Cache Modeling in Probabilistic Execution Time Analysis

Yun Liang, Tulika Mitra
School of Computing, National University of Singapore
{liangyun,tulika}@comp.nus.edu.sg

ABSTRACT

Multimedia-dominated consumer electronics devices (such as cellular phone, digital camera, etc.) operate under soft real-time constraints. Overly pessimistic worst-case execution time analysis techniques borrowed from hard real-time systems domain are not particularly suitable in this context. Instead, the execution time distribution of a task provides a more valuable input to the system-level performance analysis frameworks. Both program inputs and underlying architecture contribute to the execution time variation of a task. But existing probabilistic execution time analysis approaches mostly ignore architectural modeling. In this paper, we take the first step towards remedying this situation through instruction cache modeling. We introduce the notion of *probabilistic cache states* to model the evolution of cache content during program execution over multiple inputs. In particular, we estimate the mean and variance of execution time of a program across inputs in the presence of instruction cache. The experimental evaluation confirms the scalability and accuracy of our probabilistic cache modeling approach.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems.

General Terms

Measurement, Performance.

Keywords

Probabilistic Execution Time Analysis, Cache Modeling.

1. INTRODUCTION

Moore's Law has moved the center of gravity of computing from personal computers to numerous embedded computers hidden away inside our everyday electronic products. The application domain of embedded computing systems ranges from automotive, avionics, health-care to the multimedia-dominated consumer electronics devices. The safety-critical systems employed in automotive, avionics, and health-care domain demand strong timing predictability in addition to the functional correctness. Traditional schedulability analysis techniques can guarantee the satisfiability of timing constraints for such hard real-time systems consisting of multiple concurrent tasks. One of the key inputs required for the schedulability analysis is the Worst-Case Execution Time (WCET) of each of the tasks. WCET of a task on a target processor is defined as its maximum execution time across all possible inputs [16].

Multimedia-dominated consumer electronics devices, on the other hand, are known as soft real-time systems. These systems require the timing constraints to be satisfied *most of the time*. One can employ WCET-driven schedulability analysis in the context of soft real-time systems. But this approach leads to over-dimensioning of the processor resources due to the over-estimations inherent in static WCET estimation techniques. In particular, the complexity of static WCET analysis techniques has grown significantly over the years as embedded processors include performance enhancing features such as cache, branch prediction, out-of-order pipeline, etc [19]. While such complexities of WCET analysis have to be tolerated for hard real-time systems, systems with somewhat relaxed timing constraints call for novel performance analysis approaches.

Probabilistic schedulability analysis [6, 9] is gaining popularity in providing timing guarantees for soft real-time systems. Probabilistic analysis techniques can exploit the timing flexibility of soft real-time systems to offer better resource dimensioning while meeting the quality of service (QoS) requirements. Most proposals in probabilistic schedulability analysis assume probabilistic distribution of the execution times of the tasks. The distribution of execution times is also equally important in design space exploration, compiler optimizations and parallel program performance prediction for partitioning, scheduling, and load balancing [11, 18].

A naïve approach to derive this distribution through simulation or execution of a large number of "representative inputs" is not suitable for the following reasons. First, it is extremely difficult, if not impossible, to identify representative inputs for a complex program with billions of possible inputs. If the inputs are not chosen appropriately, then the corresponding distribution can be completely different from the actual distribution. Second and most importantly, the target platform may not be available during the design phase of an embedded system, thereby leaving slow simulation for a large number of inputs as the only choice. Thus static program analysis techniques need to be explored in this context. However, despite its importance, static analysis techniques to predict the distribution of execution times remain largely unexplored. Most importantly, the few static analysis approaches proposed either completely ignore the micro-architectural effects or leave it as future work [7, 11, 18]

In this work, we take the first step towards incorporating the timing effects of architectural features in probabilistic execution time analysis. In particular, we focus on the instruction cache in this pa-

DAC 2008, June 8–13, 2008, Anaheim, California, USA.

