An Analytical Approach for Fast and Accurate Design Space Exploration of Instruction Caches

Yun Liang, Advanced Digital Sciences Center, Illinois at Singapore
Tulika Mitra, School of Computing, National University of Singapore

Application-specific system-on-chip platforms create the opportunity to customize the cache configuration for optimal performance with minimal chip area. Simulation, in particular trace-driven simulation, is widely used to estimate cache hit rates. However, simulation is too slow to be deployed in design space exploration, especially when there are hundreds of design points and the traces are huge. In this paper, we propose a novel analytical approach for design space exploration of instruction caches. Given the program control flow graph (CFG) annotated only with basic block and control flow edge execution counts, we first model the cache states at each point of the CFG in a probabilistic manner. Then, we exploit the structural similarities among related cache configurations to estimate the cache hit rates for multiple cache configurations in one pass. Experimental results indicate that our analysis is 28 — 2,500 times faster compared to the fastest known cache simulator while maintaining high accuracy (0.2% average error) in estimating cache hit rates for a large set of popular benchmarks. Moreover, compared to a state-of-the-art cache design space exploration technique, our approach achieves 304 — 8086 times speedup and saves up to 62% energy and on average 7% for the evaluated benchmarks.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems.

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Cache, Design Space Exploration

ACM Reference Format:
Liang, Y., Mitra, Tulika. 2012. An Analytical Approach for Fast and Accurate Design Space Exploration of Instruction Caches. ACM Trans. Embedd. Comput. Syst. V, N, Article A (January YYYY), 26 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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. Cache memory subsystem bears significant importance in embedded system design as it bridges the performance gap between the fast processor and the slow main memory. Generally, for a well-tuned and optimized memory hierarchy, most of the memory accesses can be fetched directly from the cache instead of the main memory, which requires more power consumption and longer delay per access. In this work, we focus on instruction cache, which is present in almost all embedded systems. Instruction cache is one of the foremost power consuming and performance determining microarchitectural feature of modern embedded systems as instructions are fetched almost every clock cycle. For example, instruction fetch consumes 22% of the power in the Intel Pentium Pro processor [Brooks et al. 2000]. 27% of the total power is spent by instruction cache for StrongARM 110 processor [Montanaro et al. 1997]. Thus, careful tuning and optimization of instruction cache memory can lead to significant performance gain and energy saving.

Author's addresses: Y. Liang, Advanced Digital Sciences Center 1 Fusionopolis Way, #08-10 Connexis North Tower Singapore 138632; T. Mitra, Department of Computer Science, School of Computing, Computing 1, 13 Computing Drive, Singapore 117417 National University of Singapore.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Application specific processor designs (such as Tensilica's Xtensa processor [Tensilica, Xtensa processor]) and soft-core processors in the FPGAs [Nios processor ; Microblaze processor] allow the embedded system designer to customize the cache configuration for a particular application. Choosing a suitable cache configuration for the application helps to improve the cache hit rate and thus performance and energy. However, cache design parameters include the size of the cache, the line size, the degree of associativity, the replacement policy, and many others. Hence, cache design space consists of a large number of design points. The most popular approach to explore the cache design space is to employ trace-driven simulation or functional simulation [Uhlig and Mudge 1997; Zhang et al. 2003]. Although the cache hit/miss rate results are accurate, simulation could be very slow, typically much longer than the execution time of the program. Moreover, the address trace tends to be large even for a small program. Thus, huge trace sizes put practical limit on the size of the application and its input. In this article, we explore analytical approach as an alternative to simulation for fast and accurate estimation of cache hit rates.

We first introduce the concept of probabilistic cache states, which captures the set of possible cache states at a program point along with their probabilities. We also define operators for update and concatenation of probabilistic cache states. Then, we propose a static program analysis technique that computes the probabilistic cache states at each point of the program control flow graph (CFG), given the program branch probability and loop bound information. The static analysis makes effective use of the update and concatenation operators. With the computed probabilistic cache states, we are able to derive the cache hit rate for each memory reference in the CFG and the cache hit rate for the entire program. More importantly, there exist structural similarities among related cache configurations. Based on this observation, we extend our analytical approach to model multiple cache configurations in one pass. To achieve this goal, we borrow the data structure, called Generalized Binomial Tree (GBT), proposed by Sugumar and Abraham [Sugumar and Abraham 1995] 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. Thus, we propose probabilistic GBT to capture the cache states corresponding to all cache configurations and all contexts at any program point. Cache state operators such as update and concatenation are extended for probabilistic GBT. Now, given these probabilistic GBT, we can easily estimate the cache hit rate of the entire program for multiple 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 an analytical method for fast and accurate design space exploration of instruction caches. Our analysis method can estimate the cache hit rates for multiple cache configurations in one pass. The input to our analysis is simply the basic block and control flow edge count profiles, which is significantly more compact compared to memory address traces required by trace-driven simulators and other trace based analytical works. Our experimental evaluation with a set of popular embedded and SPEC benchmarks reveals that our estimation is highly accurate (0.2% average error) and our single pass cache analysis is 28 — 2,500 times faster compared to the fastest known single pass cache simulator *Cheetah*. We also extend our approach for exploring design tradeoffs between performance and energy. Compared to a state-of-the-art [Zhang and Vahid 2003] cache design space exploration technique, our approach achieves 304 — 8086 times speedup and saves up to 62% energy and on average 7% for the evaluated benchmarks.

The rest of this article is organized as follows: In section 2, we summarize the related work. Our analysis framework is outlined in section 3. We present our analytical approach for a single and multiple cache configurations in section 4 and 5, respectively. Finally, we present the experimental results in section 6 and conclude the paper in section 7.

2. RELATED WORK

Cache memory plays a critical role for embedded systems design in terms of both performance and energy consumption. Various methods targeting improvement of cache performance have been proposed such as efficient cache design space exploration [Liang and Mitra 2008a; 2008b; Ghosh and Givargis 2004; Sugumar and Abraham 1995], instruction cache locking [Liang and Mitra 2010a] and code reorganization [Guillon et al. 2004; Liang and Mitra 2010b]. In this paper, we focus on cache design space exploration, which computes the cache hits/misses for various cache configurations. To explore cache design spaces, we can rely on detailed trace driven simulation, analytical modeling, or hybrid approach using both simulation and analytical modeling.

2.1. Trace Driven Simulation

Trace-driven simulation is widely used for evaluating cache design parameters [Uhlig and Mudge 1997]. The collected application trace is fed to the cache simulator which mimics the behavior of some hypothetical cache configurations and outputs the cache performance metrics such as cache hit/miss rate. However, complete trace simulation could be very slow and sometimes is not necessary. Hence, lossless trace reduction techniques have been described in [Wang and Baer 1991; Wu and Wolf 1999]. Wang and Baer observed that the references that hit in small direct mapped cache will hit in larger caches. They exploited the observation to remove certain references from the trace before simulation [Wang and Baer 1991]. In [Wu and Wolf 1999], cache configurations are simulated in a particular order in order to strip off some redundant information from the trace after each simulation. However, both of these techniques still need multiple passes of simulation. Single pass simulation techniques have been proposed in [Sugumar and Abraham 1995; Hill and Smith 1989; Mattson et al. 1970]. Based on the inclusion property that roughly states that the content of a smaller cache is included in a bigger cache for certain replacement policy, multiple cache configurations can be evaluated simultaneously during a single pass. Various data structures, such as single stack [Mattson et al. 1970], forest [Hill and Smith 1989], and generalized binomial tree [Sugumar and Abraham 1995], have been proposed for utilizing the inclusion property. Cheetah [Sugumar and Abraham 1995] is shown to be the most efficient single pass simulator so far. However, address traces could be very big even for a small program and they have to be compressed for practical usage. Simulation methodology that operates directly on a compressed trace have been presented in [Li et al. 2004]. Recently, Mohammad et al. proposed a single pass simulation framework [Haque et al. 2009]. However, it is not clear how fast their technique is when compared to *Cheetah* and address trace is still needed by their technique. In contrast, our solution is shown to be much faster than *Cheetah* and does not need address trace.

2.2. Other Approaches

Analytical approach has been proposed as an alternative to trace-driven simulation for efficient cache design space exploration. Ghosh and Givargis [Ghosh and Givargis 2004] proposed an efficient analytical approach for design space exploration of caches. Given the application trace and desired performance constraint, the analytical model generates the set of cache configurations that meet the performance constraints directly. However, for realistic cache design parameters (limited associativity), the proposed analytical model is as slow as trace simulation.

Hybrid approaches are used to explore cache design space, where heuristics are used to prune the design space and simulations are employed to obtain the cache hits/misses for selected cache configurations. Zhang and Vahid proposed heuristics to find the cache configurations with the best energy consumption, performance and pareto-optimal points with different energy and performance tradeoffs [Zhang and Vahid 2003]. However, hybrid approaches may fail to find the global optimal solution and they are still slow because quite a number of simulations are required.

Gordon-Ross et al. presented a one-shot configurable-cache tuner for hardware implementation [Gordon-Ross et al. 2007]. The hardware module non-intrusively collects the data access, predicts the best cache configuration, and configures the cache dynamically. In comparison, our

technique does not need special hardware support; it only needs the basic block and control flow edge count profiles as inputs which can easily collected through profiling. Arnold et al. described a static cache analysis technique for estimating the Worst Case Execution Time (WCET) for hard real-time systems [Arnold et al. 1994]. Their static program analysis relies on loop level analysis and abstract cache state. In contrast, our technique aims to improve the average-case performance of general embedded applications. Thus, we model the cache state in a probabilistic manner. Our static cache analysis is based on efficient cache update and concatenation operators.

3. ANALYSIS FRAMEWORK

The analysis framework is shown in Figure 1. The inputs to our analysis framework are the executable program code and its corresponding data set. 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 using edge profiling [Ball 1994]. 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 constructs the loop-procedure hierarchy graph (LPHG) corresponding to the whole program [Li et al. 2000]. 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 loops 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. Then, our analysis proceeds with the singe pass cache design space exploration.

