Probabilistic Modeling of Data Cache Behavior

Vinayak Puranik
Indian Institute of Science
Bangalore, India
vinayaksp@csa.iisc.ernet.in

Tulika Mitra
National Univ. of Singapore
Singapore
tulika@comp.nus.edu.sg

Y. N. Srikant
Indian Institute of Science
Bangalore, India
srikant@csa.iisc.ernet.in

ABSTRACT

In this paper, we propose a formal analysis approach to estimate the expected (average) data cache access time of an application across all possible program inputs. Towards this goal, we introduce the notion of *probabilistic access history* that intuitively summarizes the history of data memory accesses along different program paths (to reach a particular program point) and their associated probabilities. An efficient static program analysis technique has been developed to compute the access history at all program points. We estimate the cache hit/miss probabilities and hence the expected access time of each data memory reference from the access history. Our experimental evaluation confirms the accuracy and viability of the probabilistic data cache modeling approach.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques; D.2.8 [**Software Engineering**]: Metrics—*Performance Measures*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms

Performance, Verification

1. INTRODUCTION

Multimedia-dominated consumer electronic devices (e.g., digital camera, cell phones, audio/video recorders and players) are the major drivers of the embedded systems market today. These devices operate under soft real-time constraints and hence require the timing constraints to be satisfied *most* of the time. For example, a video player might impose the constraint that the MPEG decoder must process 30 frames per second. However, the system can still tolerate marginally slower frame rate and hence some missed frames occasionally. Thus, the embedded software running on devices with soft real-time constraints should be analyzed to ensure that timing constraints are satisfied frequently enough.

Most embedded software, in particular, multimedia software exhibit significant variability in their timing behavior [13]. In a com-

plex software, the large number of feasible program paths are exercised in different ways by different program inputs resulting in variable execution time. Additionally, the escalating performance requirements of embedded systems are forcing the designers to migrate towards complex hardware platforms. These platforms involve processors with performance enhancing features such as pipelines, caches, and branch prediction. These features further increase the timing variability [20]. For example, given a fixed program executing on a fixed architecture, the cache miss rate can vary widely across the different program inputs [14].

The timing variability issue is addressed currently in two different ways. The first approach borrows techniques from the hard real-time systems domain to provide worst-case timing guarantees. As hard real-time systems are mostly safety-critical in nature, they cannot afford to miss any deadline. Therefore, research in the past two decades have focused on developing program analysis methods to derive a bound on the worst-case execution time (WCET) of software [9, 15]. Unfortunately, employing the WCET value in design leads to major over-dimensioning of the system resources. In other words, the designer cannot exploit the somewhat relaxed timing constraints in soft real-time systems.

An alternative approach for soft real-time systems that is slowly gaining traction is the probabilistic schedulability analysis [10, 6, 5]. Probabilistic analysis techniques offer better resource dimensioning by ensuring that the timing guarantees are satisfied frequently enough (but may not be always). Most proposals in probabilistic schedulability analysis assume that the the execution time distributions of the tasks are known. The distribution of execution times also plays an important role in design space exploration, compiler optimizations and parallel program performance prediction for partitioning, scheduling, and load balancing [17, 12].

But how do we obtain the execution time distribution of a complex software running on an equally complex high-performance architectural platform? The prevailing practice is to simply simulate or execute the program on the platform with a "representative" set of inputs and derive an execution time distribution. Clearly, this approach is quite ad-hoc and the distribution is as good as the choice of the program inputs. It should be mentioned here that it is extremely difficult, if not impossible, to choose a representative set of program inputs for a complex software with millions (and sometimes billions) of possible inputs. In summary, existing approaches are inadequate in accurately deriving the execution time distribution of a soft real-time embedded software.

Static program analysis techniques can potentially offer a formal and rigorous approach towards deriving the execution time distribution of programs. A few static analysis approaches have been proposed in the literature in this context [17, 12, 4]. These approaches either completely ignore the architectural timing effects

or leave it as future work. However, the architectural features (in particular the memory hierarchy) have major impact on the execution time variation. Indeed, instruction and data cache modeling for WCET estimation have received considerable attention from the research community [9, 15]. In contrast, the recent instruction cache modeling work by Liang and Mitra [14] is so far the only attempt to include architectural timing effects in probabilistic execution time analysis. In this paper, we present *a static program analysis method for probabilistic data cache modeling to estimate the expected execution time of a software across all possible inputs.*

Data cache modeling is significantly more challenging compared to instruction cache modeling. An instruction is always fetched from the same memory address. But a single instruction can access a number of data memory locations. The set of memory locations accessed and their sequencing are hard to predict at compile time, especially for programs with irregular data access patterns. Therefore, timing variability is introduced in data caches not only by the control flow context in which an instruction is executed (which is the case for instruction cache), but also by the values of program variables that determine the sequence of data memory locations accessed by an instruction.

Liang and Mitra [14] introduce the concept of probabilistic cache states to capture the instruction cache state at any program point. A probabilistic cache state is nothing but a set of concrete cache states each associated with their probabilities. The number of concrete instruction cache states at any program point is bounded by the number of control flow contexts and is quite small in practice. However, the number of concrete data cache states can quickly become exponential as a single instruction (with a large set of potential data memory access locations) can generate multiple concrete cache states. Therefore, in this work, we apply an abstraction on the concrete cache states to capture all possible cache states at any program point and their probabilities in a compact representation with minimal loss of accuracy, as verified by the experimentation. We call this abstract cache state *probabilistic access history*. We define various operators that can work with probabilistic access history and present a program analysis technique to compute this abstract cache state at every program point. Finally, we show how the expected access times of the data memory reference (and from there the expected program execution time) can be derived given the probabilistic access histories.

2. RELATED WORK

Static estimation of cache behavior of programs has been extensively studied using the technique of Abstract Interpretation with abstract cache domains [7]. An extension has been proposed in [11] for data caches. However, an important step which is a prerequisite for data cache analysis, namely, address computation has not been discussed. Significant research has been carried out on modeling the data cache behavior of programs [8, 16, 19]. The work in [8] contributes methods for the worst-case timing analysis of data caches and set-associative caches. A bounded range of relative addresses for each data reference is determined using data flow analysis. Categorizations of data references are produced by a static cache simulator. The Cache Miss Equations (CME) approach is used in [16] for analyzing worst case data cache behavior. [19] describes techniques to estimate the WCET of executable code on architectures with data caches. Abstract Interpretation is used for tracking address computations and cache behavior. Techniques developed in [18] are employed to compute a safe approximation of the set of memory locations that can be accessed by any data memory reference. The results of address analysis are used to predict data cache behavior using the Abstract Interpretation technique.

After the computation of worst case execution costs for each basic block, an approximation of the overall worst case cost is obtained by solving an Integer Linear Program.

However, all the above data cache analysis techniques have been developed in the context of WCET analysis and hence the data cache is modeled for the worst-case scenario. In contrast, our work aims to model the data cache behavior for the determination of the mean execution time of a program across inputs.

3. OVERVIEW

In this section, we present a formal problem definition and an overview of our probabilistic data cache modeling approach.

