Program Performance Spectrum

Sudipta Chattopadhyay Lee Kee Chong Abhik Roychoudhury
National University of Singapore
{sudiptac,cleekee,abhik}@comp.nus.edu.sg

Abstract

Real-time and embedded applications often need to satisfy several non-functional properties such as timing. Consequently, performance validation is a crucial stage before the deployment of real-time and embedded software. Cache memories are often used to bridge the performance gap between a processor and memory subsystems. As a result, the analysis of caches plays a key role in the performance validation of real-time, embedded software. In this paper, we propose a novel approach to compute the cache performance signature of an entire program. Our technique is based on exploring the input domain through different *path programs*. Two paths belong to the same path program if they follow the same set of control flow edges but may vary in the iterations of loops encountered. Our experiments with several subject programs show that the different paths grouped into a path program have very similar and often exactly same cache performance.

Our path program exploration can be viewed as partitioning the input domain of the program. Each partition is associated with its cache performance and a symbolic formula capturing the set of program inputs which constitutes the partition. We show that such a partitioning technique has wide spread usages in performance prediction, testing, debugging and design space exploration.

Categories and Subject Descriptors C.3 [*Special-purpose and Application-based Systems*]: Real-time and embedded systems

General Terms Design, Performance, Verification

Keywords Cache Memories, Performance testing, Path Exploration, Symbolic Execution

1. Introduction

It is hard to build both *functionally correct* and *high performance* systems. For real-time and embedded software, it is often important to validate the system for certain non-functional properties, such as timing. Due to the huge amount of effort employed on the functionality validation of a software, the problem of performance validation is usually ignored. As a result, the deployed software may suffer from some serious performance bottlenecks. Such a loss of performance is often undesirable for real-time and embedded software, as most of such software are not just expected to produce a *correct* output, but also to produce a *correct* output within a *specified time bound.*

LCTES'13, June 20–21, 2013, Seattle, Washington, USA.

Memory subsystems, especially *caches*, have a significant impact on the performance of embedded software. In a typical memory subsystem, cache memory is several hundred times faster than the main memory. Therefore, a huge number of *cache misses* may lead to several magnitudes of performance degradation. Clearly, the performance of memory subsystem depends on the memory accesses made by the processor. On the other hand, the set of memory accesses made by the processor depends on the type of application it is running. As a result, the performance of an embedded system critically depends on the input provided to this specific application.

In this paper, we present a novel approach to partition the input domain of an application for validating performance. Such a partitioning strategy produces a performance spectrum of the entire program. Each partition in the spectrum is associated with a range of performance and a symbolic formula capturing the set of program inputs which constitutes the partition. In particular, we focus on the performance of the memory subsystem in this work, as cache misses are often the dominating factors for the performance degradation in embedded software. Although targeted towards embedded software, we believe that such a partitioning strategy could be useful for a variety of validation techniques.

To build a performance spectrum of the entire program, we face two significant challenges. The first problem appears due to the absence of any performance metric in a user program. Program behavior is usually captured by its input-output relationship. To overcome this problem, we instrument the program such that it computes a performance metric (in particular number of cache misses) when run on a particular input. Such an instrumentation is *entirely automatic* and it does not require any user annotations.

The second and more significant challenge appears in clustering the input domain with similar cache performance. It is clearly infeasible to execute a program for all possible inputs and measure the program performance for each of them. We therefore propose to *explore feasible path programs* (instead of *feasible program paths*) to partition the input domain of the program. A *path program* is a fragment of the original program where all the paths belonging to the same path program follow the same set of control flow edges, but may vary in the iterations of loops encountered. Therefore, a path program groups potentially *unbounded* number of paths together.

A crucial observation is that all the paths in a path program execute the same set of instructions (but may be in different number of times and in different order). It is possible that the ordering and frequency of different instructions may have a significant impact on the cache performance and therefore, the cache performance of different paths grouped into a path program may have a wide variation. However, we observed that a variation of this form mostly captures some serious cache performance issues, such as cache misses linearly increasing with the number of loop iterations due to *cache thrashing* (in the presence of small cache size or improper memory layout). As a result, path program creates a suitable abstraction for cache performance debugging. On the other hand, if the order-

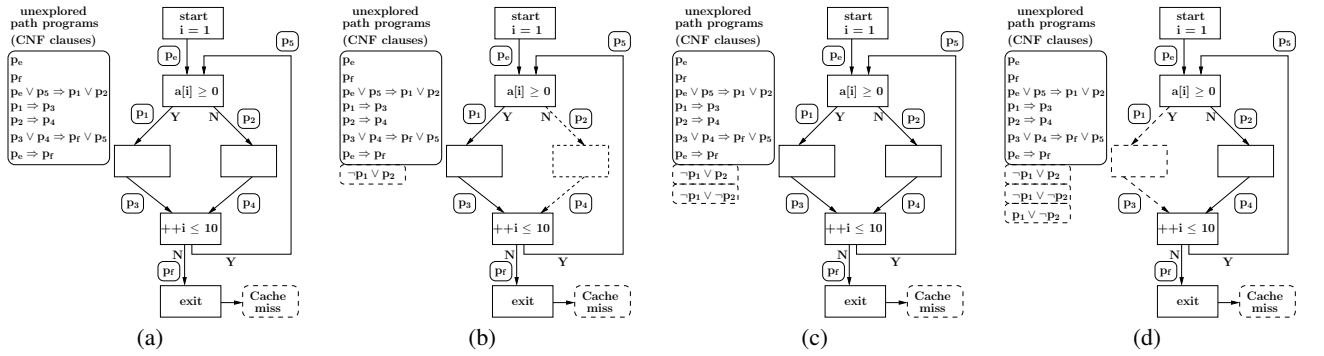

Figure 1. (a): Original program where array a is an input variable, (b)&(c)&(d): exploring different path programs, path program fragments are denoted by *thick and solid control flow edges*

ing and frequency of different instructions do not have a significant impact on cache misses, a path program can combine several paths having similar cache performance. Such a combination strategy is extremely useful for cache performance prediction and testing, as it partitions the input domain into a finite number of path programs. Each such path program corresponds to a set of inputs for which the program has very similar cache performance. As a result, instead of concentrating on the set of all inputs, we can only focus on the set of path programs produced by our framework. Moreover, the path program abstraction allows us to concentrate on a smaller part of the program at a time. Therefore, any analysis based on a path program is more *precise* and *scalable* than the same if applied on the entire program. We build a framework which dynamically explores path programs and each path program is analyzed only once during this exploration. Moreover, such analyses on path programs incrementally build the performance spectrum of the program. Therefore, at any time, the output from the set of already analyzed path programs captures a cache performance signature for a subset of the input domain (covered by the analyzed path programs).

The application of our framework can be summarized as follows. An immediate usage of our framework will be in *performance prediction*. For some *arbitrary input* provided to an application, we can locate the path program which captures the respective input and predict the performance of the application for the same input. The second significant usage is in *performance testing*. From the performance spectrum (*i.e.* the set of all path programs), we can generate concrete inputs which drive the execution of the program towards performance loss. Such critical test cases can often be missed by random testing. Thirdly, our framework can be used as a feedback to the compiler to perform *performance optimization*. Finally, as the application run on an embedded system is usually fixed, our framework can be used to decide an appropriate cache configuration for the system, as evidenced by our experiments.

We have evaluated our framework on several subject programs. Our experimental results show that we can achieve high accuracy in predicting performance as the different paths in a particular path program exhibit similar (and sometimes exactly same) cache behaviour. In addition, our experimental results also show that the different paths in a path program are similar in terms of overall execution time (and not just in terms of the number of cache misses). This is due to the reason that cache misses are often the key factors dominating the execution time of these programs.

2. Overview

In this Section, we shall give an outline of our overall technique using the example in Figure 1. The first problem in performance analysis appears due to the absence of any performance metric in a user program. To solve this problem, we instrument the original program to measure performance. Such an instrumentation is *entirely automatic*. In this paper, we focus on the cache performance of a program and for the time being, we shall assume that our instrumented program computes the cache performance (*i.e* number of cache misses) of the original program when run with a particular input (as shown in Figure 1(b)-(d)). In this context, we must mention that the instrumented program may itself suffer more number of cache misses (compared to the original program) due to the additional instrumented code. However, the instrumented program manipulates a variable $miss$ during execution. At the end of executing the instrumented program, the variable $miss$ *exactly* captures the number of cache misses suffered by the original program.