Given a basic block B and an edge $B_i \rightarrow B_j$, we use N_B and $N_{B_i \rightarrow B_j}$ to denote their execution counts, respectively. For control flow edge $B_i \rightarrow B_j$, the edge frequency $f(B_i \rightarrow B_j)$ is defined as the probability that B_i is reached from B_j, that is, $f(B_i \rightarrow B_j) = \frac{N_{B_i \rightarrow B_j}}{N_{B_j}}$[1]. By definition, $\sum_{e \in In(B)} f(e) = 1$, where $In(B)$ represents all the incoming edges of B. Figure 1 shows an example of annotated control flow graph and all basic blocks and control flow edges are annotated with their execution information.

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

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

where H_m is the cache hit rate of the m_{th} memory block access $\in M_B$.

More clearly, in Equation 1, the numerator represents the estimated number of cache hits and the denominator represents the number of dynamically executed instructions. N_B (the execution counts of basic block B) and I_B (the number of instructions of basic block 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 through our cache modeling technique.

4. CACHE MODELING FOR A SINGLE CONFIGURATION

In this section, we describe our cache analytical cache modeling approach for single configuration and we extend it for multiple cache configurations in the next section.

Cache Terminology. A cache memory is defined in terms of four major parameters: *block or line size* L, *number of sets* K, *associativity* A, and *replacement policy*. The block or line size determines

[1] Frequency of edge (loop pre-header to loop start) is 1.

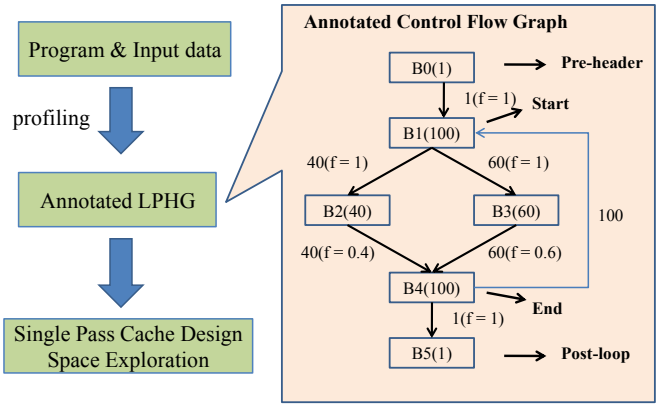


Fig. 1. Our analysis framework. In the annotated LPHG, each basic block is annotated with its execution count and each edge is associated with its execution count and frequency (probability). For example, the execution count of basic block $B2$ is 40 and the execution count of edge $B2 \rightarrow B4$ is 40 too. Probability of edge $B2 \rightarrow B4$ is 0.4.

the unit of transfer between the main memory and the cache. A cache is divided into K sets. Each cache set, in turn, is divided into A cache blocks, where A is the associativity of the cache. For a direct-mapped cache $A = 1$, for a set-associative cache $A > 1$, and for a fully associative cache $K = 1$. In other words, a direct-mapped cache has only one cache block per set, whereas a fully-associative cache has only one cache set. Now the cache size is define as $(K \times A \times L)$. For a set-associative cache, the replacement policy (e.g., LRU, FIFO, etc.) defines the block to be evicted when a cache set is full.

Assumptions. Without loss of generality, we will limit our discussion to a fully associative cache. A set-associative cache with associativity A can be easily modeled by modeling each cache set as a fully associative cache containing A blocks. Let M_i denote the set of all the memory blocks that can map to the i^{th} cache set. Given a memory block m, it is mapped to only one cache set given by $(m \ \ modulo \ \ K)$. Thus, $\bigcap_{i=0}^{K-1} M_i = \emptyset$. In other words, there is no interference among the cache sets and they can be modeled independently. More concretely, in the following, we consider a fully-associative cache with A cache blocks and the program store as a set of memory blocks M. To indicate the absence of any memory block in a cache line, we introduce a new element $\perp$. In this work, we consider LRU (least recently used) replacement policy, where the block replaced is the one that has been unused for the longest time.

4.1. Concrete Cache States

Let us first formally define the concrete cache states and their corresponding operations. These definitions will be used later to introduce the notion of probabilistic cache states.

DEFINITION 1 (**Concrete Cache States**). *A concrete cache state c is a vector $\langle c[1], \ldots, c[A] \rangle$ of length A where $c[j] \in M \cup \{\perp\}$. If $c[j] = m$, then m is the j^{th} most recently used memory block in the cache. Ω denotes the set of all possible concrete cache states. We also define a special concrete cache state $c_\perp = \langle \perp, \ldots, \perp \rangle$ called the empty cache state. Figure 2 shows some of the concrete cache states corresponding to the control flow graph.*

DEFINITION 2 (**Cache Hit**). *Given a concrete cache state $c \in \Omega$ and a memory access $m \in M$*

$$hit(c, m) = \begin{cases} 1 & \text{if } \exists j \ (1 \leq j \leq A) \ \text{ s.t. } c[j] = m \\ 0 & \text{otherwise} \end{cases}$$

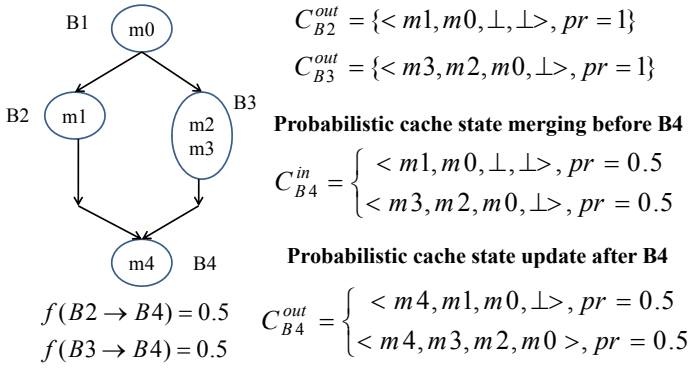


Fig. 2. Control flow graph consists of two paths with equal probability (0.5). The illustration is for a fully-associative cache with 4 blocks starting with empty cache state. $m0$–$m4$ are the memory blocks. Two probabilistic cache states before B_4 are shown. The probabilistic cache states merging and update operation are shown for B_4.

DEFINITION 3 (**Concrete Cache State Update**). *We define $\lhd$ as concrete cache state update operator. Given a concrete cache state $c \in \Omega$ and a memory block $m \in M \cup \{\perp\}$, $c \lhd m$ defines the cache state after memory access m following LRU policy.*

$$c \lhd m = \begin{cases} c, & \text{if } m = \perp \\ c', & \text{where } c'[1] = m; \\ & c'[j] = c[j-1], 1 < j \leq k \\ & c'[j] = c[j], k < j \leq A \qquad \text{if } \exists k \text{ s.t. } c[k] = m \\ c', & \text{where } c'[1] = m; \\ & c'[j] = c[j-1], 1 < j \leq A \quad \text{otherwise} \end{cases}$$

4.2. Probabilistic Cache States

At any program point, the concrete cache state is dependent on the program path taken before reaching this program point. In general, a program point can be reached through multiple program paths leading to a number of possible cache states at that point. We have to model the probability of each of these cache states. For this purpose, we introduce the notion of probabilistic cache states.

DEFINITION 4 (**Probabilistic Cache States**). *A probabilistic cache state $\mathcal{C}$ is a 2-tuple: $\langle C, X \rangle$, where $C \in 2^{\Omega}$ is a set of concrete cache states and X is a random variable. The sample space of the random variable X is the set of all possible concrete cache states Ω. Given a concrete cache state c, we define $Pr[X = c]$ as the probability of the cache state 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 cache state $\mathcal{C}_{\perp}$ denoting the empty cache state. That is $\mathcal{C}_{\perp} = \langle \{c_{\perp}\}, X \rangle$, where $Pr[X = c_{\perp}] = 1$.*

DEFINITION 5 (**Cache Hit/Miss Probability**). *Given a probabilistic cache state $\mathcal{C} = \langle C, X \rangle$ and a memory block m, the cache hit probability $PHit(\mathcal{C}, m)$ of memory access m is*

$$PHit(\mathcal{C}, m) = \sum_{c \in C, hit(c,m)=1} Pr[X = c]$$

In other words, we add up the probability of all the concrete cache states $c \in C$ that contain the memory block m. The cache miss probability can now be defined as

$$PMiss(\mathcal{C}, m) = 1 - PHit(\mathcal{C}, m)$$

DEFINITION 6 (**Probabilistic Cache State Update**). *We define $\trianglelefteq$ as the probabilistic cache state update operator. Given a probabilistic cache state $\mathcal{C} = \langle C, X \rangle$ and an access to memory*

block $m \in M$, $\mathcal{C} \trianglelefteq m$ defines the updated probabilistic cache state.

$$\mathcal{C} \trianglelefteq m = \mathcal{C}' \text{ where } \mathcal{C}' = \langle C', X' \rangle$$
$$C' = \{c \triangleleft m | c \in C\}$$
$$Pr[X' = c' | c' \in C'] = \sum_{c \in C, c' = c \triangleleft m} Pr[X = c]$$

For example, in Figure 2, the probabilistic cache state at the end of basic block B_4 (starting with empty cache state) consists of two concrete cache states with equal probability 0.5. The cache miss probability of memory blocks $m1$–$m3$ in this probabilistic cache state is 0.5 whereas the miss probability of $m0$ and $m4$ are 0.

4.3. Static Cache Analysis

In this subsection, we first describe cache analysis for a loop in isolation, i.e., we assume an empty cache state at the loop entry point. Subsequently, we will extend this analysis to the whole program. We consider the control flow graph (CFG) to be a *directed acyclic graph* (DAG), representing the body of the loop. We first perform the analysis on the DAG to model cache behavior for a single iteration of a loop. This will be followed by probabilistic cache state modeling across iterations.

4.3.1. Analysis of DAG. Let $\mathcal{C}_B^{in}$ and $\mathcal{C}_B^{out}$ be the incoming and outgoing probabilistic cache states of a basic block B. Similarly, $\mathcal{C}_L^{in}$ and $\mathcal{C}_L^{out}$ denote the incoming and outgoing probabilistic cache states 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, $\mathcal{C}_L^{in} = \mathcal{C}_\perp$. We relax this constraint in the next section.

