace (MB)	Time(sec)			
			Prof	Cheetah	Analysis	Ratio
bitcount	8	3583	17.04	138.8	0.036	3855.56
dijkstra	8	4700	9.05	143.87	0.298	482.79
adpcmdec	8	791	3.22	33.055	0.086	384.36
adpcmenc	8	961	4.10	41.321	0.197	209.75
sha	8	706	0.69	21.524	0.063	341.65
rijndael	32	1600	0.99	32.827	0.065	505.03
susans	8	4206	5.70	118.9	0.268	443.66
susanc	16	896	0.25	28.234	0.577	48.93
gsmenc	16	2089	2.01	67.65	1.777	38.07
gsmdec	16	1800	8.29	35.26	1.462	24.12

Table 1: Runtime comparison of Cheetah simulator and our analysis.

and then proceeds with the cache hit estimation. We perform all experiments on a 3GHz Pentium 4 CPU with 2GB memory.

The estimation and simulation times are shown in Table 1. Our static analysis method is significantly faster (24–3,855 X speedup) compared to Cheetah simulation. To compare accuracy, for each benchmark, we show the cache hit rates of both simulation and estimation across all the 20 configurations in Figure 7. The estimation for `rijndael` is identical to simulation for all configurations; so it is not shown. The estimation results from analysis track the simulation results quite closely. For all the benchmarks and cache configurations, we achieve high accuracy (0.7% average error). The error is defined as $|est - sim|$ where $est(sim)$ is the estimated (simulated) cache hit rate.

6. RELATED WORK

Trace-driven simulation is widely used for evaluating cache design parameters. A. Janapsatya et al. [7] propose an instruction cache simulation methodology that can operate directly on a compressed program trace file. Simulating reduced traces obtained by statistical sampling is proposed in [8]. In addition, lossless techniques for trace reduction are studied in [14, 15]. Inclusion property is exploited to remove certain references from the trace prior to simulation [14]. By simulating the cache configurations in a particular order, some redundant information can be stripped off from the trace after each simulation [15]. Single pass simulation is proposed in [13, 6, 11]. They are based on the inclusion property which states that the content of a smaller cache is included in a bigger cache for certain replacement policy. Various data structures, such as single stack [11], forest [6], and generalized binomial tree [13], have been proposed for utilizing the inclusion property.

Given an address trace, [3, 12] propose probability based analytical models to compute cache hit ratio. But their approaches are either for only direct mapped caches or fully associative caches. In contrast, our method works on the program control flow graph and does not require address traces. We also predict hit rates for multiple configurations in a single pass. Ghosh and Givargis [4] propose an analytical approach for design space exploration that can directly compute cache parameters satisfying the desired performance.

7. CONCLUSION

In this paper, we present a fast and accurate design space exploration technique for instruction caches via static analysis. We introduce probabilistic Generalized Binomial Tree (GBT) to represent the cache contents for multiple paths and configurations, define operations on the probabilistic GBT, and discuss optimization to improve their space and time efficiency. Finally, we show how to derive these probabilistic GBTs at any point in the program. The experimental results indicate that our method achieves significant speedup compared to simulation while maintaining high accuracy.

8. ACKNOWLEDGMENTS

This work was supported by NUS project R-252-000-292-112.

9. REFERENCES

[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[2] T. Ball. Efficiently counting program events with support for on-line queries. *ACM Transactions on Programming Languages and Systems*, 16(5),1994.

[3] E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *ISPASS*, 2004.

[4] A. Ghosh and T. Givargis. Analytical design space exploration of caches for embedded systems. In *DATE*, 2003.

[5] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *Workshop on Workload Characterization*, 2001.

[6] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12), 1989.

[7] A. Janapsatya et al. Instruction trace compression for rapid instruction cache simulation. In *DATE*, 2007.

[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*, 37(11), 1988.

[9] Y. Li et al. Hardware-software co-design of embedded reconfigurable architectures. In *DAC*, 2000.

[10] Y. Liang and T. Mitra. Cache modeling in probabilistic execution time analysis. In *DAC*, 2008.

[11] R. L. Mattson et al. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

[12] G. Rajaram and V. Rajaraman. A probabilistic method for calculating hit ratios in direct mapped caches. *Journal of Network and Computer Applications*, 19(3), 1996.

[13] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems*, 13(1), 1995.

[14] W. Wang and J. Baer. Efficient trace-driven simulation method for cache performance analysis. In *SIGMETRICS*, 1990.

[15] Z. Wu and W. Wolf. Iterative cache simulation of embedded CPUs with trace stripping. In *CODES*, 1999.