To partition the input domain with respect to cache performance, we propose to explore *feasible path programs* (instead of *feasible program paths*). A path program is defined as follows:

DEFINITION 2.1. *Assume $G = (V, E)$ is the control flow graph (CFG) of the program. Given an execution trace π, a path program P_π captures a subgraph $G_\pi = (V_\pi, E_\pi)$ of the control flow graph G, such that V_π is the set of basic blocks and E_π is the set of control flow edges executed in π.*

Intuitively, a path program is a subset of the original program, where all the paths belonging to the path program follow exactly same set of control flow edges, but may vary in the iterations of loops encountered. Depending on the underlying cache configuration, path program creates a suitable abstraction for cache performance debugging or cache performance prediction/testing. For very small cache sizes, different paths grouped into a path program may show wide variation in cache performance – such as cache misses linearly increasing with the number of loop iterations due to cache thrashing. On the other hand, for an appropriate cache configuration (*i.e.* in the absence of heavy cache thrashing), different paths grouped into a path program may show very similar cache performance – making the path program a suitable abstraction for cache performance testing and prediction. Finally, it is still possible to have a wide variation in *unavoidable* (*i.e.* cold) data cache misses for a single path program. However, such effects can easily be distinguished by instrumenting the cold data cache misses separately (*i.e.* not counting the cold data cache misses within variable $miss$).

The example in Figure 1(a) has 2^{10} different paths, but it has only three path programs, as shown in Figures 1(b)-(d). The path programs can be summarized by the following three symbolic formulae on input array a:

- Figure 1(b): $\mathcal{S}_1 \equiv a[1] \geq 0 \wedge \ldots \wedge a[10] \geq 0$.
- Figure 1(d): $\mathcal{S}_2 \equiv a[1] < 0 \wedge \ldots \wedge a[10] < 0$.
- Figure 1(c): $\mathcal{S}_3 \equiv \neg\mathcal{S}_1 \wedge \neg\mathcal{S}_2$.

Before any exploration, we first produce a SAT encoding of all the unexplored path programs. The basic intuition behind the SAT encoding is to capture the structure of a program control flow graph (CFG). We associate an atomic proposition p_e for each control flow edge e. p_e is *true* if control flow edge e is executed and *false* otherwise. The entire encoding is captured by a propositional formula in conjunctive normal form (CNF). Figure 1(a) shows the set of CNF clauses which encodes all the available path programs (for a formal presentation of this encoding, refer to Section 4.1).

During exploration, we first execute the test program with a random input and collect the execution trace π. In example 1(a), assume that only the left leg of the branch conditional `a[i] >= 0` appears in the execution trace π. From this execution trace, we construct the respective path program. Such a path program also contains the cache performance instrumentation, where a single integer variable $miss$ captures the number of cache misses suffered by the original program. As a result, bounding the value of variable $miss$ bounds the number of cache misses suffered by any execution which might visit *exactly the same set of control flow edges as* π. The path program is statically analyzed to compute the bounds on variable $miss$. Therefore, the analysis of the path program produces a range of cache misses of the form $MIN \leq miss \leq MAX$, where MIN and MAX represent the minimum and maximum number of cache misses, respectively, over the path program being analyzed. The analysis also computes the weakest invariant on input variables satisfied by any path in the path program (i.e. the condition $\mathcal{S}_1 \equiv a[1] \geq 0 \wedge \ldots \wedge a[10] \geq 0$ for the path program shown in Figure 1(b)). Any standard invariant generation method can be used for such analysis.

After the analysis of a path program, we add additional CNF clauses to the SAT encoding, so that we can explore a *different path program in subsequent iterations*. As shown in Figure 1(b), we add the clause $\neg(p_1 \wedge \neg p_2) \equiv \neg p_1 \vee p_2$ in the existing SAT encoding. This additional clause symbolically captures the information that *the path program in Figure 1(b) will not be explored in future*.

We manipulate the path conditions obtained from previous executions and try to deviate towards a different path program. Manipulation of a path condition involves negating the different branches along the path condition. Assume that we negate the 10-th branch in the initial execution trace (which results the path program in Figure 1(b)). Therefore, we get a partial path condition $\theta \equiv a[1] \geq 0 \wedge a[2] \geq 0 \wedge \ldots \wedge a[9] \geq 0 \wedge \neg(a[10] \geq 0)$. θ is satisfiable, however, we additionally need to check whether θ belongs to some already analyzed path program. From the execution trace obtained for Figure 1(b), we can find that both the legs of the conditional `a[i] >= 0` must be executed to satisfy θ. Therefore, both p_1 and p_2 must be *true*. Given $p_1 = p_2 = true$, if we find a satisfying assignment of the CNF clauses in Figure 1(b), then θ may belong to some *unexplored path program*. It turns out that a satisfying assignment of the CNF clauses (reported in Figure 1(b)) is possible with $p_1 = p_2 = true$. As a result, we execute the program on some input satisfying θ and analyze the respective path program. This path program is shown in Figure 1(c), which also adds the clause $\neg(p_1 \wedge p_2) \equiv \neg p_1 \vee \neg p_2$ to the SAT encoding. Now consider a partial path condition $\theta' \equiv a[1] \geq 0 \wedge a[2] \geq 0 \wedge \ldots \wedge \neg(a[9] \geq 0)$, after we explore and analyze the path program in Figure 1(c). Even though θ' is satisfiable, we know that it must execute both the legs of the condition `a[i] >= 0`. Therefore, both p_1 and p_2 must be *true* to satisfy θ'. However, there is no satisfying assignment for the CNF clauses reported in Figure 1(c) with $p_1 = p_2 = true$. As a result, we discard the partial path condition θ', since any path satisfying θ' has already been analyzed via the path program in Figure 1(c). We eventually explore the path program in Figure 1(d), which adds the clause $\neg(\neg p_1 \wedge p_2) \equiv p_1 \vee \neg p_2$ to the SAT encoding.

After analyzing the path program in Figure 1(d), our SAT encoder *blocks* all the three path programs. Note that the three path programs are blocked by the CNF clauses $\neg(p_1 \wedge \neg p_2) \equiv \neg p_1 \vee p_2$, $\neg(p_1 \wedge p_2) \equiv \neg p_1 \vee \neg p_2$ and $\neg(\neg p_1 \wedge p_2) \equiv p_1 \vee \neg p_2$, as shown in Figure 1(d). All of the three clauses are *satisfiable* if and only if $p_1 = false$ and $p_2 = false$. However, due to the structure of the CFG, we encode the CNF clause $p_e \vee p_5 \Rightarrow p_1 \vee p_2$. Therefore, if both p_1 and p_2 are *false*, both p_e and p_5 are *false* as well. This leads to a contradiction to all the CNF clauses reported in Figure 1(d), as p_e (which captures the single program entry) must be *true* for any path program (as denoted by the CNF clause p_e separately in the encoding). As a result, our exploration loop terminates at this stage, since we do not have any more *unexplored* path program.

It is important to note that the path program in Figure 1(c) does not cover the path programs in Figure 1(b) and Figure 1(d). The path program in Figure 1(c) takes both legs of `a[i] >= 0` conditional *at least once*. Therefore, the path program in Figure 1(c) groups $2^{10} - 2$ different paths.

3. Cache performance instrumentation

Given a program $\mathcal{P}$, we annotate the program to compute the number of cache misses suffered by $\mathcal{P}$. Let us assume $\mathcal{P}_{miss}$ is the annotated program. $\mathcal{P}_{miss}$ depends on the underlying cache parameters, namely, *number of cache sets*, *cache line size*, *cache associativity* and the *cache replacement policy*. The main advantage of such a code instrumentation technique is that it can easily be changed to handle a variety of cache architectures. The primary goal of such instrumentation is to integrate a *light-weight cache model* inside the original program, rather than just measuring the cache performance for a particular execution. Such an integrated cache model is used to compute the bound on cache misses via static analysis.