Let $gen_B = \langle m_1, \ldots, m_k \rangle$ be the sequence of memory blocks accessed within a basic block B. Then

$$\mathcal{C}_B^{out} = \mathcal{C}_B^{in} \trianglelefteq m_1 \trianglelefteq \ldots \trianglelefteq m_k \tag{2}$$

That is, the outgoing probabilistic cache state of a basic block can be derived by repeatedly updating the incoming probabilistic cache state with the memory accesses in B. Now in order to generate the incoming cache state of B from its predecessor cache states, we need to define the following new operator.

DEFINITION 7 (**Probabilistic Cache States Merging**). *We define $\bigoplus$ as the merging operator for probabilistic cache states. It takes in n probabilistic cache states $\mathcal{C}_i = \langle C_i, X_i \rangle$ and a corresponding weight function w as input s.t. $\sum_{i=1}^{n} w(\mathcal{C}_i) = 1$. It produces a merged probabilistic cache state $\mathcal{C}$ 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 concrete states in C is the union of all the concrete cache states in $C_1, \ldots, C_n$. The probability of a concrete cache state $c \in C$ is a weighted summation of the probabilities of c in the input probabilistic cache states.

Let $in(B)$ define the set of predecessors basic blocks. Then, we can derive the incoming probabilistic cache state of B by employing the merging operation $\bigoplus$ on the outgoing probabilistic cache states of $in(B)$. We define the weight function w as $w(\mathcal{C}_{B'}^{out}) = f(B' \rightarrow B)$, where $B' \in in(B)$ is a predecessor of block B. Then given $in(B) = \{B', B'', \ldots\}$

$$\mathcal{C}_B^{in} = \bigoplus(\mathcal{C}_{B'}^{out}, \mathcal{C}_{B''}^{out}, \ldots, w) \tag{3}$$

Figure 2 shows the merging operator at the input of B_4. There are two probabilistic cache states $\mathcal{C}_{B2}^{out}$ and $\mathcal{C}_{B3}^{out}$ at the entry of B_4. As the two incoming edges to B_4 have equal probability, the

resulting probabilistic cache state at the entry of B_4 contains $\mathcal{C}^{out}_{B2}$ and $\mathcal{C}^{out}_{B3}$ with equal probability. The output probabilistic cache state $\mathcal{C}^{out}_{B4}$ is obtained by updating input probabilistic cache state $\mathcal{C}^{in}_{B4}$ with memory block $m4$ inside B_4.

4.3.2. Hit Rate Computation.. Recall that $gen_B = \langle m_1, \ldots, m_k \rangle$ is the sequence of memory blocks accessed within a basic block B. Now let us define k random variables $Y_1, \ldots, Y_k$ corresponding to the memory blocks $m_1, \ldots, m_k$ in gen_B. Y_i denotes the cache hit/miss event for the access of memory block m_i. Now Y_i can be modeled as a random variable with Bernoulli distribution by assuming $Y_i = 1$ if m_i is a cache miss and $Y_i = 0$ otherwise.

$$\begin{aligned} Pr[Y_1 = 1] &= PMiss(\mathcal{C}^{in}_B, m_1) \\ Pr[Y_i = 1] &= PMiss(\mathcal{C}^{in}_B \trianglelefteq m_1 \ldots \trianglelefteq m_{i-1}, m_i), \;\; 1 < i \leq k \\ Pr[Y_i = 0] &= 1 - Pr[Y_i = 1], \;\; 1 \leq i \leq k \end{aligned}$$

By definition of Bernoulli distribution, the hit rate of memory block m_i can be computed as $Pr[Y_i = 1]$.

4.3.3. Analysis of Loop. In the previous section, we have derived the incoming and outgoing probabilistic cache states of each basic block for a single iteration of the loop body starting with the empty cache state $\mathcal{C}^{in}_L = \mathcal{C}_\perp$. However, for a loop iterating multiple times, the input cache state 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 $\mathcal{C}^{in}_{start\langle n \rangle} = \mathcal{C}^{out}_{end\langle n-1 \rangle}$ for $n > 1$. However, in order to compute $\mathcal{C}^{in}_{start\langle 1 \rangle}, \ldots, \mathcal{C}^{in}_{start\langle N \rangle}$, where N is the loop bound, we *do not* need to traverse the DAG N times. Instead, we introduce two new operators.

DEFINITION 8 (**Concatenation of Concrete Cache States**). *Given two concrete cache states* $c1, c2$

$$c_1 \odot c_2 = c \textit{ where } c = c_1 \lhd c_2[A] \ldots \lhd c_2[1]$$

DEFINITION 9 (**Concatenation of Probabilistic Cache States**). *Given probabilistic cache states* $\mathcal{C}_1 = \langle C_1, X_1 \rangle$ *and* $\mathcal{C}_2 = \langle C_2, X_2 \rangle$

$$\begin{aligned} &\mathcal{C}_1 \bigodot \mathcal{C}_2 = \mathcal{C} \textit{ where } \mathcal{C} = \langle C, X \rangle \\ &C = \{c | c = c_1 \odot c_2, c1 \in C_1, c_2 \in C_2\} \\ &Pr[X = c] = \sum_{c_1 \in C_1, c_2 \in C_2, c = c_1 \odot c_2} (Pr[X_1 = c_1] \times Pr[X_2 = c_2]) \end{aligned}$$

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

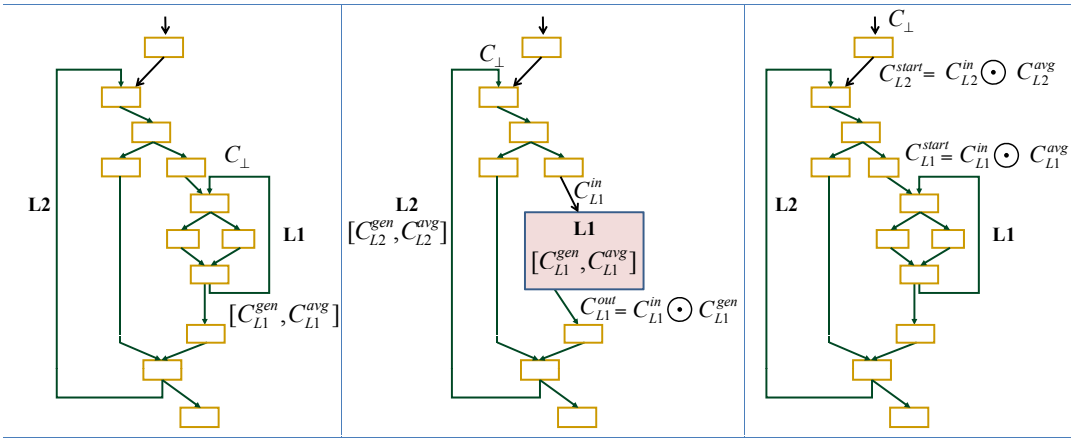
Now we can compute the outgoing probabilistic cache state of a loop L for each iteration by applying the $\bigodot$ operator. First, we note that $\mathcal{C}^{in}_{start\langle 1 \rangle} = \mathcal{C}^{in}_L = \mathcal{C}_\perp$. Then for iteration $n > 1$

$$\begin{aligned} \mathcal{C}^{in}_{start\langle n \rangle} &= \mathcal{C}^{out}_{end\langle n-1 \rangle} \\ \mathcal{C}^{out}_{end\langle n \rangle} &= \mathcal{C}^{in}_{start\langle n \rangle} \bigodot \mathcal{C}^{out}_{end\langle 1 \rangle} \end{aligned} \tag{4}$$

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

$$\mathcal{C}^{gen}_L = \mathcal{C}^{out}_{end\langle N \rangle} \tag{5}$$

The hit/miss of memory blocks in basic block B is dependent on the input probabilistic cache state $\mathcal{C}^{in}_B$ of the corresponding basic block B, which in turn is dependent on $\mathcal{C}^{in}_{start\langle n \rangle}$ of the loop L. To compute these probabilities for each memory block in each iteration is computationally expensive



(a) Inner loop analysis (b) Outer loop analysis (c) Whole program analysis

Fig. 3. Analysis of whole program.

and is equivalent to complete loop unrolling. Instead, we observe that we only need to compute an "average" probabilistic cache state $\mathcal{C}^{avg}_L$ at the $start$ node of the loop body. This captures the input cache state of the loop over N iterations.

$\mathcal{C}^{avg}_L$ can be defined as

$$\mathcal{C}^{avg}_L = \bigoplus(\mathcal{C}^{in}_{start\langle 1\rangle}, \ldots, \mathcal{C}^{in}_{start\langle N\rangle}, w) \tag{6}$$

where $w(\mathcal{C}^{in}_{start\langle n\rangle}) = \frac{1}{N}$. Now, in Section 4.3.1, we simply replace $\mathcal{C}^{in}_{start} = \mathcal{C}_\perp$ with $\mathcal{C}^{in}_{start} = \mathcal{C}^{avg}_L$. The rest of the analysis for the DAG remains unchanged.

More importantly, for any cache configuration, the operator $\odot$ need not be invoked N times in practice. The probabilistic cache states converge very quickly for most loops. After convergence point, both the content of probabilistic cache state and its associated probability do not change.

4.3.4. Special case for Direct Mapped Cache. The computation of $\mathcal{C}^{avg}_L$ and $\mathcal{C}^{gen}_L$, as discussed earlier, is quite general and works for fully associative, set-associative, as well as direct mapped caches. However, for direct mapped caches (where $A = 1$), the computation of average probabilistic cache state is much simpler. As mentioned earlier, in a direct mapped cache with K cache sets, each cache set is treated independently. Let M_i denote the set of all the memory blocks that can map to the i^{th} cache set. Then, a concrete cache state c corresponding to the i^{th} cache set is a vector $\langle c[1]\rangle$ of length 1, where $c[1] \in M_i \cup \{\perp\}$.