Problem Definition. Given the executable program code and data cache parameters, our goal is to model the data cache behavior during program execution over multiple inputs. In particular, we compute the data cache hit/miss probability and hence the mean execution time of each data memory reference instruction in the program. Subsequently, we determine the mean execution time of the input program.

Note that ideally the static analysis technique should derive the probability distribution function 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 its mean. If necessary, our work can be easily extended to include additional statistical moments (such as variance, skewness and kurtosis) so that the distribution can be completely reconstructed [12]. Thus, our work can be used to compute an (unsafe) estimate of the worst case execution time (WCET) and the best case execution time (BCET) of tasks.

Assumptions. We make the following assumptions about the inputs available to our analysis technique and the platform.

- We model set-associative data caches with perfect Least Recently Used (LRU) replacement policy.

- The statistical information about the loop bounds (i.e., the mean iteration count for each loop across all possible program inputs) and the truth probability of the conditional branches are available to our analysis framework. This information can be derived through either program analysis [4], user annotation, comprehensive profiling, or a combination of these approaches. We assume that the statistical information about the program control flow across all possible inputs are derived external to our analysis framework and is not the focus of this paper.

Intuition behind Data Cache Modeling. In LRU replacement policy, the contents of data cache at any program point p are given by the most recent memory blocks accessed along the path ending at p. In other words, if we know the list of memory blocks accessed along a path ending at p, arranged according to 'recent accesses first' order, then we can predict the exact contents of the data cache at p. When there exist multiple paths ending at p, clearly, the contents of data cache at p depend on the actual path taken.

In static analysis we must account for all program paths from the entry point of the program to any point p. Intuitively, our idea is to maintain lists of memory blocks accessed along each path that ends at p, arranged according to 'recent accesses first' order, along with the probability of each path. Clearly, this information gives us all possibilities of the contents of the data cache at p, along with the probability that the data cache contains a particular set of memory blocks at p. This enables us to compute the data cache hit/miss probability of a memory block access at p.

Maintaining separately the lists of memory blocks accessed along each path from the entry point of a program to a point p is expensive, because the number of paths that end at p may grow exponentially. Moreover, as discussed in section 1, even for the same path, there can be many possible cache contents depending on the exact data memory locations accessed by the instructions along the path. Hence, we introduce an abstraction. In this abstraction, we group together the blocks which are i^{th} recently accessed on each path before ending at p, for $i = \{1, 2, \ldots\}$. With each memory block m contained in the set corresponding to the ith recent access, we associate the probability of the path ending at p, on which m occurs as the ith recent access. Thus, in this representation, each ith recent access is a set of memory blocks, and each memory block in the set has a probability value associated with it.

The data cache hit/miss probability of an access to memory block m at program point p is determined as follows. First, we convert the abstract collection of recent accesses into separate lists of individual memory blocks, where each list has memory blocks arranged according to 'recent accesses first' order. These lists are derived by taking a cross-product of all $i^{th}\{i = 1, 2, \ldots\}$ recent accesses. Then, with each individual list of memory blocks, we associate a probability value which is equal to the product of probability values associated with each memory block (of the list). A list of memory blocks (arranged in 'recent accesses first' order) gives the possible contents of the data cache at p and the probability value associated with the list gives the probability that the data cache contains this particular set of memory blocks. In theory, we need to keep track of all the memory blocks accessed along a path from program entry point to p. However, we conclude that, in practice, it is sufficient to keep track of only a constant number of accesses.

4. DATA CACHE MODELING

The data cache analysis problem comprises of two important sub-problems: calculating addresses of memory accesses and inferring cache behavior. We adapt the techniques developed in [18] for the first sub-problem, namely, calculating addresses of memory accesses (or address analysis). For the second sub-problem, namely, inferring cache behavior (or cache analysis), we describe a static analysis technique to determine the data cache hit/miss probability and hence the mean execution time of data memory reference instructions of the program. Subsequently, given the mean execution time of memory reference instructions within the program and the program statistical information about the loop bounds and the truth probability of conditional branches, the mean execution time of the whole program in the presence of data cache is computed.

4.1 Program Representation

Given a program, we first construct the Control Flow Graph (CFG) for each procedure in the program. Our analysis is flow sensitive, context insensitive. Hence, CFGs for individual procedures are linked together at call and return points to form a SuperGraph. We assume that the SuperGraph and every loop body corresponds to a directed acyclic graph (DAG). If the SuperGraph (or a loop) contains other loops within its body, then the inner loops are represented by dummy nodes. Thus, whenever we refer to 'nodes' or 'basic blocks' of a DAG, we include both the actual basic blocks of the program and the dummy nodes representing the DAGs of inner loops.

Given conditional branch truth probabilities and mean loop bounds, we can compute the basic block and edge frequencies. The basic block execution frequency N_B (of a basic block B) is defined relative to the start basic block of the innermost loop it is in. The edge execution frequency $f(B' \rightarrow B)$ (of an edge $B' \rightarrow B$) is defined

as the probability that B is reached from B'. Additionally, We define the probability of a path, $Pr(B' \rightarrow \cdots \rightarrow B)$ as the product of execution frequencies of edges that constitute the path.

4.2 Address Analysis

An important aspect of data cache analysis is determining the memory addresses accessed by data reference instructions. For immediate addressing, address determination is straightforward. However, in most cases addresses are computed by program instructions before the access is made. Also, the address set for data references may not be a singleton set, in for example, array references. Hence, we need a mechanism to determine the set of memory locations that are accessed by the data reference instructions. We adapt the analysis technique described in [18] for this purpose.

[18] proposes the Circular Linear Progressions (CLP) abstract domain for static analysis of executable code.

A CLP is represented as a 3-tuple (l, u, δ), where $l, u \in \mathbb{Z}(n), \delta \in \mathbb{N}(n)$, and the parameter n denotes n-bit representation. The components denote the starting point, ending point and positive step increment, respectively. The finite set of values abstracted by the CLP $C(l, u, \delta)$ is computed by the concretization function

$$conc(C) = \{a_i = l +_n i\delta | a_0 = l, a_s = u, i \le s, i \in \mathbb{N} \cup \{0\}\}$$

where $+_n$ denotes addition in n-bits, which is addition modulo 2^n.

The CLP abstract domain is used to track the values of statically identifiable memory locations and registers at program points. This is achieved by abstract transfer functions that are defined for a wide range of operations. For a data reference instruction, the abstract values of memory locations and registers (at the program point) are used to compute the set of memory locations that may be accessed. For our purpose, we assume that this analysis technique gives us a safe-approximation of the set of memory locations that can be accessed by any data memory reference.

Now, we describe our cache modeling framework, by introducing the concept of *access history*. Intuitively, *access history* refers to a history of memory block accesses along path(s) ending at a program point.

4.3 Cache Terminology

A cache memory is characterized by four major parameters: *capacity C, block or line size B, associativity A, number of sets K*.

The *capacity* is the number of bytes in the cache. The *block or line size* is the number of contiguous bytes transferred between the main memory and the cache. The *associativity* is the number of lines in a *set* where a particular block may reside. Depending on whether a block can reside in any cache line or in exactly one cache line or in exactly *A* cache lines of a set, the cache is called *fully-associative* or *direct mapped* or *A-way set-associative*, respectively. For A-way set-associative cache, the replacement policy (e.g., LRU, FIFO etc.) decides the block to be evicted when a cache set is full.

