Static Analysis Driven Cache Performance Testing

Abhijeet Banerjee Sudipta Chattopadhyay Abhik Roychoudhury
National University of Singapore
{abhijeet,sudiptac,abhik}@comp.nus.edu.sg

***Abstract*—Real-time, embedded software are constrained by several non-functional requirements, such as timing. With the ever increasing performance gap between the processor and the main memory, the performance of memory subsystems often pose a significant bottleneck in achieving the desired performance for a real-time, embedded software. Cache memory plays a key role in reducing the performance gap between a processor and main memory. Therefore, analyzing the cache behaviour of a program is critical for validating the performance of an embedded software. In this paper, we propose a novel approach to automatically generate test inputs that expose the cache performance issues to the developer. Each such test scenario points to the specific parts of a program that exhibit anomalous cache behaviour along with a set of test inputs that lead to such undesirable cache behaviour. We build a framework that leverages the concepts of both static cache analysis and dynamic test generation to systematically compute the cache-performance stressing test inputs. Our framework computes a test-suite which does not contain any *false positives*. This means that each element in the test-suite points to a *real cache performance issue*. Moreover, our test generation framework provides an assurance of the test coverage via a well-formed coverage metric. We have implemented our entire framework using Chronos worst case execution time (WCET) analyzer and LLVM compiler infrastructure. Several experiments suggest that our test generation framework quickly converges towards generating cache-performance stressing test cases. We also show the application of our generated test-suite in design space exploration and cache performance optimization.**

I. Introduction

Real-time, embedded software are required to satisfy several extra-functional properties, such as timing. Therefore, performance validation marks a crucial stage before certifying such time-critical software. In the absence of appropriate performance-validation techniques, the deployed software may suffer from severe performance problems, such as missing deadlines. For example, in the context of an *anti lock braking systems* (ABS), missing a deadline may lead to serious accidents, potentially costing human lives.

Due to the inherent gap between *processor* and *memory* performances, memory subsystems may significantly affect the performance of an embedded software. To reduce such effect, a fast cache memory is often employed between a processor and main memory. In a modern embedded processor, cache memories are several magnitudes faster than the main memory. Therefore, at any point in execution, the content of the cache memory significantly impacts the performance of the underlying embedded software. The content of a cache is managed at runtime and such content depends on the accessed memory block sequence. Since different inputs to the same application may follow different execution paths, the sequence of accessed memory blocks in an execution critically depends on the input provided to the application. As a consequence, the performance of caches (and hence the performance of an application) critically depends on the input provided to the underlying embedded software.

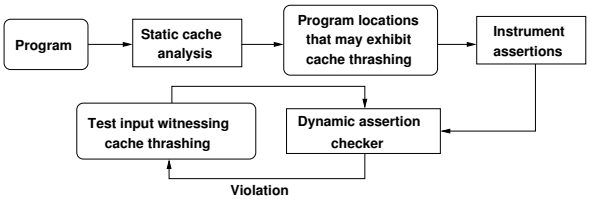


Fig. 1: Test generation framework

In this paper, we propose a novel approach to automatically generate test inputs that expose performance problems due to memory subsystems. In particular, we generate test inputs to automatically detect performance stressing memory access sequences. Such *poor* memory access sequences are *undesirable*, as they may lead to critical cache performance issues, specifically *cache thrashing* at runtime. We propose a test generation framework that aims to report cache thrashing scenarios that exist in some program execution. Each element in our report contains a *unique* cache thrashing scenario and a symbolic formula capturing the set of inputs that expose the issue in a program execution.

However, the generation of cache-performance stressing test inputs requires solving several technical challenges. This is primarily due to the fact that cache performance issues cannot be detected solely by monitoring the program execution (unlike most of the problems in functionality testing). To overcome this problem, we employ novel strategies to instrument the original program with a set of *assertions* at appropriate locations. Such an instrumentation is *entirely automatic*. The violation of any assertion captures a *unique* cache thrashing scenario in the original program (and not in the program instrumented with assertions). Thus, such assertion violations can be reported to the developer for investigation. We first carry out static cache analysis on the program to decide the set of program points that *may* exhibit cache thrashing. Subsequently, we systematically generate assertions at such places to expose cache thrashing in the program itself. *In a broader view, therefore, we reduce the problem of testing cache performance to an equivalent functionality testing problem*. The required functionality of the software is augmented with the set of assertions introduced by us.