As before, we assume $\mathcal{C}^{in}_L = \mathcal{C}_\perp$ and let $\mathcal{C}^{out}_{end\langle n\rangle} = \langle C_{\langle n\rangle}, X_{\langle n\rangle}\rangle$ for $1 \leq n \leq N$ be the outgoing cache state after n^{th} iteration. It is easy to see that if associativity $A = 1$, then given any two iterations n, n' where $n \neq n'$, the set of concrete cache states remain unchanged, i.e., $C_{\langle n\rangle} = C_{\langle n'\rangle}$. Moreover, if $c_\perp \notin C_{\langle n\rangle}$, then the probability distribution function remains unchanged across iterations as well. That is, $Pr[X_{\langle n\rangle} = c] = Pr[X_{\langle n'\rangle} = c]$ for any concrete cache state c. If $c_\perp \in C_{\langle n\rangle}$, then the probability distribution function changes across iterations. Let us assume that the concrete cache state $c \in C_{\langle n\rangle}$ contains the memory block $m \in M_i$. That is, $c[1] = m$. Also, let p be the probability of this cache state after first iteration, i.e., $Pr[X_{\langle 1\rangle} = c] = p$. Similarly, let q be the probability of the empty cache state after first iteration, i.e., $Pr[X_{\langle 1\rangle} = c_\perp] = q$. Now what is the probability of cache state c after n iterations? It is the summation of (1) the probability of the corresponding memory block m being accessed in n^{th} iteration, and (2) the probability of cache state c after $n-1$ iterations and no memory block being access in n^{th} iteration. Thus

$$Pr[X_{\langle n\rangle} = c] = p + q \times Pr[X_{\langle n-1\rangle} = c] \tag{7}$$

By solving this recursion with the base case $Pr[X_{\langle 1\rangle} = c] = p$,

$$Pr[X_{\langle N\rangle} = c] = p \cdot \tfrac{1-q^N}{1-q}$$
$$\sum_{n=1}^{N-1} Pr[X_{\langle n\rangle} = c] = p \cdot \frac{(N-1)(1-q) - q(1-q^{N-1})}{(1-q)^2}$$

For $c_\perp$, it is in cache after n iterations if no memory block is accessed in the previous n iterations, therefore the following equations hold

$$Pr[X_{\langle N\rangle} = c_\perp] = q^N$$
$$\sum_{n=1}^{N-1} Pr[X_{\langle n\rangle} = c_\perp] = \frac{q \cdot (1-q^{N-1})}{1-q}$$

$\mathcal{C}_L^{avg}$ is defined as the the average input probabilistic cache state at loop entry. Thus, based on Equation 6 and 4, the average probabilistic cache state at loop entry $\mathcal{C}_L^{avg} = \langle C, X\rangle$ can be computed as follows with $\mathcal{C}_L^{in} = \mathcal{C}_\perp$

$$C = C_{\langle 1\rangle}$$
$$\begin{aligned} Pr[X = c] &= \tfrac{1}{N}\left(Pr[X_{\langle 1\rangle} = c] + \ldots + Pr[X_{\langle N-1\rangle} = c\right) \\ &= \tfrac{p}{N} \cdot \tfrac{(N-1)(1-q)-q(1-q^{N-1})}{(1-q)^2} \end{aligned}$$

$$\begin{aligned} Pr[X = c_\perp] &= \tfrac{1}{N}(1 + Pr[X_{\langle 1\rangle} = c_\perp] + \ldots \\ &\quad + Pr[X_{\langle N-1\rangle} = c_\perp]) = \tfrac{1-q^N}{N\cdot(1-q)} \end{aligned}$$

The final probabilistic cache state after N iterations starting with empty cache state $\mathcal{C}_L^{in} = \mathcal{C}_\perp$, $\mathcal{C}_L^{gen}$ is

$$C = C_{\langle 1\rangle}$$

$$Pr[X = c] = Pr[X_{\langle N\rangle} = c] = p \cdot \frac{1-q^N}{1-q}$$

$$Pr[X = c_\perp] = Pr[X_{\langle N\rangle} = c_\perp] = q^N$$

4.3.5. Analysis of Whole Program. So far we have assumed that the execution of a loop starts with an empty cache state. In this section, we show how to compute the probabilistic cache state in the context the whole program. Recall that $\mathcal{C}_L^{gen}$ represents the final cache state of loop L after N iterations starting with an empty cache state. Also, we use $\mathcal{C}_L^{avg}$ to denote the average probabilistic cache state at loop entry across N iterations, again assuming that the loop L is executing in isolation. If $\mathcal{C}_L^{in}$ is the initial cache state for loop L in the context of the whole program, then the average probabilistic cache state of a basic block in loop L is computed by simply starting with the cache state $\mathcal{C}_L^{in} \bigodot \mathcal{C}_L^{avg}$. The analysis of the whole program then requires computing the initial probabilistic cache states for all the loops and procedures in the program.

In order to compute the initial cache states, we construct the loop-procedure hierarchy graph (LPHG) for the whole program. The LPHG represents the procedure call and loop nest relations in the application. 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 as shown in Figure 3(a). Next, we replace the innermost loops/procedures with "dummy" nodes in the DAG of the enclosing loop/procedure. While traversing the DAG of the enclosing loop/procedure, special care is taken for the dummy nodes. Let $\mathcal{C}_L^{in}$ be the input cache state for dummy node L during traversal of the

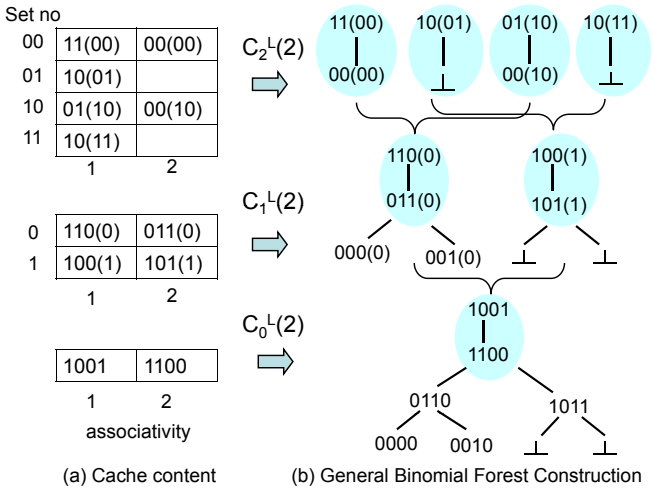


Fig. 4. Cache content and construction of generalized binomial forest. Memory blocks are represented by tags and set number, for example, for memory block $11(00)$, 00 denotes the set and 11 is the tag.

DAG. Then we treat the dummy node as a black box and compute the output cache state of the dummy node as $\mathcal{C}_L^{out} = \mathcal{C}_L^{in} \bigodot \mathcal{C}_L^{gen}$ as shown in Figure 3(b). At the end of this bottom-up traversal process, we reach the root node (`main` procedure). We have already computed $\mathcal{C}_L^{gen}$ and $\mathcal{C}_L^{avg}$ for all loops/procedures. Now we perform a top-down traversal to compute the cache state at each basic block in the context of the whole program. Suppose L is a dummy node in `main` with input cache state $\mathcal{C}_L^{in}$ and start node $start$. Then we traverse the DAG of L starting with $\mathcal{C}_{start}^{in} = \mathcal{C}_L^{in} \bigodot \mathcal{C}_L^{avg}$ as shown in Figure 3(c) and compute the probabilistic cache state at each node of the DAG. This top-down process continues till we reach all the innermost loops. At this point, we have computed the "average" probabilistic cache state for each basic block in the context of the whole program.

5. CACHE MODELING FOR MULTIPLE CONFIGURATIONS

Although the cache modeling we propose in section 4 can estimate the cache hit rate of a program for a specific configuration, it does not solve the problem of design space exploration due to vast number of cache configurations in the cache design space. Fortunately, there exist structural relations among the related cache configuration [Sugumar and Abraham 1995]. Based on this observation, in this section, we extend our analytical approach to model multiple cache configurations in one pass by utilizing the structural relations among related cache configurations. To exploit the structural relations among related cache configurations, we rely on **Generalized Binomial Tree (GBT)** [Sugumar and Abraham 1995] data structure. 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. Therefore, we propose probabilistic GBT to capture the cache states corresponding to all cache configurations and all contexts at any program point. Cache state operators such as update and concatenation are extended for probabilistic GBT. As for the underlying static program analysis, it almost remains the same as that of single configuration. Thus, we can derive the probabilistic GBT at each point of the program as before. Now, given a probabilistic GBT, we can easily estimate the cache hit rate of a memory access and entire program 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.

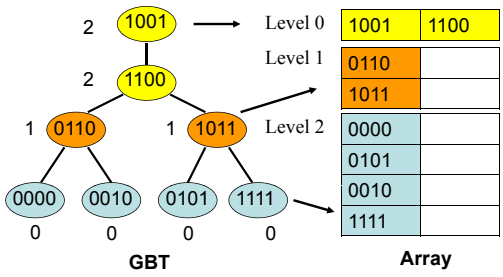


Fig. 5. Mapping from GBT to array. The nodes in GBT are annotated with their ranks.

5.1. Generalized Binomial Tree (GBT)

To exploit the inclusion property among related cache configurations, we rely on **General Binomial Forest (GBF)** data structure. Let us explain the Generalized Binomial Forest GBF data structure with an example. Let us recall that we consider LRU as the cache replacement policy. Figure 4(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 4. 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 [Sugumar and Abraham 1995].

Array Implementation. We use an array based implementation of GBT [Sugumar and Abraham 1995]. 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 5 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 5, 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 $\sum_{i=1}^{n} hit[m][i]$.

5.2. Probabilistic GBT

We now describe the probabilistic cache modeling based on GBT. The multiple cache configurations we support varying number of cache sets, degree of associativity and line size. In other words, we are interested in the set of configurations $\{C_S^L(n)|S_{min} \leq S \leq S_{max}; n \leq N; L_{min} \leq L \leq L_{max}\}$, where $2^{S_{min}}$ ($2^{S_{max}}$) is the minimum (maximum) number of cache sets, $2^{L_{min}}$ ($2^{L_{max}}$) is the minimum (maximum) line size and N is the maximum associativity.

Assumptions. First, each GBF corresponds to the set of cache configurations with constant line size but varying number of cache sets and associativity. In the following, we assume constant line size (L). As for the cache configurations with more than one GBFs (i.e. different line size), each GBF can be modeled independently. Second, for each GBF, we 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 $C_{S_{min}}^L(N)$ (i.e. the least significant S_{min} bits). Thus, there is no interference among 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 10 (**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 $Pr[X = c_\perp] = 1$.*

Now, 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 redefine 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. Also we extend operator $\bigoplus$ in section 4.3 to merge multiple probabilistic GBT.