We consider memory as a set of blocks of size B each, $M = \{m_0, m_1, \ldots\}$. Each memory block m always maps to exactly one set of the cache and hence each cache set can be modeled independently. In the discussion that follows, we consider cache as a set of A lines. We assume perfect Least Recently Used (LRU) replacement policy, where the block replaced is the one that is unused for the longest time. To indicate that absence of a memory block access, we introduce an element $\perp$.

4.4 Concrete Access History

Concrete access and *concrete access history* are used later to introduce *probabilistic access* and *probabilistic access history*.
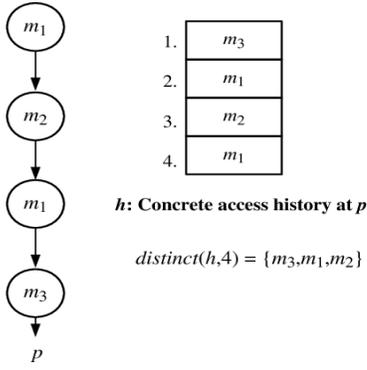
Figure 1: A program path with four memory accesses and the concrete access history h at the end of the program path.

DEFINITION 1. **(Concrete Access).** A concrete access *refers to a single memory block* $m \in M \cup \{\bot\}$.

DEFINITION 2. **(Concrete Access History).** A concrete access history *is a vector* $h = \langle h[1], h[2], \ldots \rangle$ *of unbounded length, where* $h[i]$ *is a* concrete access. *If* $h[i] = m$, *then* m *is the* i^{th} *most recently accessed memory block. We define* $h_\bot = \langle \bot \rangle$ *as an empty concrete access history.*

Given a concrete access history h and $n > 0$, $distinct(h, n)$ gives the first n distinct concrete accesses in h. By the definition of concrete access history, $distinct(h, A)$ gives the memory blocks that currently reside in the cache.

DEFINITION 3. **(Cache Hit).** *Given a concrete access history* h *and a concrete access* m,

$$hit(h, m) = \begin{cases} 1 & if\ m \in distinct(h, A) \\ 0 & otherwise \end{cases}$$

DEFINITION 4. **(Concrete Access History Update).** *We define* $\lhd$ *as the concrete access history update operator. Given a concrete access history* h *and a concrete access* m, $h \lhd m$ *defines the access history after memory access* m.

$$h \lhd m = \begin{cases} h, & if\ m = \bot \\ h', & where\ h'[1] = m \\ & h'[i] = h[i-1], i > 1 \end{cases}$$

Figure 1 shows an example of a program path with four memory accesses along the path and the concrete access history h at the end of the program path. The first entry in h is the most recent access, second entry is the second most recent access and so on. Note that access history is different from abstract cache state [11] in that a memory block may appear multiple times in the access history as seen in figure 1. Thus, the length of the access history is not bounded by the associativity of the cache. The length of the access history is equal to the cache associativity only in the case of a direct mapped cache. For higher associativity, a larger access history is desirable for better precision.

4.5 Probabilistic Access History

The concrete access history at any program point is dependent on the program path taken before reaching this point. A program point may be reached through multiple program paths leading to different possible access histories. Hence, in probabilistic data cache modeling we must model the probability of each access history possible at a program point. Hence, we introduce the notion of *probabilistic access history*.
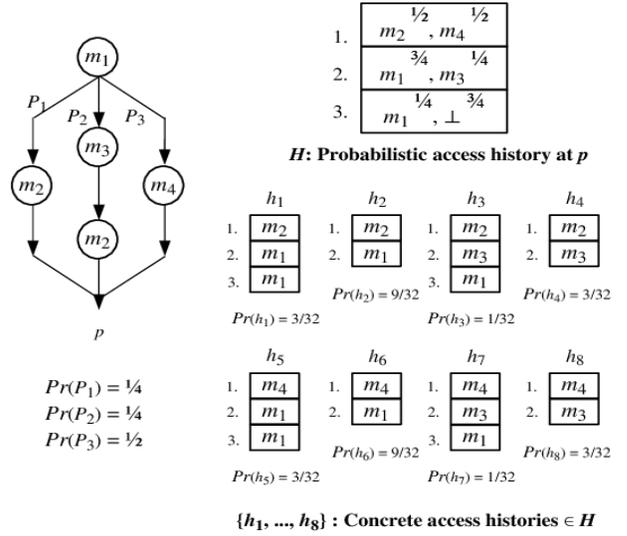

Figure 2: P_1, P_2 & P_3 **are program paths ending at point** p. H **is the probabilistic access history at** p. $\{h_1, \ldots, h_8\}$ **are concrete access histories obtained by taking a cross-product of probabilistic accesses in** H. $Pr(h_i)$ **is the probability associated with concrete access history** h_i.

DEFINITION 5. **(Probabilistic Access).** *A probabilistic access is a 2-tuple:* $\langle M, X \rangle$, *where* $M \subseteq M \cup \{\bot\}$ *is a set of concrete accesses and* X *is a random variable. The sample space of the random variable is* $M \cup \{\bot\}$. *Given a concrete access* m, *we define* $Pr[X = m]$ *as the probability of the concrete access* m *in* M. *If* $m \notin M$, *then* $Pr[X = m] = 0$. *By definition,* $(\sum_{m \in M \cup \{\bot\}} Pr[X = m]) = 1$.

DEFINITION 6. **(Probabilistic Access History).** *A probabilistic access history is a vector* $H = \langle H[1], H[2], \ldots \rangle$ *of unbounded length, where* $H[i]$ *is a probabilistic access. If* $H[i] = \langle M, X \rangle$, *then* $\langle M, X \rangle$ *is the* i^{th} *most recent probabilistic access. This means, for each concrete access* $m \in M$, $Pr[X = m]$ *is the probability that* m *is the* i^{th} *most recently accessed memory block. Intuitively,* $Pr[X = m]$ *is the probability of the path on which* m *occurs as the* i^{th} *most recently accessed memory block. We define* $H_\bot = \langle \langle \{\bot\}, X \rangle, Pr[X = \bot] = 1 \rangle$ *as an empty probabilistic access history.*

A probabilistic access history can also be described as follows: Given H, where $H[i] = \langle M_i, X_i \rangle, i = \{1, 2, \ldots\}$, define a set H_{CP} as the cross-product of the sets $M_i, i = \{1, 2, \ldots\}$. Since for each i, $M_i \subseteq M \cup \{\bot\}$, we can see that each element of H_{CP} is a concrete access history. That is, considering each $M_i = \{m_{i_1}, m_{i_2}, \ldots\}$, then each $h \in H_{CP}$ is of the form $\langle m_{1_j} \in M_1, m_{2_j} \in M_2, \ldots \rangle$, where $m_{i_j} \in M \cup \{\bot\}$. We define the probability of each $h \in H_{CP}$ as,

$$P_h = \prod_{i = \{1, 2, \ldots\}} Pr[X_i = m_{i_j}] \quad m_{i_j} \in M_i, j = \{1, 2, \ldots\}$$

Since for each i, $(\sum_{m \in M_i} Pr[X_i = m]) = 1$, we can see that $\sum_{h \in H_{CP}} P_h = 1$. Thus, a probabilistic access history can also be viewed as a collection of concrete access histories such that, the sum of the probability of each concrete access history is 1. Theoretically, it can be guaranteed that, the set of concrete access histories obtained from a probabilistic access history is *always* a superset of the actual concrete access histories at a program point. Hence, the set of possible cache states we get is *always* a superset of the actual concrete cache states possible at that point.
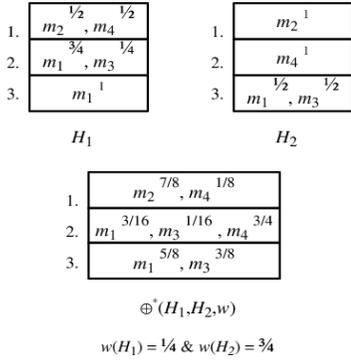
Figure 3: Merging operation for probabilistic access histories.