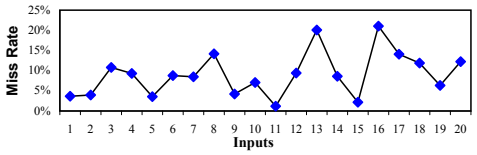

Figure 1: Variation in instruction cache miss rates for susan-c (cache configuration: 32 sets, 32-byte block, 2-way associativity).

per as (1) it is most commonly used in embedded processors, and (2) the variation in execution time from instruction cache effects can be quite large. For example, Figure 1 shows the wide variation in instruction cache miss rate of `susan-c` benchmark that translates to large variation in the execution time. Ignoring the instruction cache effect may produce a distribution that is far from the actual distribution. More concretely, our contribution in this paper is modeling the instruction cache timing effects in static estimation of the execution times distribution of a program. Ideally the static analysis technique should derive the probability distribution function (pdf) of the execution time. However, such elaborate analysis will be, in general, computationally expensive and often of little practical value. Instead, we characterize the execution-time distribution through *mean* and *variance*. If necessary, our work can be easily extended to include additional statistical moments (such as skewness and kurtosis) so that the distribution can be completely reconstructed [11].

Motivating Example. To illustrate the difficulty of modeling the instruction cache for probabilistic execution time analysis, let us consider the following program fragment.

```
for (i=1; i<=n; i++)
   if (a[i] != b[i])
       c[i] = a[i] * b[i]; /* S1 */
   else
       c[i] = a[i] / b[i]; /* S2 */
```

For the purpose of this example, let us ignore the execution time of the `for` and `if` conditions. Also let p be the truth probability of the branch condition `a[i] != b[i]`. Then the expected execution time of this program fragment per iteration will be estimated by existing techniques as $E[T] = p \times T_{S1} + (1-p) \times T_{S2}$ where T_{S1} (T_{S2}) is the constant execution time of statement S1 (S2). This assumption of constant execution time for each instruction fails in the presence of instruction cache, which has global timing effects.

In the presence of instruction cache, let us assume S1 and S2 conflict with each other for the same cache block. Now the execution time depends on whether S1 (S2) incurs cache miss, which in turn depends on the cache content after the previous iteration. For any loop iteration $i > 1$, the cache contains S1 with probability p and S2 with probability $1-p$ at the end of the previous iteration. Therefore, S1 (S2) will incur cache miss with probability $1-p$ (p). Thus the expected execution time per iteration now changes to $E[T] = p \times (T_{S1} + (1-p) \times \delta) + (1-p) \times (T_{S2} + p \times \delta)$ where δ is the cache miss penalty.

It is clear from the previous example that the cache content at any program point $\mathcal{P}$ depends on the probability of execution of the various paths leading to $\mathcal{P}$. Thus we introduce the notion of *probabilistic cache states* to capture the probability of the different cache contents at any program point. Our analysis technique then proceeds by transforming the probabilistic cache states as we traverse the control flow graph of the program. Once we compute the probabilistic cache states at all the program points, we can estimate the cache miss probability of any code block. Our experimental results with a number of embedded benchmarks confirm that the model is accurate in estimating the miss probability as well as mean and variance of execution time in the presence of instruction caches.

2. RELATED WORK

A general framework for determining average program execution time and their variance has been presented by Sarkar [18]. A static analysis framework to obtain probabilistic distributions of execution times has been proposed by David et al. [7]. Based on the assumption that external variables (input data) are independent and their probability distributions are known, they derive the execution time and probability of each path through the program. Gautama et al. [11] presents a program performance prediction approach. In their work, loop bounds, branch probabilities, and execution time of basic blocks all are characterized by their statistical moments. The objective of these works is to predict the statistical moments of performance distribution or even full probability distribution function. However, architectural features such as caches have not been modeled so far. In other words, the execution time variations are only from program level (variation in loop bounds, branch direction, etc.) and not from the architecture. Finally Bernat et al. [5] propose a WCET analysis technique (without architectural modeling) that estimates the WCET with a high probabilistic guarantee. In contrast, we are interested in the entire distribution rather than just the tail end of the distribution.