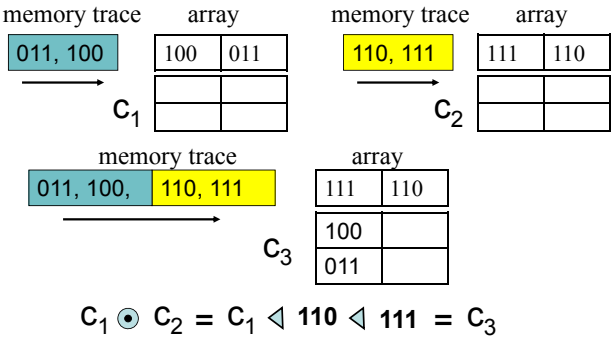


Fig. 6. Concatenation for GBTs where $M = 1$ and $N = 2$.

5.2.1. 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 c_1 and c_2
output: $c = c_1 \odot c_2$

```
1  c = c1;
2  for lev ← M to 0 do
3  |  Let T be the two dimension array at level lev in c2;
4  |  foreach row ∈ T do
5  |  |  for col ← N to 1 do
6  |  |  |  if T[row][col] ≠⊥ then
7  |  |  |  |  c = c ◁ T[row][col] ;
8  |  |  |
9  |  |
10 |
11 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 6. 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 11 (**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 $\mathcal{C}_1 \bigodot \mathcal{C}_2$.

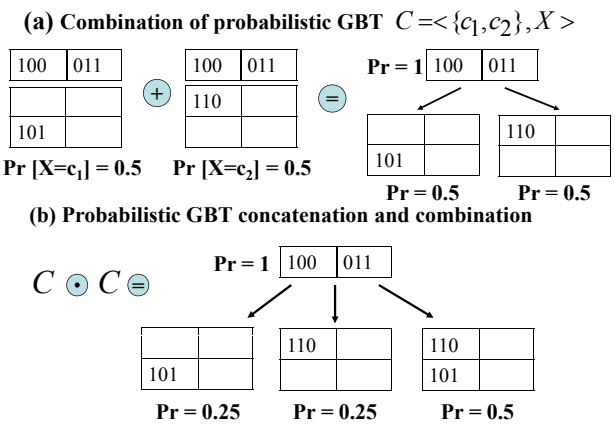


Fig. 7. Probabilistic GBT combination and concatenation.

5.2.2. Combining 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 combining 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 combine 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 combined together to have only one copy of the top k levels as shown in Figure 7(a). Also as the GBTs are combined together, the probabilities are now associated with each level rather than with the GBTs.

It is possible to perform combination at finer granularity, for example, using rows rather than levels. However, the complexity of the combination 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 combination 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 combine GBTs only if they are identical at the top levels.

The implementation of a combined GBT can be viewed as a tree with the sub-arrays (levels) of the original GBTs as nodes (see Figure 7(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 combination, m will be searched in all the original GBTs; now it will be searched only once in the combined GBT. For example, in Figure 7(a), before combination, the reference to memory block 100 is searched in both c_1 and c_2. With combined GBT, it is only searched once. In Figure 7(b), we show the combined probabilistic GBT after concatenation operation.

5.2.3. 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 combined GBT with two nodes at level k. Each node is a two dimension array 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 8.

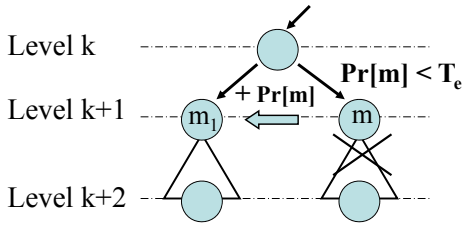


Fig. 8. Pruning in probabilistic GBT.

$$dist(n_1, n_2) = \sum_{\forall i,j} \begin{cases} N - j + 1, & \text{if } n_1[i][j] \neq n_2[i][j] \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

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 8, 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.

5.2.4. Cache Hit Rate of a Memory Block. Recall that in section 5.1, 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.

5.3. Static Cache Analysis

The static cache analysis remains almost the same as the one presented in section 4.3. All the probabilistic cache state operators such as update, merging, concatenation have been extended for probabilistic GBT. The analysis for DAG and loop iterations, and the whole program are not changed.

More importantly, the operator $\bigodot$ need not be invoked N times in practice 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 $\mathcal{C}^{out}_{end\langle n\rangle}$ and $\mathcal{C}^{out}_{end\langle n+1\rangle}$ 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}.

6. EXPERIMENTAL EVALUATION

In this section, we will first evaluate our analytical approach for cache design space exploration in terms of accuracy and efficiency. Then, we present how to use our approach for exploring design

Table I. Embedded benchmark characteristics

Benchmark	Trace Size (MB)		Time(sec)					Speedup	
	Trace1	Trace2	Prof	Trace	Dinero	Cheetah	Estimation	Dinero	Cheetah
bitcount	3583	1700	17.04	314.99	15569	198.15	0.091	171,096	2, 177
dijkstra	4700	2000	9.05	437.11	15380	235.07	0.094	163,625	2, 500
adpcmdec	791	1215	3.22	76.85	4415	42.14	0.265	16,661	159
adpcmenc	961	1261	4.10	94.01	4352	52.19	0.100	43,522	521
sha	706	622	0.69	65.14	3181	24.54	0.170	18,714	144
rijndael	1600	1400	0.99	136.07	23590	153.09	0.122	193,360	1,254
susans	4206	1400	5.70	392.49	15603	204.07	0.306	50,990	666
susanc	519	896	0.25	83.43	7878	74.55	0.747	10,546	99
gsmenc	2089	469	2.01	215.04	18749	174.16	1.757	10,671	99
gsmdec	1800	1400	8.29	154.51	14556	112.66	1.627	8,946	69

Table II. SPEC benchmark characteristics

Benchmark	Trace Size (MB)	Time(sec)					Speedup	
		Prof	Trace	Dinero	Cheetah	Estimation	Dinero	Cheetah
swim	2600	1	854	19880	108.62	0.882	22539	123
mgrid	3040	0.125	127	21910	110.15	3.891	5630	28
equake	6500	1.549	563	49245	136.25	3.685	13363	36
applu	4338	0.662	298	32886	175.69	5.189	6337	33

tradeoffs between performance and energy. Finally, we compare our approach with a state-of-the-art cache design space exploration technique [Zhang and Vahid 2003].

6.1. Experiments Setup

We try a set of embedded applications from MiBench benchmark suite [Guthaus et al. 2001] and larger general-purpose applications from SPEC2000. The details of the benchmarks are shown in Table I and Table II, respectively. We compare the accuracy and efficiency of our approach with trace-driven simulator *Dinero* [Edler and Hill] and *Cheetah* [Sugumar and Abraham 1995]. *Dinero* is a widely used trace-driven cache simulator, but it can only simulate one cache configuration at one time. Thus, it has to be invoked for each cache configuration in the design space. On the other hand, *Cheetah* is the fastest known cache simulator, which can simulate multiple cache configurations in a single pass. Both *Dinero* and *Cheetah* are simulators that can produce exact cache hit/miss count; but they are different in terms of running time.

We use Simplescalar toolset [Austin et al. 2002] for the experiments. We instrument *sim-profile*, a functional simulator, to collect the execution count of basic blocks and control flow edges for our analysis. The time spent in our instrumentation is shown in column *Prof* in Table I and II. We also change *sim-profile* to output the execution trace for trace-driven simulators and the overhead for trace generation is shown in column *Trace*. As shown, our profiling overhead is relatively small thanks to the efficient edge profiling [Ball 1994]. For embedded benchmark applications, we provide two inputs (*Trace1* and *Trace2*) to evaluate the robustness of our analysis against different inputs. Column *Trace Size* shows the corresponding trace size. As shown, the trace size can be quite large even for small inputs. The input to our analysis is just the basic block and control flow edge execution profile, whose size is so small that it can be ignored.

Given the program executable and execution profile, our analysis first disassembles the executable to construct CFG and LPHG, and then proceeds with the cache hit estimation. We perform all the experiments on a 3GHz Pentium 4 CPU with 2GB memory.

6.2. Accuracy

For each benchmark, we compare the estimated cache hit rate returned by our technique and the exact hit rate returned by simulation. We evaluate the embedded and SPEC benchmarks using different cache sizes as they have different working set sizes.

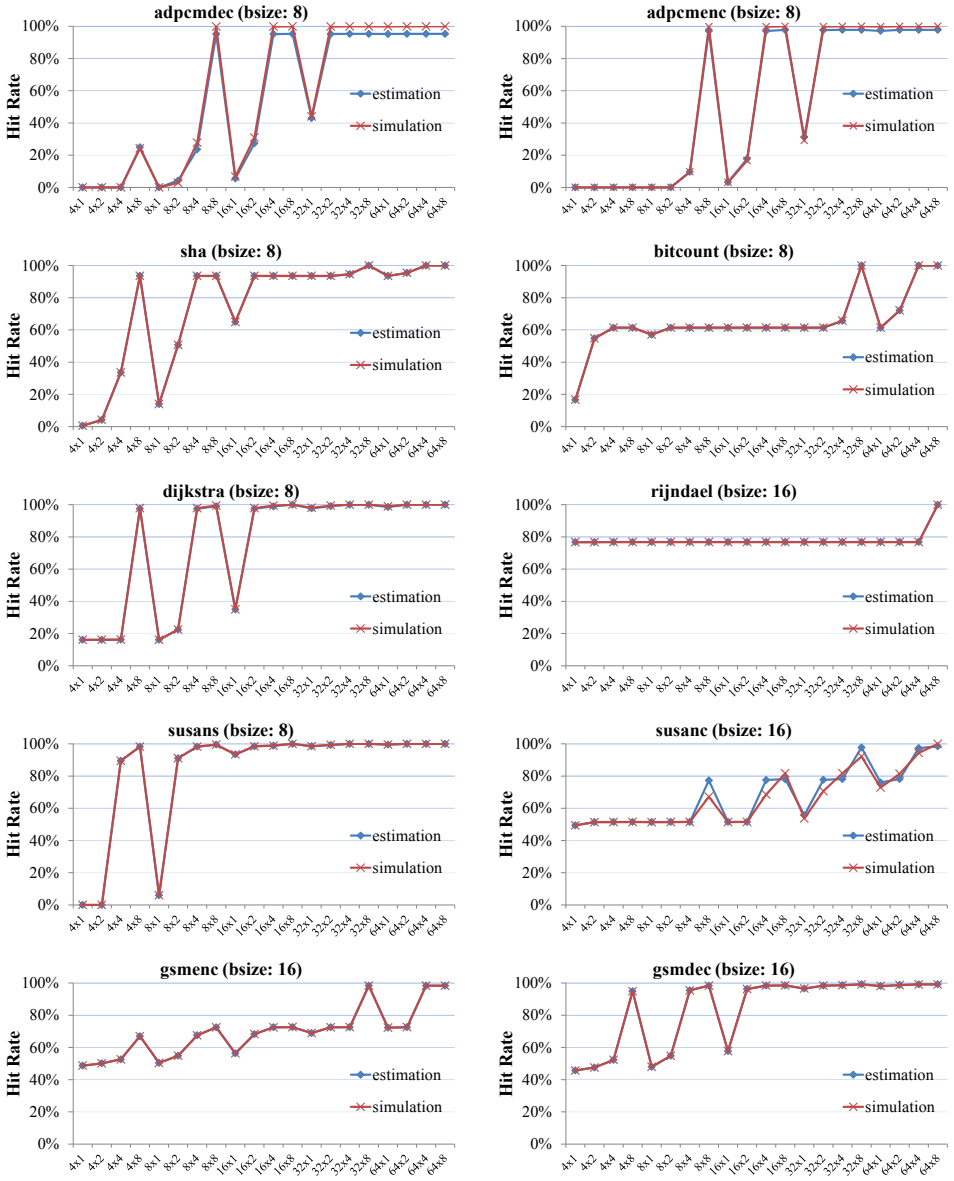


Fig. 9. Estimation vs simulation for embedded applications. *bsize* represents the cache block/line size.

For each embedded benchmark application, we vary the number of cache sets (4, 8, 16, 32, 64), associativity (1, 2, 4, 8), and block size (8, 16, 32 bytes). That is, a total of 60 configurations are estimated and simulated. The cache size in the design space ranges from 32 bytes to 16K. Figure 9 shows the results of simulation and estimation across 20 configurations (varying cache sets and associativity but with constant line size).

For each SPEC benchmark application, we vary the number of cache sets (16, 32, 64, 128, 256), associativity (1, 2, 4, 8), and block size (8, 16, 32 bytes). There are 60 configurations in the design space, which covers a wide range of caches from 128 bytes to 64K. Figure 10 shows the results of