Figure 2 shows an example of three program paths P_1, P_2 and P_3 ending at point p, where $Pr(P_i)$ is the probability of path P_i. H is the probabilistic access history at p and each entry of H is a probabilistic access. For example, the most recent probabilistic access of H consists of memory blocks m_2, with probability (indicated as a superscript for memory block in the figure) $1/2$ (corresponding to the paths P_1 & P_2 on which m_2 is the most recent access) and m_4, with probability $1/2$ (corresponding to path P_3 on which m_4 is the most recent access). H can also be viewed as a collection of concrete access histories $\{h_1, \ldots, h_8\}$ where $Pr(h_i)$ is the probability associated with h_i. Note, the set $\{h_1, \ldots, h_8\}$ is an over-approximation of the actual concrete access histories $\{h_2, h_3, h_6\}$ at p.

DEFINITION 7. *(Cache Hit/Miss Probability).* *Given a probabilistic access history H and a concrete access m, the cache hit probability $PHit(H, m)$ of memory access m is,*

$$PHit(H, m) = \sum_{h \in H_{CP}, hit(h,m)=1} P_h$$

DEFINITION 8. *(Probabilistic Access History Update for Concrete Access).* *We define $\unlhd$ as the probabilistic access history update operator. Given a probabilistic access history H and a concrete access m, $H \unlhd m$ defines the probabilistic access history after concrete access m.*

$$H \unlhd m = \begin{cases} H, & \text{if } m = \bot \\ H', & \text{where } H'[1] = \langle\{m\}, X\rangle, Pr[X = m] = 1 \\ & H'[i] = H[i-1], i > 1 \end{cases}$$

In data cache analysis, the address set for data references may not be a singleton set, as described in section 4.2. Therefore, we describe a mechanism to update a probabilistic access history H with a safe-approximation of the set of memory locations that can be accessed by a data memory reference.

Let the address set of a memory reference be $\{m_1, \ldots, m_r\}$. In our analysis we assume that all the r references are equally likely, i.e., the instruction refers to each address with equal probability. Therefore, we treat the address set as a probabilistic access $\langle M, X \rangle$ where $M = \{m_1, \ldots, m_r\}$ and $Pr[X = m_i] = 1/r, 1 \le i \le r$. Then, we update H with each $m_i \in M$ separately and "merge" the resulting probabilistic access histories.

First, we formally define the "merging" operation for probabilistic access histories.

DEFINITION 9. *(Probabilistic Accesses Merging).* *We define $\oplus$ as the merging operator for probabilistic accesses. It takes in n*

probabilistic accesses $\langle M_i, X_i \rangle$ and a corresponding weight function w as input s.t. $\sum_{i=1}^{n} w(\langle M_i, X_i \rangle) = 1$. It produces a merged probabilistic access $\langle M, X \rangle$ as follows.

$$\oplus(\langle M_1, X_1 \rangle, \ldots, \langle M_n, X_n \rangle, w) = \langle M, X \rangle \text{ where, } M = \bigcup_{i=1}^{n} M_i,$$

$$Pr[X = m | m \in M] = \sum_{\forall i, m \in M_i} Pr[X_i = m] \times w(\langle M_i, X_i \rangle)$$

The concrete accesses in M is the union of all the concrete accesses in $M_1, \ldots, M_n$ and the probability of a concrete access $m \in M$ is a weighted summation of the probabilities of m in the input probabilistic accesses.

DEFINITION 10. *(Probabilistic Access Histories Merging).* *We define $\oplus^*$ as the merging operator for probabilistic access histories. It takes n probabilistic access histories $H_1, H_2 \ldots H_n$ and a corresponding weight function w as input s.t. $\sum_{i=1}^{n} w(H_i) = 1$. It produces a merged probabilistic access history H as follows.*

$$\oplus^*(H_1, \ldots, H_n, w) = H \text{ where,}$$

$$H[i] = \oplus(H_1[i], \ldots, H_n[i], w) \quad i = \{1, 2, \ldots\}$$

In other words, the probabilistic access history H is obtained by merging the ith most recent probabilistic accesses in each of $H_1, \ldots, H_n$, for $i = \{1, 2, \ldots\}$. Figure 3 shows an example of the merging operation for two probabilistic access histories.

Now, we define the operator to update a probabilistic access history H with a probabilistic access $\langle M, X \rangle$.

DEFINITION 11. *(Probabilistic Access History Update for Probabilistic Access).* *Given a probabilistic access history H and a probabilistic access $\langle M, X \rangle$, where $M = \{m_1, \ldots, m_r\}$, $H \unlhd \langle M, X \rangle$ defines the probabilistic access history after probabilistic access $\langle M, X \rangle$.*

$$H \unlhd \langle M, X \rangle = \oplus^*(H \unlhd m_1, \ldots, H \unlhd m_r, w) \text{ where,}$$

$$w(H \unlhd m_i) = Pr[X = m_i]$$

5. ANALYSIS OF PROGRAM

As mentioned earlier, given an executable program, we first construct the SuperGraph of the program. We assume that every loop (within the SuperGraph) has a single entry point (through the loop *start* node) and a single exit point (through the loop *end* node). As the first step, we isolate the CFG of every loop by replacing the loop body with a "dummy" node. If a loop contains other loops within its body, we isolate the CFGs of inner loops before the outer loop. Thus, we ensure that the body of the SuperGraph (and of every loop) corresponds to a directed acyclic graph (DAG). We are now ready to analyze the program.

The goal of our analysis is to compute the probabilistic access history at every program point, and use this information to compute the data cache hit/miss probability and hence mean execution time for each data reference instruction. We start with the analysis of the SuperGraph DAG, visiting the nodes of the DAG in a reverse postorder sequence. Note that, every node in the body of the SuperGraph (and every loop) DAG is either an actual basic block of the program or a dummy node representing a loop body. If the

node is an actual basic block, we update the incoming probabilistic access history with the memory references within the basic block, simultaneously calculating the data cache hit/miss probability for each memory reference (before updating the probabilistic access history with the memory reference). In the case of a dummy node, we first compute a "summary" of the accesses within the loop represented by dummy node, before using the "summary" information to analyze the loop DAG.