Instruction caches have been modeled for WCET estimation in hard real time system. Alt et al. [1] use abstract interpretation to model the cache behavior, while Li et al. [13] use cache conflict graph and propose an Integer Linear Programming(ILP) solution. Another technique based on categorization of cache accesses is proposed in [2, 15]. Lim et al. models instruction cache using timing schema in [14]. As these techniques have been developed in the context of WCET analysis, the instruction cache is modeled for the worst-case scenario. Given an address trace, [4, 17] propose analytical models to compute cache miss probability. In contrast, ours is a static analysis method that works on the program control flow graph to generate cache miss probability across multiple inputs and does not require address traces.

3. PROBABILISTIC TIMING ANALYSIS

The inputs to our analysis are the executable program code, cache parameters and program statistical information. We assume that the statistical information about the loop bounds and the truth probability of the conditional branches are provided as inputs to our analyzer. This information can be derived through either program analysis [7], user annotation, comprehensive profiling, or a combination of these approaches, which is beyond the scope of this paper. In the following, we use $E[X]$, $Var[X]$, $Cov[X,Y]$, Pr (where X and Y are random variables) to represent the expected value, variance, covariance, and probability, respectively.

Given a program, we first construct the loop-procedure hierarchy graph (LPHG) for the whole program [12]. The LPHG represents the procedure call and loop nest relations in the application. We assume that the loop or procedure body corresponds to a directed acyclic graph (DAG). The nodes of a DAG are the basic blocks within that loop or procedure. If a loop (procedure) contains other loops (procedures) within its body, then these inner loops (procedures) are represented by dummy nodes. The control flow graph within a loop is transformed such that every loop has a loop preheader and a post-loop node. In addition, there exists a unique start and end basic block corresponding to each such DAG (Figure 2).

Given a Basis block B, its execution frequency N_B is defined relative to the start basic block of the innermost loop or procedure it is in. Given the truth probability of the conditional branches, it is easy to compute $E[N_B]$. 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'. Again edge frequencies can be easily derived by

propagating the branch truth probabilities. As shown in Figure 2, $f(B_3 \rightarrow B_4)$ can be obtained from branch truth probability. By definition of edge frequency, $\sum_{e \in In(B)} f(e) = 1$, where $In(B)$ represents the incoming edges of B.

For each loop L, we define both relative loop bound N_L and absolute loop bound N'_L. Relative loop bound is the execution count of the loop in one execution of its preheader, while absolute loop bound is the total execution count of the loop in one complete execution of the program. For a procedure L, we only define the total number of invocations N'_L. Relative loop bound is used to derive the probabilistic cache states. Absolute loop bound is used to compute program execution time. Usually the loop bound of inner loop and outer loop are not independent. The expected value of loop bounds($E[N_L]$, $E[N'_L]$), the variance of loop bounds($Var[N'_L]$) and the covariance between two loop bounds ($Cov[N'_L, N'_{L'}]$) are input to our analyzer.

Mean Execution Time. Let us use $\mathbf{L}$ to represent the set of loops and procedures of the program. Let us use random variable T to represent the program execution time. Then the mean execution time of the program can be defined as

$$E[T] = \sum_{L \in \mathbf{L}} E[T_L] = \sum_{L \in \mathbf{L}} \left(E[N'_L] \times E[t_L] \right) \tag{1}$$

where T_L is the total execution time and t_L is the execution time per iteration of L. Let $\mathbf{B}$ be the set of basic blocks (excluding the dummy nodes for inner loops and callee procedures) in L and t_B be the execution time of basic block B per execution. Then

$$E[t_L] = \sum_{B \in \mathbf{B}} E[N_B] \times E[t_B] \tag{2}$$

As $E[N'_L]$ and $E[N_B]$ are known, the computation of $E[T]$ boils down to the computation of $E[t_B]$.