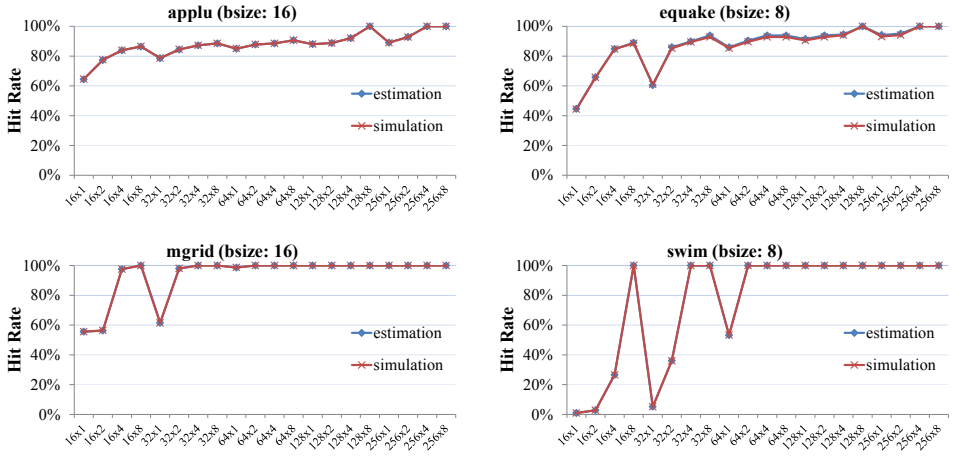


Fig. 10. Estimation vs simulation for SPEC applications. *bsize* represents the cache block/line size.

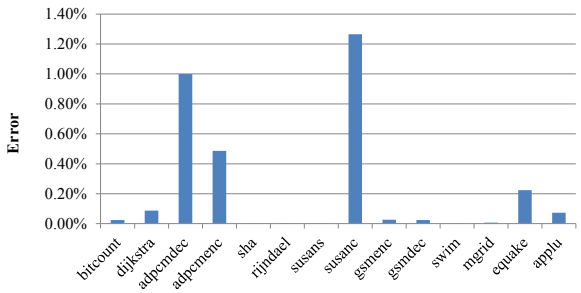


Fig. 11. Average estimation error.

simulation and estimation across 20 configurations (varying cache sets and associativity but with constant line size).

In Figure 9 and 10, the horizontal axis represents the selected 20 configurations, where $a \times b$ represents the cache with a cache sets and b associativity. As shown, the estimation results from our analysis track the simulation results quite closely for both the embedded and SPEC applications (*most of them are almost the same*). Figure 9 and 10 show the results of 20 configurations. *Only 20 configurations are shown but similar accuracy has been observed for other line sizes.* We define the estimation error as $|est - sim|$, where $est(sim)$ is the estimated (simulated) cache hit rate. The average error across 60 configurations for each benchmark is shown in Figure 11. For all the benchmarks and configurations, we achieve high accuracy (average error 0.2%).

In the above experiments, the estimation is based on the profile (basic block and control flow edges execution count) of one input. Then, the simulation results are collected using the same input as estimation. Here, we evaluate the sensitivity of our analysis across different inputs for the embedded applications (*Trace1* and *Trace2* in Table I). More clearly, we use the profile of one input and estimate its cache hit rate. Then, we compare it with the simulation result of another input. Overall, for all the benchmarks and cache configurations, our analysis is still accurate (0.5% average error). We achieve high accuracy because different inputs stay stable in terms of control flow edge probability. Thus, our derived probabilistic cache states are still accurate.

6.3. Efficiency

Recall that in section 5.3, we mentioned that the concatenation operator does not need to be invoked N (loop bound) times, because the probabilistic GBTs could converge. Here, we support this claim by concrete experimental results. Figure 12 shows the distribution of the number of iterations that the cache sets take to converge. Almost 80% of the cache sets converge after the second iteration for all the loops in all our benchmarks.

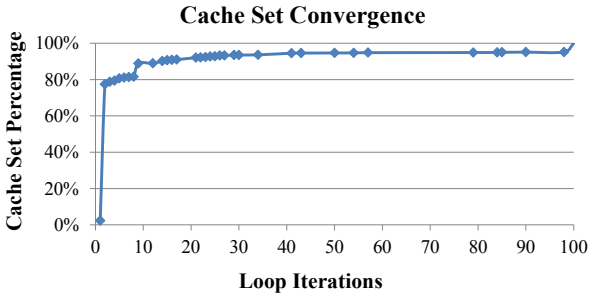


Fig. 12. Cache set convergence.

The runtime comparison between simulation and our analytical approach is shown in Table I and II.For simulation, both *Dinero* and *Cheetah* running time are shown. We first observe that compared to *Dinero*, *Cheetah* is much more efficient as it exploits the inclusion property among related cache configurations. The instrumentation overhead for basic block and control flow edge execution count collection required by our analysis is shown in column *Prof*. As for trace-driven techniques, the trace generation overhead is shown in column *Trace*. Both the instrumentation and trace generation need to be done only once, as the basic block execution profile and execution trace remain unchanged across different cache configurations. However, our analysis and simulation can be invoked multiple times for different set of cache configurations. Thus, we do not include the instrumentation or trace generation overhead for speedup calculation. The runtime of our analysis is shown in column *Estimation* and the speedup of our analysis over both *Dinero* and *Cheetah* are shown in Table I and II. Overall, our analysis is much faster than simulation. Compared to the fastest simulator *Cheetah*, our analysis is shown to be significantly faster (28 - 2,500X).

6.4. Performance and Energy Tradeoff

In the previous section, we have shown that our cache modeling is accurate in terms of cache hits/misses estimation. In this section, we aim to optimize the performance and energy consumption of the memory hierarchy using our analysis. It is very important to evaluate the design tradeoff between performance and energy consumption for various cache configurations. First, high cache hit rate does not always guarantee better performance as complex caches (large block size) have significantly longer access times. Furthermore, the energy per access of large and high associativity caches are greater than that of small and low associativity caches [Steven and Norman 1996]. Thus, large and high associativity cache may improve the performance at the expense of more energy consumption.

Performance and Energy Model. As for the cache access latency, we assume 1 cycle latency for cache hit. The main memory access is considered to be pipelined. The first access to main memory is 100 cycles, while the subsequent accesses take 2 cycles each. We utilize the energy model from [Zhang et al. 2003] which accurately models the energy consumption of the memory hierarchy. The energy model includes both dynamic and static energy consumption. The dynamic energy consumption is computed as follows

$$energy\_dynamic = cache\_hit \times energy\_hit + cache\_miss \times energy\_miss$$

where $cache\_hit(cache\_miss)$ are the number cache hits/misses and $energy\_hit(energy\_miss)$ are the energy consumption per cache hit or miss. We determine $energy\_hit$ of different cache configurations using the CACTI [Steven and Norman 1996] model for $0.13\mu m$ technology. The $energy\_miss$ includes the energy of off-chip memory access, cache block filling and processor stall [Zhang et al. 2003]. The $cache\_hit(cache\_miss)$ can be obtained by simulation or our analysis. Finally, each design point represents a cache configuration (number of cache sets, associativity, and block size) and it is associated with two numbers — performance and energy consumption.

Pareto-optimal Points. The entire cache design space (total 60 configurations) is explored via both simulation and estimation (our analytical modeling). The entire design space for both simulation and estimation are shown in Figure 13. Only the embedded applications are shown but similar accuracy has been observed for SPEC applications. As shown, our estimation is close to simulation results in terms of both individual points and entire design space.

In the large cache design space, we are only interested in the pareto-optimal points. Each pareto-optimal point represents a cache configuration. We now proceed to qualitatively evaluate the quality of the pareto-optimal points generated through simulation and our analysis. For each pareto-optimal point generated by either method, we additionally perform detailed simulation to obtain highly accurate performance and energy numbers using Wattch [Brooks et al. 2000]. Wattch is a micro-architecture level energy/performance simulator [Brooks et al. 2000]. However, Wattch does not model the energy consumption of memory. We extend it to include the energy consumption of memory component. We consider an in-order issue processor. Our focus is instruction cache, so we disable the data cache component in the Wattch simulator.

Now, we have two sets of pareto-optimal points — one from simulation and the other one from estimation. To compare these two sets of pareto-optimal points, we rely on the metric in [Zitzler et al. 2000]. Let X', X'' be two sets of pareto-optimal points,

$$C(X', X'') = \frac{|\{a'' \in X''; \exists a' \in X' : a' \preceq a''\}|}{|X''|}$$

where $a' \preceq a''$ means a' covers (dominate or equal) a''. $C(X', X'')$ is in interval $[0, 1]$, where $C(X', X'') = 1$ means that all solutions in X'' are covered by solutions in X'; $C(X', X'') = 0$ means that none of the solutions in X'' are covered by the set X'. Let sim, est be the two sets of pareto-optimal points for simulation and estimation, respectively. Then, we are interested in $C[est, sim]$. For all the benchmarks, $C[est, sim] = 1$. In other words, all the exact solutions (cache configurations returned via simulation) are covered by the solutions returned by our analysis.