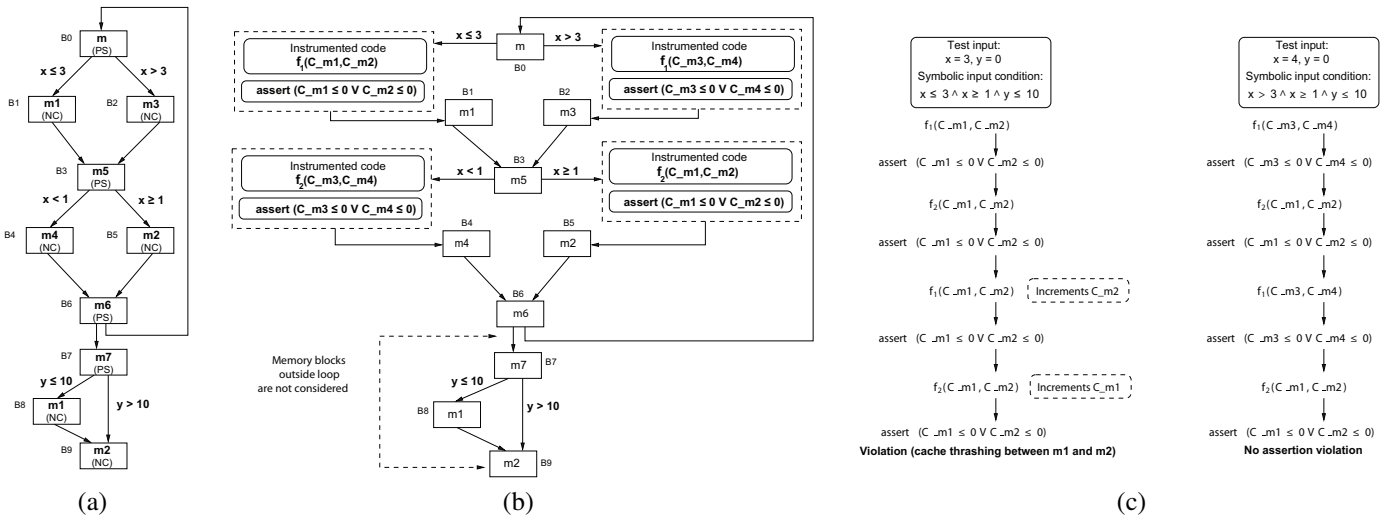


Fig. 2: Overview of test generation (a) program control flow graph with accessed memory blocks shown inside each basic block. The *cache hit-miss classifications* (CHMC) are also shown along with each memory block. The branch conditions are shown beside the respective control flow edges, (b) instrumented program with assertions to expose *cache thrashing behaviour*, (c) violation of assertion in an execution trace captures the *cache thrashing scenario* involving memory blocks $m1$ and $m2$

To check the validity of different assertions, we build a *dynamic path exploration* strategy that directs the path searching process towards the set of instrumented assertions. Each time an assertion is encountered during execution, its *validity* is checked *on-the-fly*. If an assertion is not satisfied during program execution, a cache-performance issue is recorded along with the respective input state (*i.e.* the set of inputs that leads to the violation of the assertion, cf. Figure 1). Primary objective of the path exploration strategy is to check maximum number of unique assertions, in a given amount of time. Therefore, to improve the search efficiency of path exploration, we direct the search process towards a control flow that has maximum number of unchecked assertions *control dependent* on it. Such a directed search is accomplished by consulting the control dependency graph of the instrumented program. Finally, since we dynamically explore the set of assertions, our computed test-suite does not contain any *false positives*. Precisely, any test case included in the computed test-suite captures a cache performance issue (specifically, a cache thrashing scenario) in some *feasible execution* of the software.

Contributions: In summary, we propose a test-generation framework that exposes the cache performance issues of an embedded software to the developer. Such a framework leverages the results obtained from an abstract interpretation based cache analysis to determine the cache behaviour and directs a dynamic test generation process to explore only the relevant portions of a program (*i.e.* program subparts that may exhibit cache performance issues). Due to the usage of dynamic analysis in test generation process, one appealing nature of our generated test suite is that it does not include any *spurious test cases* (*i.e.* a test-case that does not capture a cache performance issue in any *feasible* execution). We have implemented the entire framework using Chronos [11], an open source, freely available worst case execution time (WCET) analyzer. and LLVM [2], an open source, freely available compiler infrastructure. Specifically, the dynamic test generation process is implemented on top of LLVM and the static cache analysis is built on top of Chronos. Our experience with several open source subject programs suggests that we can quickly find several cache-performance stressing test cases. Last but not the least, we show the application of our test suite in *(i)* design space exploration and *(ii)* cache performance optimization.