The annotated program $\mathcal{P}_{miss}$ captures the number of cache misses suffered by $\mathcal{P}$ via a single global variable $miss$. It is important to note that the annotated program $\mathcal{P}_{miss}$ may itself suffer more number of cache misses than $\mathcal{P}$ due to the additional instrumented code. However, during execution, $\mathcal{P}_{miss}$ manipulates a global variable $miss$ in such a fashion that the final value of $miss$ exactly captures the number of cache misses suffered by $\mathcal{P}$. The relation between the original program $\mathcal{P}$ and the instrumented program $\mathcal{P}_{miss}$ can be formalized via the following property:

PROPERTY 3.1. *For a particular cache configuration $\mathcal{CF}$, let us assume that $\mathcal{CM}$ is the number of cache misses suffered by $\mathcal{P}$ for some input combination $\mathcal{I}$. For the same cache configuration $\mathcal{CF}$, assume that $\mathcal{P}_{miss}$ is the instrumented version of program $\mathcal{P}$. If $\mathcal{CM}'$ denotes the value of variable $miss$ at the end of executing $\mathcal{P}_{miss}$ on input combination $\mathcal{I}$, then $\mathcal{CM} = \mathcal{CM}'$.*

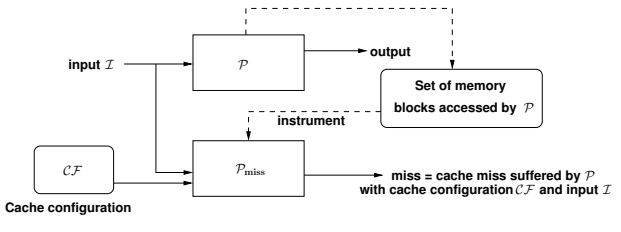

Figure 2. Cache performance instrumentation

Figure 2 shows the key relation (captured by Property 3.1) between $\mathcal{P}$ and $\mathcal{P}_{miss}$ in our proposed framework.

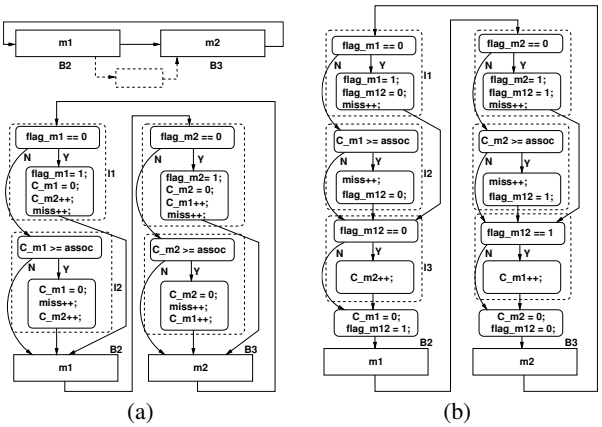

Figure 3. Instrumentation for (a) FIFO cache replacement policy, (b) LRU cache replacement policy.

In the following, we shall show the code annotation technique for two widely used cache replacement policies - *first-in-first-out* (FIFO) and *least recently used* (LRU).

Figure 3(a) shows the instrumentation for FIFO cache replacement policy. The original CFG is shown at the top of Figure 3(a). For the sake of simplicity, we shall illustrate the instrumentation of the path program which contains basic blocks $B2$ and $B3$. For the sake of illustration, let us assume that basic block $B2$ accesses memory block $m1$, basic block $B3$ accesses memory block $m2$ and $m1$ conflicts with $m2$ in the cache. In general, the cache conflict pattern of different memory blocks can be determined statically from the respective cache configuration (*i.e.* number of cache sets and cache line size). The sole purpose of the code instrumentation is to compute the number of cache misses faced by the original code. A cache miss can happen for the following reasons: i) *Cold cache miss* happens when a memory block is accessed for the first time, and ii) *Conflict miss and capacity miss* happen when the number of cache conflicts faced by a memory block exceeds the cache associativity. The total number of cache misses is captured by a variable $miss$ as shown in Figure 3(a). Block $I1$ captures the cold cache miss suffered by memory block $m1$ and block $I2$ updates the number of conflict and capacity misses suffered by $m1$. Variables C_m1 and C_m2 serve the purpose of counting cache conflicts to memory block $m1$ and $m2$, respectively and $assoc$ represents the associativity of the cache. $flag\_m1$ and $flag\_m2$ are used to distinguish the first accesses to $m1$ and $m2$, respectively.

Nevertheless, to update the number of conflict and capacity misses, we need to update the cache conflicts faced by each memory block at appropriate places. Updating such a cache conflict depends on the underlying cache replacement policy. In FIFO replacement policy, cache conflict to a memory block is increased whenever a new memory block enters the same cache set. In 3(a), since $m1$ and $m2$ conflict in the cache, the conflict count to $m2$ (*i.e.* C_m2) is increased whenever $m1$ makes a new entry to the cache (*i.e.* for all types of cache misses faced by $m1$). The annotated code for memory block $m2$ is entirely symmetric to the code inserted for memory block $m1$.

Assuming that the loop in Figure 3(a) executes 100 times, the value of $miss$ will be 200 for a direct mapped cache (*i.e.* $assoc$ is 1 in Figure 3(a)) at the end of execution. On the other hand, for a 2-way set-associative cache (*i.e.* $assoc$ is 2 in Figure 3(a)), value of $miss$ would be 2 at the end of execution. In both the cases, value of $miss$ capture the number of cache misses suffered by the original code (*i.e.* the path program containing basic blocks $B2$ and $B3$), hence satisfying Property 3.1.

Updating the cache conflict is slightly different in LRU compared to FIFO. In FIFO replacement policy, the state of the cache does not change on a *cache hit*. On the other hand, for LRU replacement policy, each memory block becomes the *most recently used* (*i.e.* its cache conflict is reset to zero) whenever it is accessed. Therefore, for any access of a memory block, we set its cache conflict count to zero (*e.g.* in Figure 3(b), C_m1 is set to zero before accessing $m1$). Moreover, accessing a memory block m increases the conflict to all the memory blocks which were *more recently used* than m before the access. In Figure 3(b), $flag\_m12$ tracks which memory block is *more recently used*, $m1$ or $m2$. $flag\_m12$ is set to *one* if $m1$ is more recently used than $m2$ and *zero* otherwise. Clearly, if $m1$ is not in the cache, we can always assume that $m1$ is the least recently used memory block. Therefore, for all types of cache misses of $m1$, we set $flag\_m12$ to zero. A symmetrical transformation is also applied before accessing $m2$. Finally, we increment the conflict to $m2$ (as shown by block $I3$ in Figure 3(b)) if and only if $m2$ was *more recently used* than $m1$ before the current access of $m1$ (*i.e.* $flag\_m12$ is set to zero).

In general, our code instrumentation technique traverses all the basic blocks of $\mathcal{P}$, computes the set of memory blocks accessed therein and it inserts additional code (as shown in Figures 3(a)-(b)) for each memory block at each *control flow edge* of $\mathcal{P}$. The essence of such a transformation is to integrate a cache model inside the original program $\mathcal{P}$. As shown in Figures 3(a)-(b), such a modeling of cache has been accomplished via the manipulation of a variable $miss$. Therefore, statically bounding the value of $miss$ directly gives a bound on the number of cache misses suffered by $\mathcal{P}$.

Challenges to handle data accesses For measuring data cache performance, the basic structure of the instrumentation is exactly the same as described in the preceding. However, it is worthwhile to note that statically estimating the set of memory blocks for a data access is very challenging. Existing research on address analysis [6] has looked at the problem of estimating an over-approximation of memory blocks accessed by each data reference. Since our instrumentation walks through the static control flow graph, it also needs to know the set of memory blocks accessed by each data reference. Once such an address analysis is performed, we can use the set of computed memory blocks by address analysis to instrument the code.

In our current implementation, however, we do not handle complex data memory accesses through pointers. Moreover, as our instrumentation statically needs to know the set of memory blocks, we currently do not handle dynamic memory allocations. Our framework currently handles accesses to scalar variables and static arrays. Array accesses inside a loop captures a special case. Note that different elements of an array can be accessed in different iterations of a loop. This is different from instruction memory block accesses, as every iterations of a loop access the same memory block for a specific instruction. As a result, we can add the instrumentation of an instruction memory block independent of loop iteration values (as shown in Figure 3).

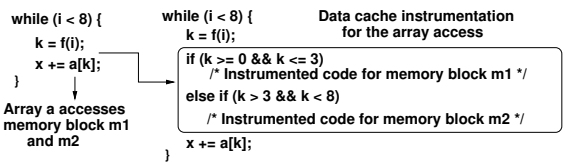

Figure 4. Instrumentation for array accesses

The minor change required for array accesses is shown in Figure 4. Assume that array elements $a[0 \ldots 3]$ accesses memory block

$m1$ and array elements $a[4 \ldots 7]$ accesses memory block $m2$. Such information can be computed from the base address of array a. The instrumentation for the array access is shown in Figure 4. Before the access, we check the bound of the array index. If the array index (*i.e.* k) is between 0 and 3, we add the instrumentation code for memory block $m1$ (exactly in the same fashion as shown in Figure 3). Similarly, if $k \in [4, 7]$, we add the instrumentation code for memory block $m2$. Note that the important difference is made by the conditional instrumentations. Such conditional checks are performed to compute the specific memory blocks being accessed from an array. The instrumentation is *entirely automatic*.