Overall, our analysis can be employed in the early design stage. It can help the embedded system designers to accurately and efficiently explore the large cache design space to identify the cache configurations that optimize certain objectives, such as performance, energy consumption, or a combination of the two.

6.5. Comparison with Zhang-Vahid Method

In this section, we compare our technique with Zhang-Vahid method [Zhang and Vahid 2003] — a state-of-the-art cache design space exploration technique. Zhang-Vahid method is a hybrid approach, which uses heuristics to prune the cache design space but still needs multiple rounds of simulations to obtain the cache hits/misses for the selected cache configurations. In [Zhang and Vahid 2003], Zhang and Vahid demonstrated that their method can be used for searching the design points with the best energy consumption, best performance, and the pareto-optimal points in between. In the following, we first compare our technique with Zhang-Vahid method in terms of energy optimization. Then, we compare the two approaches for other design objectives.

When searching for the best configuration (cache size, line size and associativity) in terms of energy consumption, Zhang-Vahid method holds two parameters steady and vary the third one. The search depends on the impact of the varied parameters on miss rates and energy consumptions. In

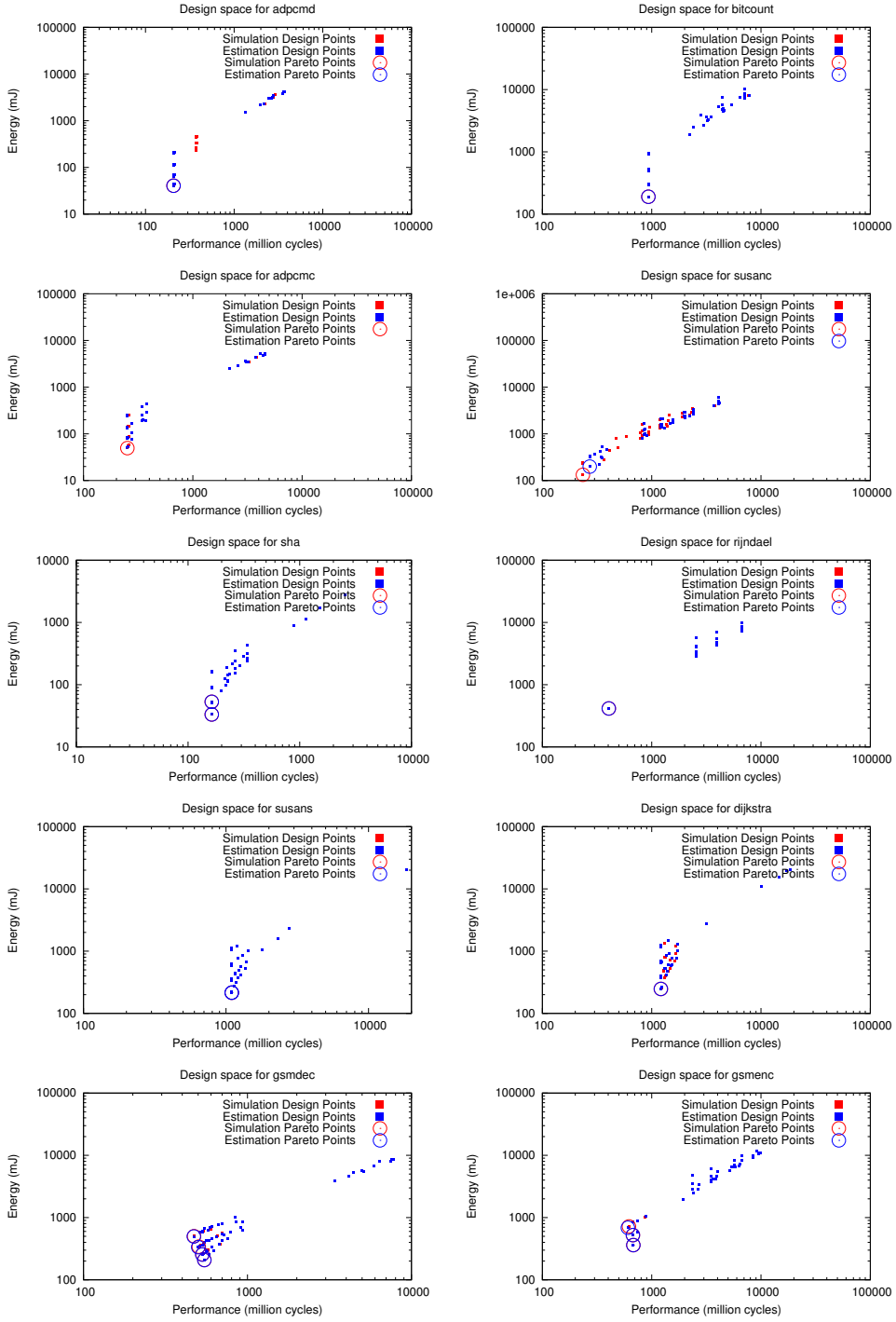


Fig. 13. Performance-energy design space and pareto-optimal points for both simulation and estimation.

Table III. Comparison with Zhang-Vahid method. (2048, 64, 1) represents the 2048 bytes cache with 64 bytes line size and 1-way associativity.

Benchmark	Zhang-Vahid		Ours		Optimal	Speedup
	run-time(sec)	Configuration	run-time(sec)	Configuration	Configuration	
bitcount	667	(2048, 64, 1)	0.15	(2048, 64, 1)	(2048, 64, 1)	4,446
dijkstra	659	(2048, 64, 1)	0.314	(2048, 64, 1)	(2048, 64, 1)	2,098
adpcmdec	189	(2048, 64, 1)	0.145	(2048, 64, 1)	(2048, 64, 1)	1,303
adpcmenc	186	(2048, 64, 1)	0.023	(2048, 64, 1)	(2048, 64, 1)	8,086
sha	136	(2048, 64, 1)	0.248	(2048, 64, 1)	(2048, 64, 1)	548
rijndael	1011	(2048, 64, 1)	0.214	(8192, 64, 1)	(8192, 64, 1)	4,724
susans	668	(2048, 64, 1)	0.682	(2048, 64, 1)	(2048, 64, 1)	979
susanc	450	(8192, 64, 1)	1.811	(8192, 64, 1)	(8192, 64, 1)	248
gsmenc	1071	(4096, 64, 1)	3.487	(4096, 64, 1)	(4096, 64, 1)	307
gsmdec	831	(8192, 64, 1)	2.731	(4096, 64, 1)	(4096, 64, 1)	304

their method, cache size is considered to be the most important parameter, followed by line size, and finally associativity. Zhang-Vahid heuristic is summarized as follows:

(1) Begin with the smallest size of cache (direct-mapped, smallest line size). If the increase in cache size (doubling the cache size) yields energy improvement, continue to increase the cache size until the limit. Finally, the best cache size with the best energy is chosen.
(2) For the best cache size determined by Step (1), if the increase in line size (doubling the line size) yields energy improvement, continue to increase the line size until the limit. Finally, the line size with the best energy is chosen.
(3) For the best cache size determined by Step (1) and the best line size determined by Step (2), if the increase in associativity (doubling the associaitivity) yields energy improvement, continue to increase the associativity until the limit. Finally, the associativity with the best energy is chosen.

Cache size, line size, and associativity are always a power to two. Thus, the search process moves to the next point by doubling the parameter value. In Zhang-Vahid method, to compare two configurations, simulations are required to obtain the cache hits/misses for both configurations. They use the cache simulator in SimpleScalar [Austin et al. 2002] for this purpose. In our experiments, we use trace-driven simulation *Dinero* as trace-driven cache simulator is faster than the functional cache simulator in SimpleScalar. Then, the cache hits/misses are fed into energy model to derive the energy consumption. For energy model, we use the model in Section 6.4, which is exactly the same energy model used by Zhang and Vahid [Zhang and Vahid 2003].

We explore the same cache design space used by Zhang-Vahid [Zhang and Vahid 2003]. The cache design space includes varying cache size (2K, 4K, 8K), varying block size (16, 32, 64 bytes), and associativity (1, 2, 4). Thus, there are total 27 cache configurations in the design space. The results are shown in Table III. For each benchmark, we show the corresponding optimal cache configuration. The optimal solution is obtained by exhaustively searching the cache design space using the exact cache hit/miss from simulation. As for our approach, we find the best cache configuration by exhaustively searching the cache design space using our estimated cache hits/misses.

First of all, our approach returns the optimal cache configuration for all the benchmarks, but Zhang-Vahid method fails for two benchmarks (*rijndael* and *gsmdec*). For *rijndael*, both our solution and the optimal solution are (8192, 64, 1), which represents the cache with 8192 bytes size, 64 bytes line size and 1-way associativity, but Zhang-Vahid method returns (2048, 64, 1). For *rijndael*, cache (4096, 16, 1) and cache (2048, 16, 1) return the same number of cache hits. Thus, in Zhang-Vahid method, cache (2048, 16, 1) is considered to be the best cache size due to its reduced energy consumption (bigger size cache consumes more energy per access). Though cache (4096, 16, 1) and cache (2048, 16, 1) return the same number of cache hits, cache(8192, 16, 1) achieves significantly more cache hits than cache (4096, 16, 1) because *rijndael* working set fits in 8192 bytes sizes cache.

Hence, Zhang-Vahid method can not return the global optimal solution because it is stuck at a local optimal solution. For *gsmdec*, both our solution and the optimal solution choose cache (4096, 64, 1), but Zhang-Vahid method chooses cache (8192, 64, 1). Zhang-Vahid method chooses 8192 bytes cache as the best cache size because cache (8192, 16, 1) consumes less energy than cache (4096, 16, 1). After fixing the cache size to 8192 bytes, Zhang-Vahid method finally chooses 64 bytes block size and 1-way associativity. However, for caches with bigger block size (64 bytes), cache (4096, 64, 1) is better than cache (8192, 64, 1) because they have similar number of cache hits and 4096 bytes cache incurs less energy per access than 8192 bytes cache. Hence, the global optimal solution is cache (4096, 64, 1).

Given an application, its optimal cache configuration depends on its particular temporal and spatial localities. Our approach is based on the cache states at any point of the program, which accurately capture the localities. Thus, our approach returns the optimal solution for all the benchmarks. On the other hand, Zhang-Vahid method can relatively easily get trapped in a local optimal when the cache behavior does not change monotonically following the cache size, line size and associativity increases. In Figure 14, we show that the energy consumption of the cache configurations returned by Zhang-Vahid method, our approach and the optimal solution for all the embedded applications. The energy consumption is normalized to the optimal solution. For 8 out of the 10 benchmarks, Zhang-Vahid method returns the optimal solution. However, it fails to return the optimal solution when the cache hits do not follow the trend predicted by the heuristic as shown by Table III. For example, Zhang-Vahid method incurs 62% and 3% more energy consumption than the optimal solution for *rijndael* and *gsmdec*, respectively. In summary, compared to Zhang-Vahid method, our technique achieves up to 62% and on average 7% energy savings for the evaluated benchmarks.