5.1 Analysis of DAG

Let *start* and *end* be the unique start and end basic blocks of a DAG. Let H_{start}^{in} and H_{end}^{out} be the incoming and outgoing probabilistic access histories of the DAG, respectively. For the analysis of the SuperGraph and for the summary of a loop, we consider $H_{start}^{in} = H_\perp$. Now, to perform the analysis of the DAG, we consider each of its basic blocks one by one (in a reverse postorder sequence) and do the following: (a) Compute the basic block's incoming probabilistic access history; (b) For each data memory reference within the basic block first compute the data cache hit/miss probability (and hence compute the mean execution time of the data reference instruction) and then update the probabilistic access history with the memory reference; (c) Compute the basic block's outgoing probabilistic access history.

Let H_B^{in} and H_B^{out} be the incoming and outgoing probabilistic access history of a basic block B, respectively. Let $gen_B = \langle m_1, \ldots, m_k \rangle$ be the sequence of memory blocks accessed within B. Then

$$H_B^{out} = H_B^{in} \trianglelefteq m_1 \trianglelefteq \ldots m_k$$

That is, the outgoing probabilistic access history of a basic block is derived by repeatedly updating the incoming probabilistic access history with the memory accesses in B. In order to generate the incoming probabilistic access history of B from its predecessors' outgoing probabilistic access history, we employ the merging operator $\oplus^*$.

Let $in(B) = \{B', B'', \ldots\}$ define the set of predecessor basic blocks. Then, we define the weight function w as $w(H_{B'}^{out}) = f(B' \rightarrow B)$, where $B' \in in(B)$ is a predecessor of block B.

$$H_B^{in} = \oplus^*(H_{B'}^{out}, H_{B''}^{out}, \ldots, w)$$

In the previous subsection we described the mechanism to update a probabilistic access history H with a safe-approximation of the set of memory locations that can be accessed by a data memory reference. Alternately, we can treat the address set as an $r - way$ branch, where each branch is taken with equal probability $1/r$. Therefore, we update the incoming probabilistic access history H with individual addresses m_i and merge the results using the $\oplus^*$ operator.

5.2 Summary of Loop

Let H_L^{in} and H_L^{out} be the incoming and outgoing probabilistic access histories of a loop L. Let $start$ and end be the unique start and end basic blocks of the DAG corresponding to the body of L. Then, $H_L^{in} = H_{start}^{in}$ and $H_L^{out} = H_{end}^{out}$. In the previous subsection, we described the method for deriving the incoming and outgoing probabilistic access histories of each basic block of a DAG starting with an empty probabilistic access history, i.e., $H_L^{in} = H_\perp$. However, for a loop iterating multiple times, the input probabilistic access history at the start node of the loop body is different for each iteration. Let us add the subscript $\langle n \rangle$ for the nth iteration of the loop.
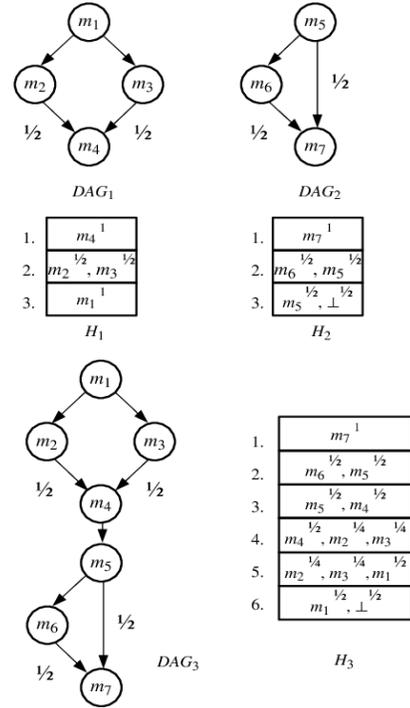

Figure 4: DAG_1 and DAG_2 and the probabilistic access histories after the execution of the corresponding DAGs, H_1 and H_2. DAG_3 is the sequential execution of DAG_1 and DAG_2, H_3 is the result of concatenation of H_1 and H_2 (**Edge probabilities are indicated**).

First, we note that $H_{start\langle 1 \rangle}^{in} = H_L^{in} = H_\perp$. Then for iteration $n > 1$

$$H_{start\langle n \rangle}^{in} = H_{end\langle n-1 \rangle}^{out}$$

Thus, $H_{start\langle 2 \rangle}^{in}, \ldots, H_{start\langle N \rangle}^{in}$ can be computed directly from $H_{end\langle 1 \rangle}^{out}, \ldots, H_{end\langle N-1 \rangle}^{out}$ respectively (here, N is the expected loop bound of L). $H_{end\langle 1 \rangle}^{out}$ is derived from $H_{start\langle 1 \rangle}^{in}$ by the analysis of the loop DAG (as described in previous subsection). However, in order to compute $H_{end\langle i \rangle}^{out}, 2 \leq i \leq N$ we do not need to traverse the complete loop DAG. Instead, we observe the following. Consider the execution of two program fragments (DAGs) each starting with an empty access history. Let, the probabilistic access history after the execution of the first and second fragments be H_1 and H_2 respectively. Then the probabilistic access history after the execution of the two fragments sequentially is the "concatenation" of H_1 and H_2. Formally, we define the "concatenation" operator as follows.

DEFINITION 12. *(**Concatenation of Concrete Access Histories**). Given two concrete access histories h_1 and h_2, where the length of h_2 is k*

$$h_1 \odot h_2 = h \text{ where } h = h_1 \triangleleft h_2[k] \cdots \triangleleft h_2[1]$$

DEFINITION 13. *(**Concatenation of Probabilistic Access Histories**). Given two probabilistic access histories H_1 and H_2, where the length of H_2 is k, i.e., $H_2 = \langle \langle M_1, X_1 \rangle, \ldots, \langle M_k, X_k \rangle \rangle$ and $Pr[X_i = \perp] = 0, 1 \leq i \leq k$*

$$H_1 \odot H_2 = H \text{ where } H = H_1 \triangleleft H_2[k] \cdots \triangleleft H_2[1]$$

That is, the most recent access of H_1 becomes the $(k+1)^{th}$ most recent access of $H_1 \odot H_2$, the second most recent access of H_1 becomes the $(k+2)^{th}$ most recent access of $H_1 \odot H_2$, and so on.

However if $\exists i, Pr[X_i = \perp] > 0$, then the definition of $\odot$ operator changes. Let i be the smallest value s.t. $Pr[X_i = \perp] > 0$. We add $H_1[1]$ to $H_2[i]$ with a probability value $Pr[X_i = \perp]$, $H_1[2]$ to $H_2[i+1]$ with a probability value $Pr[X_i = \perp]$ and so on. Consider the resulting probabilistic access history as the changed H_2. Once again, let i be the smallest value s.t. $Pr[X_i = \perp] > 0$. Continue as above until $H_1[1], H_1[2], \ldots$ are appended as the $(k+1)^{th}, (k+2)^{th}, \ldots$ most recent accesses of $H_1 \odot H_2$.