4. Analysis framework

Broadly, our input domain partitioning framework consists of two different steps: symbolic encoding of different path programs (Section 4.1) and systematic exploration of path programs to partition the input domain with respect to cache performance (Section 4.2). The symbolic encoding is used as an *oracle* to the path program exploration, so that it analyzes a path program only once.

4.1 SAT encoding

The SAT encoder symbolically encodes all the *unexplored path programs*. Due to the significant progress in SAT solver technologies, it is shown in [16] that such a symbolic encoding makes the path program enumeration feasible in practice. However in [16], all loops are treated in a *monolithic* fashion. More precisely, two path programs $P_\pi = (V_\pi, E_\pi)$ and $P'_\pi = (V'_\pi, E'_\pi)$ are distinguished in [16] if and only if there exists an edge e such that

- $e \in E_\pi$, $e \notin E'_\pi$ and e is a control edge outside of any loop, or
- $e \in E'_\pi$, $e \notin E_\pi$ and e is a control edge outside of any loop.

Therefore, to encode the notion of path program abstraction used in our framework, we extend the symbolic encoding used in [16] as follows. For each control flow edge e in the program, we introduce an atomic proposition p_e. The truth value of p_e captures the execution of control flow edge e. p_e is *true* if control flow edge e is executed, and *false* otherwise. We distinguish among the following two types of clauses: MSCC clauses and loop clauses.

MSCC clauses: These are the clauses proposed in [16]. MSCC clauses are created on *the maximal strongly connected decomposition* (MSCC) of the program control flow graph (CFG). In the MSCC decomposition of a CFG, a control flow is represented only across two different MSCCs. All other control flows inside an MSCC are hidden. Therefore, each control flow edge in the MSCC decomposition can be executed at most once. Assume E_D represents the set of control flow edges in the MSCC decomposition. Therefore, E_D includes only the set of control flow edges which do not appear inside any loop or recurrences. Without loss of generality, let us assume that n_e (n_f) represents the designated *entry* (*exit*) node of the program. We generate the following clauses:

$$onlyOne(out(n_e)),\ onlyOne(in(n_f)) \tag{1}$$

$$\bigwedge_{e \in E_D, src(e) \neq n_e} p_e \Rightarrow onlyOne(in(src(e))) \tag{2}$$

$$\bigwedge_{e \in E_D, end(e) \neq n_f} p_e \Rightarrow onlyOne(out(end(e))) \tag{3}$$

where $src(e)$ and $end(e)$ represent the source and target node of a directed control flow edge e. On the other hand, $in(n)$ and $out(n)$ represent the set of predecessors and successors of node n, respectively. $onlyOne(E)$ is a propositional formula which denotes that *exactly one* control flow edge in E can be executed. Therefore, $onlyOne(E)$ is defined as follows:

$$onlyOne(E) \equiv \bigvee_{e \in E} p_e \wedge \bigwedge_{e,f \in E. e \neq f} p_e \Rightarrow \neg p_f \tag{4}$$

Loop clauses: Assume that b is a basic block appearing inside some loop. Since b is inside a loop, we can no longer say that *exactly one* predecessor or successor can appear in the execution. However, we can generate the following clause to distinguish the execution of b:

$$\bigvee_{e \in in(b)} p_e \Leftrightarrow \bigvee_{e' \in out(b)} p_{e'} \tag{5}$$

where $in(b)$ and $out(b)$ represent the set of predecessors and successors of node b, respectively. Intuitively, the above clauses ensure that some successor of b can be executed if and only if some predecessor of b is executed.

Similarly, for any loop l, we generate the following clauses:

$$\bigvee_{e \in in(l)} p_e \Leftrightarrow \bigvee_{e' \in out(l)} p_{e'} \tag{6}$$

where $in(l)$ and $out(l)$ represent the set of entry and exit control flow edges of loop l, respectively. The above clauses denote that if a loop l is entered, it will exit eventually. These clauses also capture the fact that every loop occurring in the program is *bounded*.

Clauses in Equation 5 and Equation 6 are introduced by us to distinguish the different control flow edges inside a loop.

4.2 Dynamic exploration of path programs

Our approach iteratively explores different *path programs*. In each iteration, a new test input is generated that may force the execution through an unexplored path program. This process continues until we explore all the *feasible path programs*.

The basic idea behind our exploration algorithm is as follows. We first run the program for a random test input and collect the execution trace. We then construct a *path program* from this execution trace. The path program is analyzed to produce a cache miss range - meaning the analysis computes the lower bound and the upper bound on the number of cache misses suffered by the path program. To continue with the exploration process, we need to generate an input that may deviate the execution towards a different path program. In a broader perspective, therefore, the path program exploration process needs to perform two different tasks: first, generation of different inputs through SMT-based constraint solvers. Such an input generation involves manipulating and solving path conditions from previous executions. Secondly, before generating an input from a path condition, we need to check whether the path condition belongs to some explored path program. Such a checking is performed by *satisfiability testing* via a propositional formula, which in turn encodes the set of *all unexplored path programs*.

Algorithm 1 captures the core of our dynamic path program exploration technique. As we mentioned in the preceding, we use the SMT-based constraint solvers to generate different inputs. Moreover, we use a propositional formula to track the set of unexplored path programs. At any point of time, Φ encodes the set of of all unexplored path programs. Assume that the executed path for a test input τ is π and the corresponding path condition is $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k$. We want to find a test input τ' which deviates the execution from π and also walks through an *unexplored path program*. The deviation is made by negating any of the branch conditions appearing in π. If we want to deviate the execution at the r-th branch, the inputs to the program must satisfy the partial path condition $\theta \equiv \psi_1 \wedge \ldots \wedge \psi_{r-1} \wedge \neg\psi_r$. If θ is *unsatisfiable*, then θ resembles an *infeasible path* in the program and we discard it immediately. However, even if θ is *satisfiable*, it may walk through some previously explored path program. Therefore, we need to check whether

Algorithm 1 Dynamic exploration of path programs

Input:
$\mathcal{P}$, $\mathcal{P}_{miss}$: original and instrumented program
Output:
A set of *feasible* and *analyzed* path programs

$AllPc = unexplored =$ empty
/*build a SAT encoding of the entire program $\mathcal{P}$ */
$\Phi \leftarrow$ SATEncode($\mathcal{P}$)
select a random input τ
ExecuteAndAnalyze($\mathcal{P}$,τ,Φ)
while $unexplored \neq$ empty $\wedge \Phi$ is satisfiable **do**
    select $\varphi \in unexplored$
    remove φ from $unexplored$
    let $\varphi \leftarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{r-1} \wedge \psi_r$
    $\theta \leftarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{r-1} \wedge \neg\psi_r$
    $\{b_1, \ldots, b_k\} \leftarrow$ set of control flow edges in $\mathcal{P}$ that are executed by any path satisfying θ
    $\eta \leftarrow \Phi \wedge p_{b_1} \wedge \ldots \wedge p_{b_k}$
    /* analyze only an unexplored path program */
    if η and θ are satisfiable **then**
        $t_\theta \leftarrow$ some concrete inputs satisfying θ
        ExecuteAndAnalyze($\mathcal{P}, \mathcal{P}_{miss}$,$t_\theta$,$\Phi$)
    end if
end while

procedure EXECUTEANDANALYZE($\mathcal{P}$,$\mathcal{P}_{miss}$,τ,Φ)
    execute $\mathcal{P}$ on input τ
    let $\varphi \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k$ be the path condition
    /*build all partial path conditions */
    for $i \leftarrow 1, k$ **do**
        let $\varphi_i \leftarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$
        if $\varphi_i \notin AllPc$ **then**
            $AllPc \bigcup = \varphi_i$
            $unexplored \bigcup = \varphi_i$
        end if
    end for
    let π be the executed path on input τ
    $\{b_1, \ldots, b_m\} \leftarrow$ set of branch edges that appears in π
    $\{b'_1, \ldots, b'_n\} \leftarrow$ set of branch edges in $\mathcal{P}$ that does not appear in π
    $\xi \leftarrow p_{b_1} \wedge \ldots \wedge p_{b_m} \wedge \neg p_{b'_1} \wedge \ldots \wedge \neg p_{b'_n}$
    /* analyze only an unexplored path program */
    if $\Phi \wedge \xi$ is satisfiable **then**
        /*construct path program from execution trace π and the instrumented program $\mathcal{P}_{miss}$ */
        $P_\pi \leftarrow$ ConstructPathProgram($\pi, \mathcal{P}_{miss}$)
        /*analyze path program P_π */
        AnalyzePathProgram(P_π)
        /*block path program P_π for further exploration*/
        $\Phi \leftarrow \Phi \wedge \neg\xi$
    end if
end procedure

all the paths resulting from the partial path condition θ belong to some explored path program. Let us assume that $\{b_1, \ldots, b_k\}$ is a common set of control flow edges that must be executed by any path satisfying the partial path condition θ. This set of control flow edges can easily be computed from the execution trace π. Assume that $p_{b_1}, \ldots, p_{b_k}$ are the respective set of edge predicates used for the control flow edges $b_1, \ldots, b_k$ in the SAT encoding of $\mathcal{P}$. The formula $\eta \equiv \Phi \wedge p_{b_1} \wedge \ldots \wedge p_{b_k}$ is *unsatisfiable* only if each program path executing the set of control flow edges $\{b_1, \ldots, b_k\}$ has