II. Overview

In this section, we shall give an outline of our test generation process that stresses the cache performance of a program. We shall walk through a simple example as shown in Figure 2. For the sake of illustration, let us assume a direct-mapped cache where memory blocks $\{m_1, m_2, m_3, m_4, m, m_5, m_6, m_7\}$ in the control flow graph are mapped to different cache sets $\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5, \mathcal{S}_6\}$ as follows: $m1 \mapsto \mathcal{S}_1$, $m2 \mapsto \mathcal{S}_1$, $m3 \mapsto \mathcal{S}_2$, $m4 \mapsto \mathcal{S}_2$, $m \mapsto \mathcal{S}_3$, $m_5 \mapsto \mathcal{S}_4$, $m_6 \mapsto \mathcal{S}_5$ and $m_7 \mapsto \mathcal{S}_6$. Therefore, $m1$ and $m2$, as well as $m3$ and $m4$ conflict in the cache.

Figure 3 presents an overview of our test generation framework. Broadly, our approach contains two separate steps: (i) static cache analysis and (ii) dynamic test generation to expose different cache thrashing scenarios in the program. The static cache analysis directs the dynamic test generation process to explore only the relevant portions of a program. Such relevant portions capture designated program points that are more likely to expose cache thrashing behaviour.

Static cache analysis is performed via abstract interpretation [18]. Memory blocks are categorized as AH (*always cache hit*), AM (*always cache miss*) and PS (*persistent* or never evicted

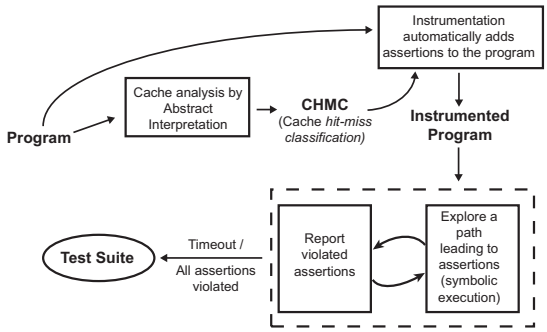


Fig. 3: Overview of our test generation framework

from the cache). If a memory block cannot be categorized as AH, AM or PS, it is categorized as NC (*not classified*). As an AH/PS categorized memory block can face only *cold cache misses*, we conclude that AH/PS categorized memory blocks can never be involved in a cache thrashing scenario. Therefore, only AM or NC categorized memory blocks exhibit potential sources of *cache thrashing*. If we employ abstract interpretation based cache analysis in the example program of Figure 2(a), we observe that memory blocks m, m_5, m_6 and m_7 are categorized as PS (note that m, m_5 and m_6 do not face any cache conflict within the loop). On the contrary, memory blocks $m1$, $m2$, $m3$ and $m4$ are categorized as NC (as $m1$ conflicts with $m2$ and $m3$ conflicts with $m4$ in the cache).

The key to our test generation approach is to create an interface between *static cache analysis* and *dynamic test generation*. Such an interface is developed via systematically generating *assertions*. The set of assertions has an *one-to-one correspondence* with the set of *cache thrashing scenarios*. The violation of any assertion exposes a unique cache thrashing scenario. Therefore, in a broader perspective, our performance testing framework can be viewed as a reduction of the cache performance problem to an equivalent functionality testing problem. Figure 2(b) demonstrates the schematic of the interface. The interface mainly consists of two parts: (i) instrumented code to count cache conflicts, and (ii) set of assertions to be checked. It is worthwhile to note that the instrumented program (*i.e.* Figure 2(b)) may have a different cache behaviour compared to the original program. This is due to the presence of additional instrumented code in Figure 2(b). However, the instrumented code (*i.e.* functions f_1 and f_2) as well as the assertions (cf. Figure 2(b)) take input from memory blocks in the original program (*i.e.* memory blocks $m1$,$m2$,$m3$ and $m4$ in Figure 2(a)). Therefore, violation of any assertion captures a cache thrashing scenario in the original program shown in Figure 2(a) (and not in the instrumented program shown in Figure 2(b)).