Execution Time Variance. The variance can be computed as [8]

$$Var[T] = \sum_{L \in \mathbf{L}} Var[T_L] + \sum \sum_{L, L'} Cov[T_L, T_{L'}] \tag{3}$$

$$\begin{aligned} Var[T_L] \quad &= Var[N'_L] \times Var[t_L] + E[N'_L]^2 \times Var[t_L] \\ &\quad + E[t_L]^2 \times Var[N'_L] \end{aligned} \tag{4}$$

By assuming $Var[t_B] = 0$ and ignoring the covariance between basic blocks, $Var[t_L]$ can be simplified as

$$Var[t_L] = \sum_{B \in \mathbf{B}} Var[N_B] \times E[t_B]^2 \tag{5}$$

The covariance between T_L and $T_{L'}$ can be approximated by

$$Cov[T_L, T_{L'}] = E[t_L] \times E[t_{L'}] \times Cov[N'_L, N'_{L'}] \tag{6}$$

$Var[N'_L]$, $Var[N_B]$ and $Cov[N'_L, N'_{L'}]$ all are known statistical information. Therefore, if $E[t_B]$ can be computed, both $E[T]$ and $Var[T]$ can be easily estimated. *In the following, we illustrate how to estimate $E[t_B]$ based on our probabilistic cache modeling.*

4. CACHE MODELING

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 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 defined as $(K \times A \times L)$. A memory block m can be mapped to only one cache set given by $(m \ \ modulo \ \ K)$. 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. Due to space limitations, 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. Clearly $\bigcap_{i=0}^{K-1} M_i = \phi$. Thus, there is no interference between the cache sets and they can be modeled independently.

In this paper, we assume LRU (least recently used) replacement policy, where the block replaced is the one that has been unused for the longest time. However, the technique presented in this work is general enough that it can be easily used for other replacement policies such as FIFO (first-in first-out).

More concretely, in the following, we consider a fully-associative LRU 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 $\perp$.

4.1 Concrete Cache States

Let us first formally define the concrete cache states and the operations involving concrete cache states. 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 loop body.*

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

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

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.*

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

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 in probabilistic execution time analysis. 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*

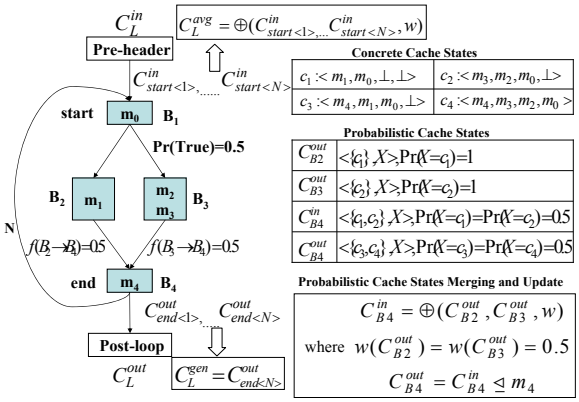

Figure 2: A loop with pre-header and post-loop node. The expected loop bound $E[N_L] = N$ and the truth probability of the branch at B1 is 0.5. The illustration is for a fully-associative cache with 4 blocks. $m0$–$m4$ are the memory blocks. Probabilistic cache states, merging and update operation are shown for B_4 in the first loop iteration starting with empty cache state $C_L^{in} = \mathcal{C}_\perp$ at loop pre-header.

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.*

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

For example, in Figure 2, the probabilistic cache state at the end of basic block B_4 after the first loop iteration (starting with empty cache state) consists of two concrete cache states c_3 and c_4 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 Analysis of Loops

In this subsection, we 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. In the following, 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

$$\mathcal{C}_B^{out} = \mathcal{C}_B^{in} \trianglelefteq m_1 \trianglelefteq \ldots \trianglelefteq m_k \tag{7}$$

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 predecessor 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 $in(B) = \{B', B'', \ldots\}$

$$\mathcal{C}_B^{in} = \bigoplus(\mathcal{C}_{B'}^{out}, \mathcal{C}_{B''}^{out}, \ldots, w) \tag{8}$$

Figure 2 shows the merging operator at the input of B_4. There are two concrete cache states $c1$ and $c2$ 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 $c1$ and $c2$ with equal probability. This probabilistic cache state is updated with memory block $m4$ inside B_4 to obtain the concrete cache states $c3$ and $c4$ with equal probability at the end of B_4.