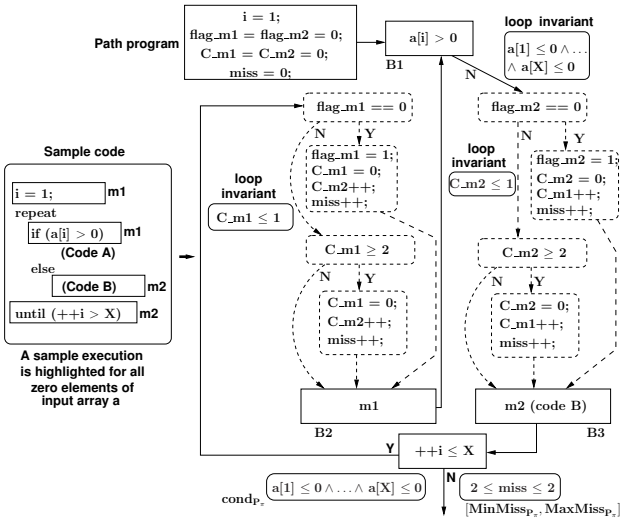

Figure 5. Analysis of a path program, array a is the input

been covered by an explored path program (recall that Φ encodes all the unexplored path programs). As a result, if η is unsatisfiable, we do not explore any path resulting from the partial path condition θ. In case, both θ and η are satisfiable, we generate a new test input from θ and try to deflect towards an *unexplored path program*.

Constructing a path program A *path program* is constructed from a particular execution trace π of $\mathcal{P}$ and the instrumented program $\mathcal{P}_{miss}$. Assume that B_π denotes the set of executed basic blocks. A path program $P_\pi = (V_\pi, E_\pi)$ includes the set of basic blocks B_π and additionally, it includes the following basic blocks:

- If the control flow (B_i, B_j) appears in π, then $\mathcal{I}_{B_i \rightarrow B_j} \in V_\pi$, where $\mathcal{I}_{B_i \rightarrow B_j}$ is the set of all instrumented basic blocks (in $\mathcal{P}_{miss}$) inserted along the edge (B_i, B_j).

As an example, consider the instrumented program fragment of Figure 3(a). If $(B2, B3)$ and $(B3, B2)$ both appears in π, the constructed path program includes all the eight instrumented basic blocks inserted along the edges $(B2, B3)$ and $(B3, B2)$.

Checking an explored path program Assume that we want to check whether a path π belongs to some *explored path program*. Further assume $\{b_1, \ldots, b_m\}$ is the set of branch edges that appears in the execution trace π and $\{b'_1, \ldots, b'_n\}$ is the set of branch edges that does not appear in the execution trace π. To check whether π has already been explored by a path program, we check the *satisfiability* of the following formula:

$$\Phi \wedge (p_{b_1} \wedge \ldots \wedge p_{b_m} \wedge \neg p_{b'_1} \wedge \ldots \wedge \neg p_{b'_n}) \quad (7)$$

Recall that $p_{b_1}, \ldots, p_{b_m}, p_{b'_1}, \ldots, p_{b'_n}$ are the set of atomic propositions (used by the SAT encoder) introduced for the branch edges $b_1, \ldots, b_m, b'_1, \ldots, b'_n$, respectively. Since, Φ is used to keep track of all the *unexplored path programs*, the above formula must be *unsatisfiable* if π has already been covered by a *previously explored path program*.

Analysis of a path program The main purpose of the instrumented program $\mathcal{P}_{miss}$ was to enable static analysis on *feasible path programs*, which are iteratively explored using Algorithm 1. The primary goal of the static analysis is to compute a sound lower bound and a sound upper bound on cache misses suffered by each path program. Additionally, the static analysis computes a symbolic formula on the input variables that must be satisfied by any

path constituting the respective path program. It is worthwhile to note that a traditional profiler cannot guarantee any bound on cache misses. Moreover, a traditional profiler computes information only for a set of representative inputs. Unlike a traditional profiler, our framework systematically partitions the input domain of a program via the exploration of feasible path programs and guarantees the bound on cache misses for each path program via static analysis.

Recall that a path program P_π is constructed from the instrumented program $\mathcal{P}_{miss}$ and a feasible execution trace π. The static analysis on a path program P_π computes a triplet $\langle cond_{P_\pi}, MinMiss_{P_\pi}, MaxMiss_{P_\pi} \rangle$ which has the following interpretation:

- $cond_{P_\pi}$: a symbolic formula on the input variables, where any feasible path in P_π must satisfy $cond_{P_\pi}$.
- $MinMiss_{P_\pi}$: minimum number of cache misses suffered by any path in the path program P_π.
- $MaxMiss_{P_\pi}$: maximum number of cache misses suffered by any path in the path program P_π.

To compute the triplet $\langle cond_{P_\pi}, MinMiss_{P_\pi}, MaxMiss_{P_\pi} \rangle$, we use standard invariant generation methods (*e.g.* polyhedra [11], symbolic execution [1] etc.). Procedure `AnalyzePathProgram` (in Algorithm 1) captures the invariant generation.

Figure 5 demonstrates a sample code and the constructed path program P_π for an execution with all zero elements of input array a. In Figure 5, $m1$ and $m2$ capture the respective memory blocks accessed by the code. For the sake of simplicity in this example, we only show the path program with the instrumentation for instruction caches. However, as discussed in Section 3, our framework can be used for both instruction and data caches. The core of path program analysis is to reason about the value of variable $miss$ (as shown in Figure 5), which derives $MinMiss_{P_\pi}$ and $MaxMiss_{P_\pi}$. This in turn requires generating *invariants* involving the variable $miss$. In the example shown in Figure 5, memory blocks $m1$ and $m2$ conflict in the cache. The two loop invariants $C\_m1 \leq 1$ and $C\_m2 \leq 1$ capture the fact that both the memory blocks $m1$ and $m2$ face at most one cache conflict within the loop. These two loop invariants ensure that the true leg of both the branches $C\_m1 \geq 2$ and $C\_m2 \geq 2$ can never be executed (since both the formulae $C\_m1 \geq 2 \wedge C\_m1 \leq 1$ and $C\_m2 \geq 2 \wedge C\_m2 \leq 1$ are *unsatisfiable*). As a consequence, the value of variable $miss$ can be incremented exactly twice – once at the *true* branch of $flag\_m1 == 0$ and the other at the *true* branch of $flag\_m2 == 0$. Therefore, the value of variable $miss$ is bounded by $2 \leq miss \leq 2$ at the exit of the loop (irrespective of the loop bound). Such an invariant on variable $miss$ captures the fact that $m1$ and $m2$ only face *cold cache misses* for *any possible execution* of P_π. The analysis also generates an invariant (on input array a) $cond_{P_\pi} \equiv a[1] \leq 0 \wedge \ldots \wedge a[X] \leq 0$, which holds over path program P_π.

Currently, our framework requires a numeric upper-bound on the inputs which directly or indirectly (via a chain of data dependencies) affect loop bounds. As a consequence, the computed invariants on cache misses are always of the form $C_1 \leq miss \leq C_2$, where both C_1 and C_2 are constants. In the presence of parametric (or symbolic) loop bounds, the computed cache miss range might be parametric in terms of the input dependent loop bounds. For the example shown in Figure 5, if the cache is a direct-mapped cache and both m_1 and m_2 map to the same cache set, such a parametric cache miss range will be of the form $2X \leq miss \leq 2X$ (considering X as an input). Lifting the numeric performance range to a parametric range is a subject of our future work.