Second, Zhang-Vahid method could be very slow as simulations are still needed to collect the cache hits/misses for the configurations chosen by the heuristics. In comparison, our technique is much more efficient as it is based on static program analysis. Compared to Zhang-Vahid method, our technique achieves 304—8086 times speedup as shown in Table III. Finally, we also observe that, direct mapped and bigger block size caches turn out to be good choices for energy optimization. Similar observations have been made in [Zhang and Vahid 2003].

Our solution can be used for searching the pareto-optimal points with different energy and performance tradeoffs as shown in Section 6.4. Now, let us compare the two methods in terms of the pareto-optimal curve. Let us define the cache configuration with the best energy as A and the cache configuration with the best performance as B. To search for point B, Zhang and Vahid method performs a similar search: start from the cache with biggest size, biggest line size and highest associativity, then decrease each parameter if it yields performance improvement. Then, in order to find the pareto-optimal points with different performance and energy tradeoffs, all the points between A and B will be tested. For example, if the cache size of point A and B are 2K and 8K respectively, then all the cache sizes in between (2K, 4K, 8K) will be tested for pareto-optimal points. Since Zhang-Vahid method may not return the accurate point A (best energy) and B (best performance) as shown in Table III, their method may miss some of the actual pareto-optimal points with interesting design tradeoffs. Moreover, let us suppose all the cache configurations in the design space exhibit different performance and energy tradeoffs. In other words, all the cache configurations are on the pareto-optimal curve. Then, Zhang and Vahid method basically needs simulations of all the cache configurations, which is extremely slow as shown previously. However, our technique has been shown to be very accurate and fast.

7. CONCLUSION & FUTURE WORK

In this paper, we present a fast and accurate design space exploration technique for instruction caches using an analytical approach. We first introduce probabilistic cache states to represent the cache contents for multiple paths and derive probabilistic cache states at all program points through static analysis. Then, to explore design spaces with multiple configurations efficiently, we extend probabilistic cache states to probabilistic GBT that exploits inclusion property among related cache configurations. We also define appropriate operators for probabilistic GBTs, and discuss optimiza-

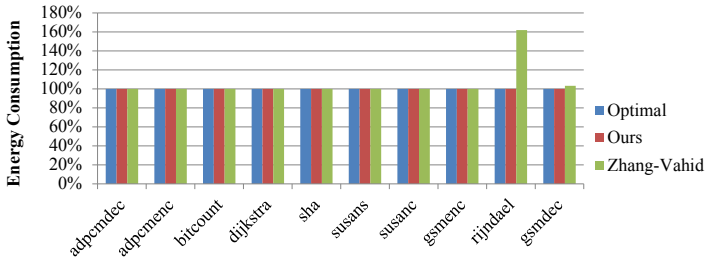


Fig. 14. Energy consumption comparison.

tions to improve their space and time efficiency. Our experimental results indicate that our method achieves significant speedup compared to the fastest trace-driven cache simulator (Cheetah) while maintaining high accuracy. We also show that through our analysis, we can efficiently evaluate the design tradeoff (performance vs energy) for different cache configurations. Compared to a state-of-the-art cache exploration technique, our approach achieves significant speedup and energy saving.

Our analytical model has been shown to be accurate and efficient for instructions caches. As future work, we will quantitatively evaluate our technique on data caches.

8. ACKNOWLEDGMENTS

This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2009-T2-1-033. We would like to thanks Mihai Pricopi for his help with SPEC benchmarks.

REFERENCES

ARNOLD, R., MUELLER, F., WHALLEY, D., AND HARMON, M. 1994. Bounding worst-case instruction cache performance. In *Real-Time Systems Symposium*. 172–181.

AUSTIN, T., LARSON, E., AND ERNST, D. 2002. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer 35,* 2, 59 – 67.

BALL, T. 1994. Efficiently counting program events with support for on-line queries. *ACM Transactions on Programming Languages and Systems 16,* 5, 1399–1410.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*. ISCA '00. 83–94.

EDLER, J. AND HILL, M. D. Dinero IV trace-driven uniprocessor cache simulator. `http://www.cs.wisc.edu/~markhill/DineroIV/`.

GHOSH, A. AND GIVARGIS, T. 2004. Cache optimization for embedded processor cores: An analytical approach. *ACM Trans. Des. Autom. Electron. Syst. 9,* 4, 419–440.

GORDON-ROSS, A. ET AL. 2007. A one-shot configurable-cache tuner for improved energy and performance. In *Proceedings of the conference on Design, automation and test in Europe*. DATE '07. 755–760.

GUILLON, C. ET AL. 2004. Procedure placement using temporal-ordering information: dealing with code size expansion. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*. CASES '04. 268–279.

GUTHAUS, M. R. ET AL. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001.* 3–14.

HAQUE, M. S., JANAPSATYA, A., AND PARAMESWARAN, S. 2009. Susesim: a fast simulation strategy to find optimal l1 cache configuration for embedded systems. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. CODES+ISSS '09. 295–304.

HILL, M. D. AND SMITH, A. J. 1989. Evaluating associativity in cpu caches. *IEEE Transactions on Computers 38,* 12, 1612–1630.

LI, X. F. ET AL. 2004. Design space exploration of caches using compressed traces. In *Proceedings of the 18th annual international conference on Supercomputing*. ICS '04. 116–125.

LI, Y. ET AL. 2000. Hardware-software co-design of embedded reconfigurable architectures. In *Proceedings of the 37th Annual Design Automation Conference*. DAC '00. 507–512.

LIANG, Y. AND MITRA, T. 2008a. In *Proceedings of the 45th annual Design Automation Conference*. DAC '08. 319–324.

LIANG, Y. AND MITRA, T. 2008b. In *Proceedings of the 6th IEEE/ACM international conference on Hardware/Software codesign and system synthesis*. CODES+ISSS '08. 103–108.

LIANG, Y. AND MITRA, T. 2010a. In *Proceedings of the 47th Design Automation Conference*. DAC '10. 344–349.

LIANG, Y. AND MITRA, T. 2010b. Improved procedure placement for set associative caches. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*. CASES '10. 147–156.

MATTSON, R. L. ET AL. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal 9,* 2, 78–117.

MICROBLAZE PROCESSOR. Xilinx, Microblze processor.http://www.xilinx.com/tools/microblaze.htm.

MONTANARO, J. ET AL. 1997. A 160-mhz, 32-b, 0.5-w cmos risc microprocessor. *Digital Tech. J. 9,* 1.

NIOS PROCESSOR. Altera, Nois Embedded Processor System.http://www.altera.com/devices/processor/nios2/ni2-index.html.

STEVEN, J. E. W. AND NORMAN, P. J. 1996. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits 31*, 677–688.

SUGUMAR, R. A. AND ABRAHAM, S. G. 1995. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems 13,* 1.

TENSILICA, XTENSA PROCESSOR. http://www.tensilica.com.

UHLIG, R. A. AND MUDGE, T. N. 1997. Trace-driven memory simulation: a survey. *ACM Comput. Surv. 29,* 2, 128–170.

WANG, W. H. AND BAER, J. L. 1991. Efficient trace-driven simulation methods for cache performance analysis. *ACM Trans. Comput. Syst. 9,* 3, 222–241.

WU, Z. AND WOLF, W. 1999. In *Proceedings of the seventh international workshop on Hardware/software codesign*. CODES '99. 95–99.

ZHANG, C. AND VAHID, F. 2003. Cache configuratoin exploration on prototying platforms. In *14th IEEE International Workshop on Rapid System Prototyping*. 164–.

ZHANG, C., VAHID, F., AND NAJJAR, W. 2003. A highly configurable cache architecture for embedded systems. *SIGARCH Comput. Archit. News 31,* 2, 136–146.

ZITZLER, E., DEB, K., AND THIELE, L. 2000. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evol. Comput. 8,* 2, 173–195.