4.3.2 Mean Execution Time of Basic Block

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.

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

By definition of Bernoulli distribution, $E[Y_i] = Pr[Y_i = 1]$. Let e_B be a constant denoting the execution time of basic block B assuming all cache hits. As defined in Section 3, t_B is the random

variable denoting the execution time of B when the cache is modeled. Then

$$E[t_B] = e_B + \left(\sum_{i=1}^{k} E[Y_i]\right) \times \delta \quad (9)$$

where δ is a constant denoting the cache miss penalty.

4.3.3 Extension to Loop Iterations

In the previous subsection, 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 $\mathcal{C}_L^{in} = \mathcal{C}_\perp$. 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 $\mathcal{C}^{in}_{start\langle 1 \rangle}, \ldots, \mathcal{C}^{in}_{start\langle N \rangle}$ as shown in Figure 2, where $N = E[N_L]$ is the expected 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* $c1, c2$

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

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

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

Let us assume the execution of two program fragments 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 $n > 1$

$$\begin{array}{ll} \mathcal{C}^{in}_{start\langle n \rangle} & = \mathcal{C}^{out}_{end\langle n-1 \rangle} \\ \mathcal{C}^{out}_{end\langle n \rangle} & = \mathcal{C}^{in}_{start\langle n \rangle} \bigodot \mathcal{C}^{out}_{end\langle 1 \rangle} \end{array} \quad (10)$$

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

$$\mathcal{C}^{gen}_L = \mathcal{C}^{out}_{end\langle N \rangle} \quad (11)$$

From Equation 9, the expected outcome of a cache access $E[Y_i]$ 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. Then the expected number of cache misses for memory block m_i (corresponding to Y_i) is the summation of the expected cache miss probabilities over $N = E[N_L]$ iterations. But computing these probabilities for each memory block in each iteration is computationally expensive 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. That is, $\mathcal{C}^{avg}_L = \langle C, X \rangle$ is defined in terms of $\mathcal{C}^{in}_{start\langle n \rangle} = \langle C_{\langle n \rangle}, X_{\langle n \rangle} \rangle$ for $1 \leq n \leq N$ as follows.

$$C = \bigcup_{n=1}^{N} C_{\langle n \rangle}$$
$$Pr[X = c | c \in C] = \frac{1}{N} \sum_{n=1}^{N} Pr[X_{\langle n \rangle} = c]$$

This can be alternatively defined as

$$\mathcal{C}^{avg}_L = \oplus(\mathcal{C}^{in}_{start\langle 1 \rangle}, \ldots, \mathcal{C}^{in}_{start\langle N \rangle}, w) \quad (12)$$

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.

The computation of $\mathcal{C}^{avg}_L$ and $\mathcal{C}^{gen}_L$ for direct mapped cache are simpler. In direct mapped cache, the concrete cache state will not change after the first iteration. Probabilistic cache state could be changed only if $\{\perp\}$ exists in it. Thus, closed form expressions exist for computing the probability of concrete cache states in $\mathcal{C}^{avg}_L$ and $\mathcal{C}^{gen}_L$, which we do not show due to space constraints.

More importantly, for any cache configuration, the operator $\bigodot$ need not be invoked $E[N_L]$ times in practice. The probabilistic cache states converge very quickly for most loops. 70% of the cache sets converge after the second iteration for all associativity settings (for all loops in all our benchmarks) and almost 80% cache sets converge within 10 iterations.