Termination Our exploration process terminates when the SAT encoder blocks all possible path programs (*i.e.* Φ becomes *unsatisfiable*). The exploration also terminates when Φ remains *satisfiable*, but all *unexplored* path conditions are visited (*i.e.* $unexplored$ becomes empty). Such a situation may arise in the presence of *infeasible path programs*, as we only explore *feasible path programs*.

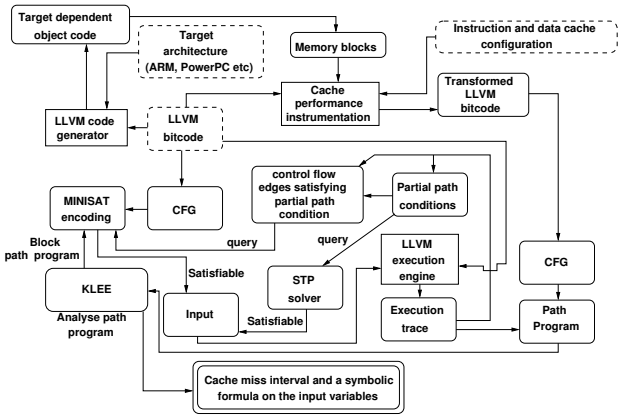

Figure 6. Implementation framework

5. Implementation

Figure 6 shows the outline of our implementation framework. We use the LLVM compiler infrastructure [2] as a baseline of our implementation. Individual benchmarks are compiled into LLVM bitcode format and their control flow graphs are extracted from the LLVM bitcode. This control flow graph is given as an input to our analysis framework. We first use the LLVM code generator to generate the object code for a specific target architecture (*e.g.* ARM, PowerPC). From the generated object code, the accessed memory blocks are extracted and they are mapped appropriately to the respective basic blocks at the LLVM bitcode level. The cache performance instrumentation is accomplished by augmenting the original control flow graph (CFG) with additional basic blocks (as explained in Section 3). Such an instrumentation accepts the set of memory blocks accessed in the target binary and a specific instruction and data cache configuration. As a result, our framework can be parameterized with respect to different target architectures and cache configurations. The execution of the test program is achieved through LLVM execution engine, from which we also collect the *basic block level execution trace* and the respective *path condition*. To manipulate and solve different path conditions, we use the *STP solver* [4]. We use the *minisat* satisfiability solver [3] to track the set of *unexplored path programs*. For analyzing a path program, any standard invariant generation method can be applied. In our current implementation, we modify LLVM-based *KLEE symbolic execution engine* [1] to generate invariants on the number of cache misses suffered by the original program (*i.e.* generating invariants on the variable $miss$ as explained in Section 4.2). In general, KLEE performs a symbolic execution of the entire program. We modify the source code of KLEE to selectively analyze a path program, and thereby making a single invocation of KLEE much faster than usual. Each analysis by KLEE produces a cache miss interval and a symbolic formula on the input variables.

6. Evaluation

In this section, we shall evaluate our framework with different subject programs. Some salient features of these subject programs are listed in Table 1. Our proposed framework aims to partition the input domain with respect to cache performance. *Such a framework is mostly suitable for an application which exhibits varying*

Program	Lines of C code	Object code size (ARM)	Basic blocks
Papabench [14]	592	14240 bytes	311
Sha [15]	236	6104 bytes	77
Susan [15]	260	4172 bytes	82
JetBench [21]	766	33872 bytes	922
Nsichneu [5]	4255	49908 bytes	754

Table 1. Subject programs used for evaluation

cache performances for the same input size, but different input values. Therefore, we try to find subject programs which can potentially show varying performances for the fixed input size but different input values. Such a program should potentially have many input-dependent branches in the control flow graph. As a result, our framework will explore multiple path programs capturing the different outcomes of such input dependent branches. The characteristics of the subject programs can be summarized as follows:

- Papabench is an *unmanned aerial vehicle* (UAV) controller program which performs the navigation and stabilization tasks of an aircraft. We use the autonavigation component of Papabench due to the presence of many input dependent paths in the program. The autonavigation component goes through different control locations. The control flow for a typical execution depends on the starting position of navigation and several input signals. Therefore, the autonavigation component has the potential to exhibit varying performances with respect to different input values.
- Susan is an image processing kernel which manipulates an image matrix. The value of different image pixels are checked multiple times and the program has several control paths depending on the value of pixel.
- JetBench is a hard real-time, simulation of jet engines. Depending on the value of input flight profile data, this program performs different thermo-dynamic calculations. Therefore, we try to categorize the input flight profile data with respect to the performance of JetBench.
- Nsichneu is a petri-net simulator, which has a huge number of input dependent branches. The program simulates a petri-net along different control paths depending on the marked positions (given as an input).
- Finally, we show a program Sha which does not have any input dependent branch. The performance of Sha depends only on the input size. For such programs, we can use our framework to see the change in performance with respect to the input size.

We use our evaluation framework to answer the following crucial questions related to the performance validation of a program.

- *Performance prediction:* Given an arbitrary input, can we predict the program's cache performance with respect to the input? During the testing of an embedded software, the real execution platform may not be available. Therefore, tools and techniques to predict performance in the absence of real execution platform are crucial for embedded software.
- *Performance testing:* Can we synthesize inputs which may force the program to suffer heavy number of cache misses?
- *Performance debugging:* Can we replay some useful information to the user/compiler to pinpoint the reason of serious cache performance issues (*e.g.* cache thrashing)?
- *Design space exploration:* Can we decide the appropriate execution platform (*e.g.* cache configuration) for a particular application so that it meets certain timing guarantees?

Key result Table 2 reports a summary of our experiments. We use a 2-way associative, instruction and data cache with 16 bytes of cache line size and FIFO replacement policy. The results are reported for a 4 KB instruction and 4 KB data cache. The cache sizes are chosen in such a fashion so that they exhibit sufficient amount of conflicts in instruction and data caches. In Table 2, we also report the overall execution time of a path program. To compute the overall execution time, we add the total instruction and data cache miss penalty suffered by all the cache misses with the computation cost of all the executed instructions. Both instruction and data cache miss penalty is taken as 100 CPU cycles. We perform all experiments on an Intel i7-core processor having 8 GB of RAM and running Ubuntu 10.04 operating system. The computation time in Table 2 reports the total time taken by our framework for each subject program; including the cache performance instrumentation, path program exploration, execution of the program and path program analysis.

Due to space constraints, we cannot report the cache performance of all the path programs. However, Table 2 reports the cache miss range of maximum variation for each subject program. Four rightmost columns of Table 2 capture the *feasible path program* where the performance range (*i.e.* the interval for instruction cache miss, data cache miss and overall execution time) has maximum variation. The maximum variation in the execution time of a feasible path program is shown in the last column. For an interval $[min, max]$, the variation is computed as $\frac{max-min}{min} \times 100\%$. Note that the reported variation is reasonably short (maximum variation of 20% in Susan where the absolute numbers of cache misses have small values). As a result, we can observe that path program is a suitable abstraction to combine several paths having similar cache performances.

Currently, our framework only computes an absolute range of cache misses for all feasible path programs. It is, in general, possible that the value of cache miss depends on the bound k of some loop, where k is an input (*e.g.* input size). As a result, the computed cache miss range could be parametric in terms of such input variable k. In our current implementation, we do not handle such parametric cache miss expressions and our framework requires the bound on such input variables k.

For fixed input size, Sha does not have any input dependent program branches. Therefore, the performance of Sha is *independent* of input values and our framework produces exactly one path program for Sha (for an input size of 8 KB). Note that program Nsichneu reports *constant* cache performance for the explored (*i.e.* feasible) path programs. Similarly, Susan shows constant data cache performance for each *feasible path program*. Such constant cache performance can be generated when each explored (*i.e.* feasible) path program satisfies at least one of the following two conditions: i) the path program contains exactly one path, or ii) any path constituting the path program faces only *cold cache misses* (for all the subject programs in Table 2, cold cache miss is constant for a given path program and input size).

In the following discussion, unless otherwise stated, the reported cache miss corresponds to the *total number of cache misses*, including the number of *instruction cache misses* and *data cache misses* for the respective cache configurations.