Figure 4 shows an example for computing the probabilistic access history after the sequential execution of two program fragments (DAG_1 and DAG_2) by applying the "concatenation" operator on the probabilistic access histories after the execution of individual program fragments (H_1 and H_2).

Now, we can compute the outgoing probabilistic access history of loop L for each iteration (i.e., $H_{end\langle i \rangle}^{out}, 2 \leq i \leq N$) by applying the $\odot$ operator.

$$H_{end\langle i \rangle}^{out} = H_{start\langle i \rangle}^{in} \odot H_{end\langle 1 \rangle}^{out}, 2 \leq i \leq N$$

The final probabilistic access history after N iterations starting with empty probabilistic access history, i.e., $H_L^{in} = H_\perp$, is denoted as H_L^{gen} where

$$H_L^{gen} = H_{end\langle N \rangle}^{out}$$

Intuitively, H_L^{gen} gives the history of accesses at the end of N iterations of the loop.

The data cache hit/miss probability of a memory reference within a loop depends on the input probabilistic access history, H_B^{in} of the corresponding basic block B, which in turn is dependent on $H_{start\langle n \rangle}^{in}$ of the loop L. Computing the data cache hit/miss probability of the memory reference in each iteration can be expensive as it involves computing the cross product of sets of accesses (of each probabilistic access) in the probabilistic access history (*Definition 7*.). Instead, we observe that we only need to compute an "average" probabilistic access history, H_L^{avg} at the *start* node of the loop body. This captures the input probabilistic access history of the loop over N iterations. Thus, H_L^{avg} is defined in terms of $H_{start\langle n \rangle}^{in}$ for $1 \leq n \leq N$ as

$$H_L^{avg} = \oplus^*(H_{start\langle 1 \rangle}^{in}, \ldots, H_{start\langle N \rangle}^{in}, w) \text{where,}$$

$$w(H_{start\langle i \rangle}^{in}) = 1/N$$

Thus, the objective of summarizing a loop L is computing the probabilistic access histories, H_L^{gen} and H_L^{avg}.

The $\odot$ operator is also used for the following. During the analysis of the DAG that contains L we directly derive the outgoing probabilistic access history of L by "concatenating" the incoming probabilistic access history with H_L^{gen}. Also, when we analyze the loop DAG in the context of the whole program, we compute the probabilistic access history at the start node of the loop, H_L^{in} by "concatenating" the incoming probabilistic access history with H_L^{avg}.

5.3   Analysis of Whole Program

In the previous subsections we described the analysis of a DAG and the summarization of a loop. We put these together to analyze the whole program. The calculation of the mean execution time of the input program happens as a part of the analysis.

The analysis of the program starts with a call to **analyze_program** (algorithm 1), which returns the mean execution time of the program. **analyze_program** calls **analyze_dag** (algorithm 2) passing the SuperGraph DAG and $H_\perp$ as parameters. The SuperGraph DAG and the constituent DAGs are analyzed via recursive calls to **analyze_dag** and **analyze_node** (algorithm 3).

Algorithm 1: analyze_program

1: $D \leftarrow$ DAG corresponding to SuperGraph
2: $exec\_time\_program \leftarrow$ analyze_dag$(D, H_\perp)$
3: **return** $exec\_time\_program$

Algorithm 2: analyze_dag (D, H_D^{in})

1: $exec\_time\_dag \leftarrow 0$
2: **for all** node B **in** D **do**
3:    **if** B is first node of D (reverse postorder sequence) **then**
4:       $H_B^{in} \leftarrow H_D^{in}$
5:    **else**
6:       Compute H_B^{in}
7:    **end if**
8:    $exec\_time\_node \leftarrow$ analyze_node(B, H_B^{in})
9:    $exec\_time\_dag \leftarrow exec\_time\_dag +$
                   $exec\_time\_node * N_B$
10: **end for**
11: **return** $exec\_time\_dag$

Algorithm 3: analyze_node (B, H_B^{in})

1: $exec\_time\_node \leftarrow 0$
2: **if** B is representative node **then**
3:    $L \leftarrow$ Loop represented by B
4:    $N \leftarrow$ Expected loop bound of L
5:    $(H_L^{gen}, H_L^{avg}) \leftarrow$ Summarize L
6:    $D \leftarrow$ DAG corresponding to L
7:    $H_D^{in} \leftarrow H_B^{in} \odot H_L^{avg}$
8:    $exec\_time\_node \leftarrow$ analyze_dag$(D, H_D^{in}) * N$
9:    $H_B^{out} \leftarrow H_B^{in} \odot H_L^{gen}$
10: **else**
11:    Compute $exec\_time\_node$ & H_B^{out}
12: **end if**
13: **return** $exec\_time\_node$

---

analyze_dag considers each node B of the input DAG D in a reverse postorder sequence. If B is the first node of the DAG, then the incoming probabilistic access history of D, H_D^{in} becomes the incoming probabilistic access history of B, H_B^{in}. Otherwise, H_B^{in} is computed as described in subsection 5.1. Next, **analyze_dag** calls **analyze_node** passing B and H_B^{in} as parameters. The mean execution time of the node returned by **analyze_node** is multiplied by the node frequency N_B and summed up to compute the mean execution time of the DAG.

analyze_node checks if B is a dummy node that represents a loop L. If so, it first summarizes L to compute H_L^{gen} and H_L^{avg} as described in 5.2, and then calls **analyze_dag** with the DAG corresponding to loop L and $H_B^{in} \odot H_L^{avg}$ as parameters. The mean execution time of the DAG returned by **analyze_dag** is multiplied by the expected loop bound of L to compute the mean execution time of the loop L represented by dummy node. Also, the outgoing probabilistic access history of B, H_B^{out} is computed as $H_B^{in} \odot H_L^{gen}$. If B is not a dummy node, then the mean execution time of B and H_B^{out} are computed as described in subsection 5.1.

6. ANALYSIS MODES

In data cache analysis each data reference is represented by a safe-approximation of the set of memory locations that can be accessed by the instruction. For example, the access of each individual array element at a program point (inside a loop) results in the approximation consisting of a set of memory blocks that represent the entire array. Thus, larger the array, larger is the size of the approximation. Further, an address approximation ignores the frequency and ordering of accesses within the approximation. This information is very important in determining the existence of spatial and temporal reuse. Therefore, as the approximation size increases, the data cache analysis becomes less precise (the loss of information about the frequency and ordering of accesses in the approximation becomes costly).

We deal with this problem by adapting a technique of access sequencing, proposed in [19]. In this technique, we employ both partial physical and virtual unrolling of loops. Partial physical unrolling of the outer loop partitions the iteration space into regions. We alternately select a region to be analyzed in either *expansion* or *summary* modes. The summary mode is described in subsection 5.2. The expansion mode can be visualized as virtual unrolling of the loop nest over the region and performing the analysis over this virtually unrolled loop. Expansion mode maintains sequencing within the region. In this mode, both Address and Cache Analysis are carried out simultaneously. The address computed for any reference in any pass is immediately used for data cache hit/miss probability calculation and to update the current probabilistic access history. The mean data reference instruction execution time is summed up as we analyze the basic blocks of a loop in this mode. This summed up value represents the mean execution time for the region analyzed in expansion mode.