4.4 Analysis of Whole Program

In this section, we first show how to compute $\mathcal{C}^{gen}_L$ and $\mathcal{C}^{avg}_L$ for all the loops and then present how to compute the "average" probabilistic cache state for each basic block in the context of the whole program. We first traverse the LPHG in bottom-up fashion, i.e., we start with the innermost loops/procedures and compute $\mathcal{C}^{gen}_L$ and $\mathcal{C}^{avg}_L$ for all such loops/procedures. 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 $\mathcal{C}^{in}_L$ be the input cache state for dummy node L during traversal of the DAG. Then we treat the dummy node as a black box and compute the output cache state of the dummy node as $\mathcal{C}^{out}_L = \mathcal{C}^{in}_L \bigodot \mathcal{C}^{gen}_L$. At the end of this bottom-up traversal process, we reach the root node (`main` procedure). 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 $\mathcal{C}^{in}_L$ and start node $start$. Then we traverse the DAG of L starting with $\mathcal{C}^{in}_{start} = \mathcal{C}^{in}_L \bigodot \mathcal{C}^{avg}_L$ and compute the probabilistic cache state 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 cache state for each basic block in the context of the whole program. We can now use Equation 9 to compute mean execution time for each basic block.

5. EXPERIMENTAL EVALUATION

In order to evaluate the accuracy of our probabilistic cache modeling, we should ideally compare our estimation result with the actual mean and variance of execution time of a program, based on the given statistical information. However, given the statistical information, there is no way to determine the actual mean and variance (that is the exact problem we are trying to solve). Therefore, we decide to compare our estimation results to the results obtained from simulation. Given an application, we select multiple inputs and profile the application to collect the statistical information we state before. By simulating the application with multiple inputs, we could get the actual mean and variance of execution time across these multiple inputs. Then we apply our analysis technique based on the statistical information of these multiple inputs. Finally, we compare our estimation with the simulation results. We evaluate our modeling technique with nine benchmarks from MiBench. We provide for each benchmark multiple inputs with high variability [10]. We use SimpleScalar toolset [3] for the experiments. The

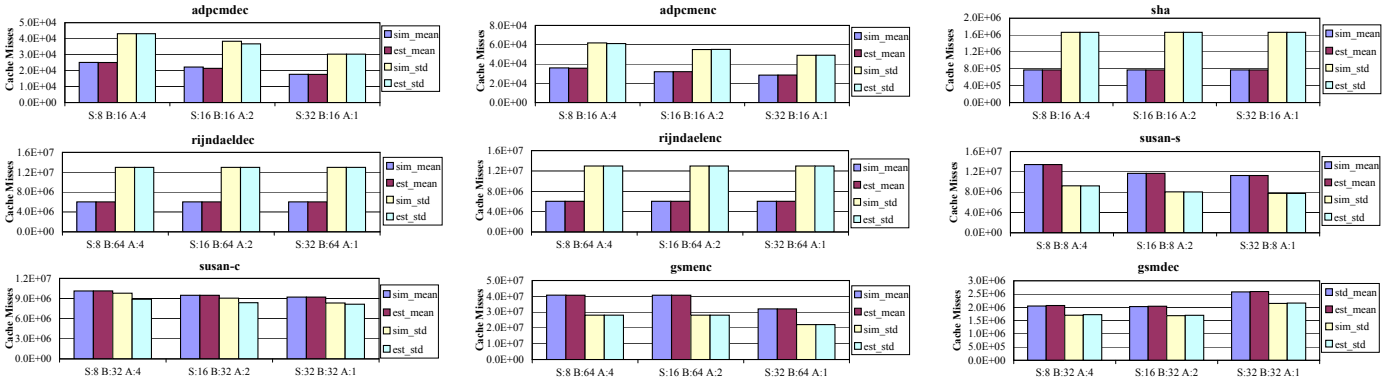

Figure 3: Est_mean (est_std) are estimated mean (standard deviation) and sim_mean (sim_std) are simulated mean (standard deviation). Est_mean (est_std) should be compared to sim_mean (sim_std). S, B, A denote number of cache sets, block size, associativity respectively.