Performance prediction Recall that the analysis of a path program p_i computes a triplet $\langle cond_{p_i}, MinMiss_{p_i}, MaxMiss_{p_i} \rangle$. $cond_{p_i}$ is a symbolic formula which captures the set of inputs along which path program p_i is reached and $MinMiss_{p_i}$ ($MaxMiss_{p_i}$) is the minimum (maximum) number of cache misses suffered by path program p_i. Assume that we have given an arbitrary input and we want to predict the performance of the program for this input. Given an arbitrary input, we can locate the path program p_i where the symbolic formula $cond_{p_i}$ is satisfied by the same input. Since we

Subject program	#partitions (#feasible path programs)	#infeasible path programs	Computation time (in seconds)	Instruction cache miss (maximum variation across any feasible path) program	Data cache miss (maximum variation across any feasible path) program	Execution time (CPU cycles) (maximum variation across any feasible path) program	Maximum variation in execution time
Papabench	20	4×10^8	158	[882,990]	[32,37]	[100851, 110279]	9.3%
Sha	1	15	20	[243,243]	[696,696]	[1130039,1130039]	0%
Susan	50	145622	163	[62,78]	[15,15]	[8109,9769]	20%
JetBench	7	2.7×10^9	250	[532,649]	[535,575]	[125003, 141213]	13%
Nsichneu	11	$> 2^{200}$	192	[9519,9519]	[196,196]	[1042732,1042732]	0%

Table 2. Path program partitioning of different subject programs. The reported cache miss range captures the *feasible path program* which exhibits *maximum variation* between the minimum and maximum number of suffered cache misses.

partition the input domain, there will be exactly one path program p_i for each such arbitrary input. The located path program p_i is also attached with its cache performance range. Therefore, our prediction will be the cache miss interval $[MinMiss_{p_i}, MaxMiss_{p_i}]$.

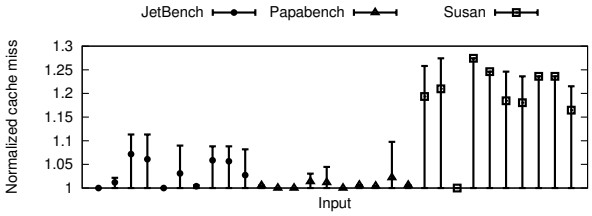

Figure 7. Cache performance prediction. The vertical bar captures the predicted cache miss interval and the point along each vertical bar captures the number of cache misses obtained by executing the program on the respective input

To check the correctness and accuracy of our prediction, we first run a subject program for a set of arbitrary inputs. For each input, we collect the number of cache misses actually suffered at the end of execution. Moreover, for each such arbitrary input, we locate the path program p_i where the symbolic formula $cond_{p_i}$ is satisfied by the same input. For each input i, we compare the three numbers – the minimum and maximum number of cache misses associated with the located path program p_i (*i.e.* $MinMiss_{p_i}$ and $MaxMiss_{p_i}$, respectively) and the actual number of cache misses (say $Cmiss_i$) obtained by executing the program on input i. Clearly, $MinMiss_{p_i} \leq Cmiss_i \leq MaxMiss_{p_i}$.

Figure 7 reports the result of cache performance prediction. For each input, the vertical bar captures the normalized cache miss interval $[1, \frac{MaxMiss_{p_i}}{MinMiss_{p_i}}]$ from our prediction and the point along each vertical bar captures the normalized number of cache misses $\frac{Cmiss_i}{MinMiss_{p_i}}$ after executing the program on the respective input. Figure 7 shows that we can obtain reasonably accurate prediction, as the length of the cache miss interval $[1, \frac{MaxMiss_{p_i}}{MinMiss_{p_i}}]$ is short. For Sha and Nsichneu, the cache miss range is *constant* for each path program and we always had *accurate* predictions. The results from Sha and Nsichneu are not reported in Figure 7.

Performance testing Our proposed framework has a significant usage in performance testing. Given the set of path programs produced by our framework, we can find the path program facing heavy cache misses or creating a severe performance bottleneck. Since such a path program is also associated with a symbolic formula on the input variables, we can use the SMT solver to generate a concrete input from this symbolic formula. Such performance stressing, concrete test inputs can be reported to the developer.

For Nsichneu, we found that two path programs experience cache misses several magnitudes (more than 10 times) higher than the other path programs. Similarly, for Papabench, we found that the path program suffering from heaviest cache misses satisfies an input value 2. Using our framework, we can quickly locate such performance stressing path programs (as shown in Table 2, there are only 11 and 20 path programs for Nsichneu and Papabench, respectively) and generate a concrete test input from the respective symbolic formula on the input variables.

Different path programs of Susan also produce diverse cache performances (with the lowest performing path program having 10 magnitudes higher cache misses than the highest performing path program). Susan manipulates a character matrix of a fixed size. It is impossible to test Susan for all such matrices. However, from the set of explored path programs, we can observe the symbolic formulae on the input matrices which stress the execution of Susan towards performance loss. Such observation will greatly help in the performance testing of Susan, as we can pick up the set of appropriate test cases from a representative testing pool.

Performance debugging In the preceding, we have discussed the use of our framework to generate performance stressing test cases. Along with performance stressing test cases, it is also useful to report the root cause of performance loss. Since we focus on cache performance, such a reporting of root cause needs to highlight the specific program locations/data accesses which are suffering from heavy cache misses.

We can easily tune our basic framework to highlight the potential root causes of performance loss. Instead of generating bound on the overall cache miss count, we can generate bound on the cache misses suffered at each cache set. Let us assume $miss_i$ captures the range of cache misses suffered at cache set i. By checking the value of each $miss_i$, we can locate the memory blocks mapped to cache set i and subsequently, trace the memory blocks back to the source code level statements.

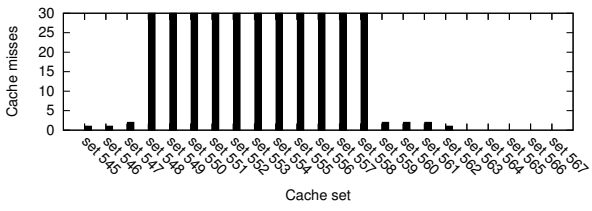

Figure 8. Cache performance debugging

Figure 8 captures a sample output for such modification. We have mentioned before that two path programs of Nsichneu suffer several magnitudes higher cache misses than the other path programs. Figure 8 shows a snapshot of the cache misses suffered by different cache sets. The snapshot is generated in such a fashion that it exhibits a few (but not all) cache sets suffering from relatively high cache misses. Since the total number of such cache sets

(*i.e.* the cache sets suffering from relatively high cache misses) may be arbitrarily high, the total number of cache misses experienced by the program is several magnitudes higher than the number of cache misses suffered by a single cache set. It is worthwhile to note that a traditional profiler may not able to highlight the root causes of such cache performance issues. A traditional profiler relies on a few training runs of a program. If the training runs do not go through the performance stressing path programs, the root causes of such cache performance issues will remain unknown. The cache size for our experiments (*i.e.* to generate Figure 8) is chosen such that there are no *capacity* misses. We can observe that a few cache sets (set 548-558) suffer relatively high cache misses, whereas other cache sets suffer a small number of cache misses (*e.g.* set 559-562) or zero cache misses (*e.g.* set 563-567). This irregularity of cache misses often appear due to the improper code/data layout in the program. We observed that the number of memory blocks mapped to any cache set starting from 548 to 558 is more than the associativity of the cache. Such memory blocks are accessed inside a loop, introducing *cache thrashing*. The information generated by our framework can be replayed back to a compiler. The compiler can utilize such information for optimizations such as *cache locking* (to selectively lock memory blocks in the cache and avoid cache thrashing) and *code positioning* (changing the layout of code/data to avoid cache thrashing).

Design space exploration Finally we show the application of our framework to choose an appropriate execution platform, specifically, the right cache configuration. Figure 9 shows the sensitivity of cache misses (we plot the maximum cache miss suffered over all the path programs) for two of our subject programs with respect to different cache configurations and target architectures. Since our framework can be parameterized with respect to cache configuration, we can produce such sensitivity graph by running the cache performance spectrum module multiple times. Figure 9 shows that an 1-way, 16 KB (1-way, 8 KB) cache should be chosen for `Papabench` targeting PowerPC (ARM), as the cache miss stabilizes beyond the particular cache configuration. In a similar fashion, a 4-way, 64 KB cache is appropriate for program `Nsichneu`. Note that a traditional profiler may not be able to find a training input to stress the execution of the program towards maximum cache misses. As a result, we might end up choosing an inappropriate cache using a traditional profiler. This is due to the reason that the maximum number of cache misses may not stabilize using the cache configuration chosen by a traditional profiler.