To incorporate expansion mode of analysis, we only need to change the algorithm **analyze_node**. If B is a representative node, we first check if the loop L represented by B must be analyzed in summary or expansion mode. If L must be analyzed in summary mode, then the algorithm remains the same. However, if L must be analyzed in expansion mode, then H_B^{in} is considered as the input probabilistic access history for loop L and then, L is analyzed in *expansion* mode. The value returned by expansion analysis is considered as the mean execution time of L.

Consider a loop nest L_1 containing L_2. Let the expected loop bounds of L_1 and L_2 be n_1 and n_2 respectively. Then, *summary* mode takes time proportional to $n_1 + n_2$, whereas *expansion* mode takes time proportional to $n_1 * n_2$. Selection between the two modes is governed by a tradeoff between tightness of analysis and analysis time and can be controlled by the user of the analysis. The user specifies *fraction*, approximate fraction of total iteration space to be analyzed in expansion mode and *samples*, number of regions over which the fraction to be analyzed in expansion mode is to be distributed.

7. EXPERIMENTAL EVALUATION

We have evaluated our probabilistic data cache modeling technique on a set of benchmarks chosen from [3]. The edn_ programs are subroutines in the edn benchmark. We have adapted the implementation of the address analysis framework of [19]. This framework is implemented for the ARM7TDMI architecture[1]. We have implemented our cache modeling framework also for the ARM7TDMI architecture. The sources of the benchmarks are compiled with gcc to create ARM7 executables.

Our focus is on the data cache modeling technique and hence we present our results in terms of the mean data cache misses for the benchmarks. Clearly, it is infeasible to calculate the actual mean data cache misses of a program across all possible inputs (this is exactly the problem we are trying to solve). Instead, we approximate the same by observing the average cache misses across multiple inputs of the program. These observed cache miss numbers are obtained by running the executable on the Simplescalar/ARM simulator [2] (with a configurable cache model added).

As mentioned earlier, we also need the mean loop bound and conditional branch truth probabilities for each benchmark as input to our framework. This information can be derived through program analysis [4] external to our framework. However, our intention is to evaluate the accuracy of our data cache modeling technique. We want to avoid the inaccuracy in program statistical information estimation from affecting the accuracy of our analysis. Hence, we profile the benchmarks with the same program inputs used to obtain the observed mean cache misses. The profiler feeds our framework with mean loop bounds and truth probabilities of conditional branches. Note that the mean loop bound and conditional branch truth probabilities are underlying architecture independent. Hence, these values can be obtained by profiling applications on any architecture, only once, and then reused for different instances of static data cache analysis for different cache configurations.

Table 1 shows the result of our analysis for the following data cache configuration: Associativity = 1, Block size = 16 bytes, Total cache size = 1KB. We observe that the analysis is very accurate for direct mapped caches when the absolute data cache misses are low. For a direct mapped cache, we need to record only the most recent access that maps to each cache set to predict the exact contents of the cache. Hence, the probabilistic access history has a length of 1. As a result, the calculation of hit/miss probability for each memory reference and the loop analysis are fast. That is, for a direct mapped cache our analysis is efficient both in terms of memory usage and analysis time. The * entries in the table (edn_iir, edn_mac and $matmult$) indicate that the benchmarks were analyzed in *expansion* mode with an expansion of 20% (*fraction* = 0.2). For the remaining benchmarks expansion was not required. For edn_iir, edn_mac and $matmult$, it was observed that a higher number of memory blocks from address approximations mapped to the same cache set. That is, at many program points the most recent access (for many cache sets) comprised of a higher number of memory blocks. Hence, the benchmarks required to be analyzed in *expansion* mode. Note that a large number of memory blocks in the most recent access does not in itself lead to imprecise results; but combined with the actual order and frequency of the accesses this may contribute to imprecision. We explain this phenomenon with an example.

Consider a loop L with the expected loop bound $N = 8$ and a single data reference inside the loop body. Let the address set of the memory reference comprise of two blocks m_1 and m_2 (assume that both m_1 and m_2 map to the same cache set). Using the description of section 5.2, we figure out that the probabilistic access history of the loop H_L^{avg} comprises of three blocks m_1, m_2 and $\perp$ with associated probabilities of $7/16$, $7/16$ and $1/8$, respectively. This means that the hit probability for the access inside L as given by the analysis is $7/16$. Now in reality, if the order of accesses is m_1 (4 times) followed by m_2 (4 times), then the actual hit probability is $3/4$ which is greater than the value returned by the analysis. On the other hand, if the order of accesses is m_1 and m_2 alternating each other, then the actual hit probability is 0 which is less than the value returned by the analysis. This effect is more pronounced when there exist a large number of blocks in the probabilistic access history. However, if there existed a much tighter

address analysis which could give us the ordering and/or frequency information of the accesses in an access set, then our data cache analysis would be more precise. A step in this direction is a *probabilistic address analysis* similar to our probabilistic cache analysis and we are working towards such an analysis technique.

We also considered a cache configuration with Associativity =1, Block size =16 bytes, Total cache size =2KB and found that our analysis was quite accurate for all the benchmarks and no expansion was required.

In order to evaluate the quality of the analysis when the absolute data cache misses are high, we scaled up each benchmark by enlarging the loop iteration space and the size of data sets wherever possible (*jfdctint* is an exception as the program is designed for a specific data set size). In tables 2, 3 and 4 we show the results of our analysis for three different data cache configurations: (a) Associativity = 1, Total cache size = 1KB;(b) Associativity = 2, Total cache size = 1KB; and (3) Associativity = 4, Total cache size = 2KB. The Block size is 16 bytes.

For each benchmark, first we show the mean data cache misses computed by the analysis with 0% expansion. We observe that without any expansion the analysis is not very useful. Larger iteration space and larger data sets result in larger address sets for data reference instructions. When address sets are large, a higher number of blocks map to the same cache set. As explained previously, in the absence of a more informative address analysis the actual order and frequency of the block accesses may affect the precision of our analysis. This necessitates the analysis of portions of iteration space in *expansion* mode. The analysis in *expansion* mode is controlled by two parameters namely, *fraction* and *samples* (section 6). From our experiments we observed that increasing the amount of expansion (value of *fraction*) progressively improves the quality of analysis results. In tables 2, 3 and 4 we show the results for an expansion of 20%, 40% and 50%. A higher value of *samples* implies that more regions are selected for expansion. But, for smaller expansion values each such region only gets smaller. Hence, for expansion of up to 20% we restricted the *samples* to 2 and for expansion greater than 20% we set *samples* as 4. For programs with nested loops (*bsort, edn_fir, matmult*), we particularly observed that increasing expansion increased the analysis time. For most benchmarks there was a marked improvement in analysis results at 40% expansion. For *latsynth* a 20% expansion itself brought the analysis results close to observed values. However, *bsort* and *matmult* showed a very gradual improvement.

Apart from the values of *fraction* and *samples*, the size of the probabilistic access history also plays an important role in deciding the accuracy of the results. Expansion is more likely to produce singleton address sets for each memory reference and hence calls for a larger probabilistic access history especially when the associativity of the cache is high. We explain this with an example.