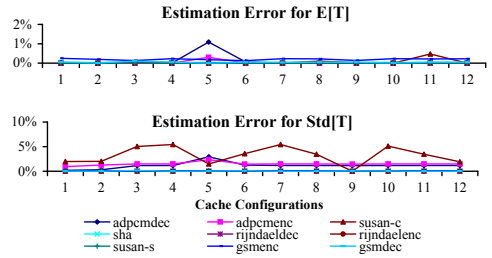

Figure 4: Estimation error for $E[T]$ and $Std[T]$ compared to simulation. Execution time standard deviation $Std[T]$ is $\sqrt{Var[T]}$. Error is defined as $\frac{|est-sim|}{sim} \times 100\%$.

profiling is done by sim-profile and cache simulation is done by sim-cheetah. Our estimator first disassembles the executable to construct CFG and LPHG, and then proceeds with the estimation.

Standard deviation is the square root of variance that measures the average deviation from mean. In the experiments we compare our estimated mean (standard deviation) to simulated mean (standard deviation). We fix a cache block size for each benchmark, but consider different number of cache sets (8,16,32) and associativity (1,2,4,8). So a total 12 cache configurations are simulated for each benchmark. As our modeling is focused on the instruction cache, we assume constant execution time for each basic block in the absence of caches. Figure 3 shows the mean and standard deviation of the total number of cache misses corresponding to simulation and estimation. Due to space consideration, we only show the values for three cache configurations per benchmark. The results are similar for other configurations. It is clear that our modeling is quite accurate in estimating both the mean and the standard deviation.

As for execution time, our estimation is accurate for both mean and standard deviation of execution time. Figure 4 shows our relative estimation error compared to simulation for all benchmark, cache configuration pairs. The average relative error across all the benchmark, cache configuration pairs are 0.05% and 0.7% for mean and standard deviation, respectively. Our estimation technique is also very fast and robust w.r.t cache configuration and benchmark size. The total runtime to estimate mean and variance for all the benchmarks and configurations is about 34 seconds on a 3.0GHz Pentium 4 CPU with 2GB memory.

6. CONCLUSION AND FUTURE WORK

This paper presents, for the first time, an approach to instruction cache modeling in probabilistic timing analysis. We introduce the notion of probabilistic cache states and define operators to manipulate probabilistic cache states at control flow merge points, across loop iterations, and within the whole program. Finally, we show how to compute the cache miss probability of a memory block at any program point given the probabilistic cache states. This allows us to include the variation due to cache behavior in estimating the execution time distribution of a program. Our experimental results indicate that the cache modeling presented is both accurate and scalable. In future, we plan to consider other architectural features (e.g., pipeline, branch predictor) in probabilistic modeling.

7. ACKNOWLEDGMENTS

This work was partially supported by NUS project R-252-000-292-112 and A*STAR SERC project R-252-000-258-305.

8. REFERENCES

[1] M. Alt et al. Cache behavior prediction by abstract interpretation. In *SAS*, 1996.

[2] R. Arnold et al. Bounding worst-case instruction cache performance. In *RTSS*, 1994.

[3] T. Austin et al. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

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

[5] G. Bernat et al. WCET analysis of probabilistic hard realt-time systems. In *RTSS*, 2002.

[6] A. Burns et al. A probabilistic framework for schedulabitity analysis. In *EMSOFT*, 2003.

[7] L. David and I. Puaut. Static determination of probabilistic execution times. In *ECRTS*, 2004.

[8] M. H. DeGroot. *Probability and Statistics*. Addison-Wesley, second edition, 1986.

[9] J. L. Diaz et al. Stochastic analysis of periodic real-time sytesms. In *RTSS*, 2002.

[10] G. Fursin et al. MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *HiPEAC*, 2007.

[11] H. Gautama and A. C. van Gemund. Static performance prediction of data-dependent programs. In *WOSP*, 2000.

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

[13] Y. S. Li et al. Performance estimation of embedded software with instruction cache modeling. *ACM TODAES*, 4(3), 1999.

[14] S. Lim et al. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, 1995.

[15] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2-3):217–247, 2000.

[16] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, 2000.

[17] G. Rajaram and V. Rajaraman. A probabilistic method for calculating hit ratios in direct mapped caches. *Journal of Network and Computer Applications*, 19(3):309–319, 1996.

[18] V. Sarkar. Determining average program execution times and their variance. In *PLDI*, 1989.

[19] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.