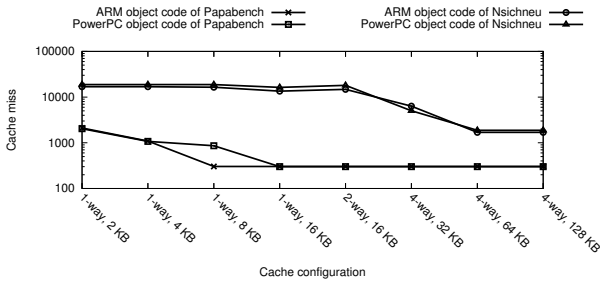

Figure 9. Cache miss sensitivity w.r.t. cache configuration

The performance of `Sha` only depends on the input size. Figure 10 shows the instruction and data cache miss sensitivity suffered by `Sha` for a direct-mapped, 1 KB cache. With respect to the input data size, we observe an exponential growth in the number of cache misses for a direct-mapped, 1 KB cache. Such an exponential growth in cache misses clearly captures a *cache thrashing* behaviour. Therefore, we can conclude that 1 KB cache is inappropriate for `Sha`. The instruction cache thrashing entirely disappears when using a two-way, 8 KB cache (as also evidenced by Figure 10). The growth in data cache misses with respect to input data size does not disappear with a two-way, 8 KB cache. However, we investigated that such growths in data cache misses are merely due to the increased *cold data cache misses* for increased input size. Therefore, we can conclude that a two-way, 8 KB cache can be used for `Sha` without an appreciable loss of performance.

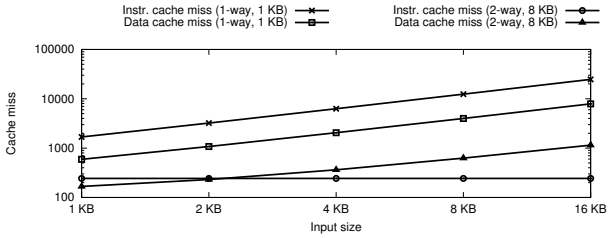

Figure 10. Cache miss sensitivity w.r.t. input size

7. Related work

Work on the performance validation of embedded software was started two decades ago. Previous approaches were mainly based on *static analysis*. Analysis of cache performance has been done, among others, in [23] via abstract interpretation. Recently, such abstract interpretation based cache analysis has been improved by a gradual and controlled used of model checking in [9]. Such analyses were performed on the entire program for computing its *worst case execution time* (WCET). On the other hand, our approach is orthogonal to these approaches. We partition the input domain of a program in terms of different path programs. Moreover, our partitioning strategy is dynamic in nature. We only explore a partition if there exists some feasible path inside it. A different work [20] uses an evolutionary algorithm to explore program paths for performance testing. However, [20] does not partition the input domain and it also does not guarantee to explore the entire input domain. The work in [12] employs the idea of pattern recognition to predict the whole program locality based on a few training runs of the program. Unlike our approach, the technique proposed in [12] does not guarantee to explore the entire input domain. Moreover, the technique proposed in [12] predicts program performance for a given input based on the past history/training-set (and the prediction may be wrong). On the other hand, as our approach is based on program analysis, we can always predict a *safe* lower bound and upper bound of performance for each of the explored partition.

Our idea is inspired by the recent advances in *program path exploration* and *satisfiability modulo theory* (SMT). *Directed* test generation [17] has made significant improvement over random testing in the past few years. Such test strategies attempt to cover *program paths* for testing. On the other hand, we are interested in building the performance footprint of the entire program. As there might be an unbounded number of paths in a program, a path-based search procedure is, in general, infeasible to build a performance footprint of the entire program. A recent work [22] has proposed to merge several paths for path-based testing. Two paths are merged if they have the same input-output relationship. However, our goal is *different* - we want to merge different paths having similar performance rather than merging paths with the same output expression.

In recent times, the idea of path program has been applied in [16] for path-sensitive functional verification. Our approach differs in two key aspects from this work: first, our goal is validating the

performance and secondly, the path program in our framework is constructed from an execution trace rather than from the static control flow graph. Therefore, we only analyze a path program that has at least one feasible execution.

The work in [18] proposes to find the computational complexity of a program automatically. Similarly, [8] proposes a new approach to automatically find test inputs for the worst-case computational complexity. Our work differs from several aspects from these two previous works: first, our notion of performance is based on the execution time rather than computational complexity. Secondly, the primary goal of our framework is to build a performance signature of the entire program by partitioning its input domain. Of course, such a partitioning can also be used to generate test inputs for worst-case performance, as evidenced by our experiments.

Several techniques for program profiling have been studied in the past few decades (such as [7, 19], among others). Such traditional profiling techniques can be used to analyze (compressed) execution traces (*e.g.* in [19]) for deriving *program hotspots*. In contrast to our approach, profiling techniques do not guarantee to cover the entire input domain, instead such profiling techniques rely on training inputs. Moreover, our approach can act as a complementary to compute the set of relevant inputs for program profiling.

Recent works [10, 24] have proposed to extend the traditional profiling technique by determining an empirical cost function. Such a cost function is found automatically and it captures an approximate cost of the program with respect to different inputs (and in particular, with respect to different input sizes). Since the approaches in [10, 24] are based on a few training runs of the program, they can only capture an approximate cost of the program for a fixed input size. On the other hand, since our approach is based on program analysis, we can provide a *sound* lower and upper bound of cache performance. Moreover, the work of [10, 24] do not guarantee to capture the performance signature of the entire program. Our approach does the same by partitioning the input domain, in the form of representative path programs.

There has been some recent work on detecting performance bugs in distributed systems [13]. Such work systematically generates random simulations to detect performance bugs. However, as the testing is based on random simulations, this approach cannot guarantee to build any performance footprint of the *entire* program.

8. Discussion

Summary In this paper, we have proposed an approach to partition the input domain of a program based on cache performance. Our partitioning is based on exploring feasible path programs. As evidenced by our experimental results, the path program abstraction is suitable for a variety of purposes, such as performance debugging (*e.g.* to detect *cache thrashing*), performance prediction and performance testing. Moreover, we have shown that our proposed framework can be used to decide an appropriate cache configuration for an application.

Limitations Our technique is most suitable for the programs which exhibit varying cache performances on different input values. Since our approach computes a sound lower and upper bound of cache misses by statically analyzing each path program, it has the general limitations faced by any static analysis technique. Such limitations include the handling of dynamic memory allocations and complex pointer aliasing. Besides, in our current implementation, the static analysis of path programs accounts to majority of overhead. However, we believe that we can use several matured and efficient techniques for invariant generations to reduce such overhead in future.

Future work Our work can be extended in several directions. In this paper, we have focused on cache miss metric. For the set of subject programs used in our evaluation, the number of cache misses dominates the performance. In future, we plan to study other performance metrics apart from the cache miss metric. Besides, our framework currently builds the cache performance signature for the uninterrupted execution of a single program. In the presence of multi-tasking, a high priority task and external interrupts may affect the cache content, leading to additional cache misses. We plan to extend our framework for multi-tasking systems in future.

Acknowledgements

We thank the anonymous reviewers for their comments and feedback. This work was partially supported by A*STAR Public Sector Funding Project Number 1121202007 - "Scalable Timing Analysis Methods for Embedded Software".

References

[1] KLEE Symbolic Virtual Machine. `http://klee.llvm.org/`.

[2] LLVM compiler infrastructure. `http://llvm.org/`.

[3] MINISAT solver. `http://minisat.se/Main.html`.

[4] STP Constraint Solver. `http://sites.google.com/site/stpfastprover/`.

[5] WCET benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[6] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.

[7] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, 1996.

[8] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, 2009.

[9] S. Chattopadhyay and A. Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *RTSS*, 2011.

[10] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *PLDI*, 2012.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[12] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, 2003.

[13] C. Killian et al. Finding latent performance bugs in systems implementations. In *FSE*, 2010.

[14] F. Nemer et al. Papabench: a free real-time benchmark. In *WCET Workshop*, 2006.

[15] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *WWC-4. 2001 IEEE International Workshop*, 2001.

[16] W. R. Harris et al. Program analysis via satisfiability modulo path programs. In *POPL*, 2010.

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[18] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[19] J. R. Larus. Whole program paths. In *PLDI*, 1999.

[20] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *RTSS*, 1998.

[21] M. Y. Qadri, D. Matichard, and K. D. McDonald-Maier. JetBench: An open source real-time multiprocessor benchmark. In *ARCS*, 2010.

[22] D. Qi, H. D. T. Nguyen, and A. Roychoudhury. Path exploration based on symbolic output. In *FSE*, 2011.

[23] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3), 2000.

[24] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI*, 2012.