Consider a loop L with the expected loop bound $N = 6$ and a single data reference inside the loop body. Also, let the address set of the memory reference be $\{m_1, m_2\}$ where both m_1 and m_2 map to the same cache set. Let the actual order of accesses (and also that determined by the address analysis in expansion mode) be $m_1, m_2, m_2, m_2, m_2, m_1$. If associativity = 2, then clearly except for the very first accesses of m_1 and m_2, all other accesses result in a cache hit. But even with an access history size of 4, the analysis cannot detect the last access m_1 as a cache hit. However, we cannot arbitrarily increase the length of probabilistic access history. In summary mode of analysis, each probabilistic access may contain many memory blocks and thus a larger access history implies larger analysis time. Therefore, in our experiments we gradually increase the size of the access history with increasing expansion. For most

benchmarks the maximum access history size was 10 to 15. But, for some benchmarks (like $matmult$) we considered a maximum access history size of 30 for better results. In all the cases, the actual runtime of our analysis was in the order of seconds.

8. CONCLUSIONS

Data caches contribute to significant variability in the program execution time. In this work, given a program and a data cache configuration, we capture the variability in cache access time across all program inputs through fast and accurate static program analysis. This is achieved by introducing the notion of probabilistic access histories that can compactly summarize all possible cache states (along with their probabilities) at any program point for LRU set-associative caches. Our experimental evaluation shows that the abstraction causes minimal loss of accuracy specially when used in conjunction with selective loop unrolling. In the future, we would like to propose a *probabilistic address analysis* to improve the precision of our data cache analysis and extend our work to estimate the complete probability distribution function of the program execution time.

9. REFERENCES

[1] ARM7TDMI. Technical Reference Manual. "http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf".

[2] Simplescalar/ARM. "http://www.simplescalar.com/v4test.html".

[3] WCET Project/Benchmarks. "http://www.mrtc.mdh.se/projects/wcet/benchmarks.html".

[4] L. David and I. Puaut. Static Determination of Probabilistic Execution Times. In *ECRTS*, 2004.

[5] A. Burns et al. A Probabilistic Framework for Schedulability Analysis. In *EMSOFT*, 2003.

[6] J.L. Diaz et al. Stochastic Analysis of Periodic Real-Time Systems. In *IEEE RTSS*, 2002.

[7] M. Alt et al. Cache Behavior Prediction by Abstract Interpretation. In *SAS*, 1996.

[8] R. T. White et al. Timing Analysis for Data Caches and Set-Associative Caches. In *IEEE RTAS*, 1997.

[9] R. Wilhelm et al. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[10] S. Manolache et al. Schedulability Analysis of Applications with Stochastic Task Execution Times. *ACM Trans. Embedded Comput. Syst.*, 3(4), 2004.

[11] C. Ferdinand and R. Wilhelm. On Predicting Data Cache Behavior for Real-Time Systems. In *LCTES*, 1998.

[12] H. Gautama and A. J. C. van Gemund. Static Performance Prediction of Data-Dependent Programs. In *WOSP*, 2000.

[13] E. A. Lee. Computing needs time. *Commun. ACM*, 2009.

[14] Y. Liang and T. Mitra. Cache Modeling in Probabilistic Execution Time Analysis. In *DAC*, 2008.

[15] T. Mitra and A. Roychoudhury. Worst-Case Execution Time and Energy Analysis. In *The Compiler Design Handbook: Optimizations and Machine Code Generation*, pages 1–1 to 1–48. CRC Press, second edition, 2007.

[16] H. Ramaprasad and F. Mueller. Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns. In *IEEE RTAS*, 2005.

[17] V. Sarkar. Determining Average Program Execution Times and their Variance. In *PLDI*, 1989.

[18] R. Sen and Y. N. Srikant. Executable Analysis using Abstract Interpretation with Circular Linear Progressions. In *MEMOCODE*, 2007.

[19] R. Sen and Y. N. Srikant. Wcet Estimation for Executables in the Presence of Data Caches. In *EMSOFT*, 2007.

[20] L. Thiele and R. Wilhelm. Design for time-predictability. In *Design of Systems with Predictable Behaviour*, 2004.

Table 1: Results (Associativity = 1, Block size = 16 bytes, Total cache size = 1KB)

Benchmark	Total cache accesses	Observation (Mean cache misses)	Analysis (Mean cache misses)
bs	32	7	7.12
bsort100	29631	31	31.00
cnt	1364	37	36.99
edn_fir	10678	37	36.78
edn_iir	1840	115	112.99*
edn_latsynth	615	28	27.99
edn_mac	1835	81	79.99*
jfdctint	313	20	20.00
matmult	4572	115	114.85*

Table 2: Results (Associativity = 1, Block size = 16 bytes, Total cache size = 1KB)

Benchmark	Total cache accesses	Observation (Mean cache misses)	Analysis (Mean cache misses)			
			Expansion 0%	Expansion 20%	Expansion 40%	Expansion 50%
bs	653	166	394.43	313.27	176.81	176.06
bsort100	195,615	589	929.84	861.53	793.33	726.47
cnt	5,264	236	538.92	234.28	234.78	234.72
edn_fir	41,328	252	267.32	252.92	252.95	252.97
edn_iir	3,640	215	1528.10	722.53	252.67	212.99
edn_latsynth	4,842	168	1448.69	181.19	165.99	165.99
edn_mac	3,237	181	873.77	369.82	178.99	178.99
jfdctint	313	20	20.00			
matmult	6,001	415	1814.22	1486.01	1030.36	874.23

Table 3: Results (Associativity = 2, Block size = 16 bytes, Total cache size = 1KB)

Benchmark	Total cache accesses	Observation (Mean cache misses)	Analysis (Mean cache misses)			
			Expansion 0%	Expansion 20%	Expansion 40%	Expansion 50%
bs	653	166	397.23	310.29	153.10	136.81
bsort100	195,615	847	2,151.28	1,044.77	992.97	957.77
cnt	5,264	220	533.13	389.11	254.84	229.60
edn_fir	41,328	131	223.99	189.36	186.19	144.19
edn_iir	3,640	243	1349.61	563.25	276.62	229.34
edn_latsynth	4,842	153	1464.68	194.02	135.96	128.96
edn_mac	3,237	163	697.60	124.66	178.27	165.71
jfdctint	313	20	20.00			
matmult	6,001	170	1,618.11	865.65	412.29	345.91

Table 4: Results (Associativity = 4, Block size = 16 bytes, Total cache size = 2KB)

Benchmark	Total cache accesses	Observation (Mean cache misses)	Analysis (Mean cache misses)			
			Expansion 0%	Expansion 20%	Expansion 40%	Expansion 50%
bs	653	164	279.88	164.60	137.37	120.68
bsort100	195615	43	49.24			
cnt	5,264	114	236.40	134.81	131.97	122.84
edn_fir	41,328	69	333.74	207.21	163.28	160.50
edn_iir	3,640	187	672.75	259.59	175.84	179.12
edn_latsynth	4,842	108	452.63	138.90	115.05	116.35
edn_mac	3,237	111	297.90	106.53	106.69	105.16
jfdctint	313	20	20.00			
matmult	6,001	104	371.49	317.62	199.62	199.10