Let us first consider the set of memory blocks $\{m1, m2\}$ in Figure 2(b). C_m1 (C_m2) captures the amount of cache conflicts generated to memory block $m1$ ($m2$). Specifically, for *least recently used* (LRU) cache replacement policy, C_m1 (C_m2) captures the number of *unique cache conflicts (i.e. number of unique memory blocks mapping to the same cache set) between two consecutive accesses of memory block* $m1$ *(*$m2$*)*. Therefore, if $C\_m1 > 0$ (recall that we assumed a direct-mapped cache) before accessing memory block $m1$, accessing $m1$ will result in a *cache miss*. The instrumented code essentially manipulates the set of variables $\{C\_m1, C\_m2, C\_m3, C\_m4\}$ through some additional code fragments. At this point, without going into the details of instrumentation, we represent the instrumentation as *functions* to show the specific variables they manipulate. As shown in Figure 2(b), a function $f_1(C\_m1, C\_m2)$ only manipulates C_m1 and C_m2 (and neither C_m3 nor C_m4). In general, a cache miss does not necessarily capture a *cache thrashing scenario*. For the set of memory blocks $\{m1, m2\}$, we informally say that a cache thrashing happens when *both* $m1$ *and* $m2$ *are evicted from the cache at least once*. Therefore, the cache thrashing scenario involving memory blocks $m1$ and $m2$ is captured by the following formula:

$$\Phi_{12} \equiv C\_m1 > 0 \wedge C\_m2 > 0$$

The placed assertion checks the formula $\neg\Phi_{12} \equiv C\_m1 \leq 0 \vee C\_m2 \leq 0$ during dynamic test generation process. As a result, any violation of the assertion (formula $\neg\Phi_{12}$) captures a *cache thrashing scenario* in a *real execution*. The cache thrashing scenario involving memory blocks $\{m3, m4\}$ can be captured in a similar fashion using the formula $\Phi_{34} \equiv C\_m3 > 0 \wedge C\_m4 > 0$. Therefore, the violation of $\neg\Phi_{34}$ during the dynamic test generation process will capture a *real cache thrashing scenario* involving memory blocks $m3$ & $m4$.

Let us now investigate our dynamic test generation process. The primary goal of the dynamic test generation module is to stress the execution towards the set of instrumented assertions. The idea of our dynamic test generation has been inspired by recent advances in *satisfiability modulo theory* (SMT) and constraint-based test generation [12]. Our test generation module first executes the instrumented program with a random input, records the set of *violated assertions* (*i.e.* the set of real cache thrashing scenarios) and collects the constraints along the executed path. We assume x and y are inputs to the program. Figure 2(c) captures the execution trace for an input $x = 3, y = 0$. Due to the increment of both C_m1 and C_m2 (by the instrumented code $f_1(C\_m1, C\_m2)$ and $f_2(C\_m1, C\_m2)$, respectively), the assertion $assert(C\_m1 \leq 0 \vee C\_m2 \leq 0)$ is violated (as shown in Figure 2(c). Such an assertion violation captures the cache thrashing scenario involving memory blocks $m1$ and $m2$. To drive the execution towards other assertions, we first collect the constraints along the current execution trace. For an input $x = 3, y = 0$, such constraints can be expressed by the formula $x \leq 3 \wedge x \geq 1 \wedge y \leq 10$. To execute a different path, one of the branch conditions (*i.e.* $x \leq 3$, $x \geq 1$ or $y \leq 10$) must be negated [12]. Our test generation employs strategies to systematically negate the branches, so that the execution may lead to *maximum number of unchecked assertions*.

To check maximum number of assertions, we consult the control dependency graph (CDG) of a program. CDG captures the set of control conditions that are *necessary* to execute a certain statement. Figure 4 shows the CDG of Figure 2(b). The

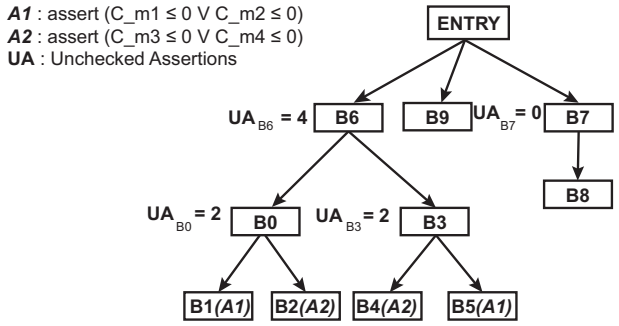


Fig. 4: Control Dependence Graph, for Figure 2(a)

two assertion from the 2(b) as shown as literals $A1$ and $A2$ in the CDG. The value against each control dependency nodes denotes the maximum number of unchecked assertion (UA) reachable from that node. In the example shown in Figure 2(b) three control conditions $x \leq 3$, $x \geq 1$ and $y \leq 10$ correspond to blocks $B0$, $B3$ and $B7$ respectively. As can be observed from Figure 4, negating the control condition at $B7$ (*i.e.* $y \leq 10$) will not lead to any unchecked assertions. Therefore, we must negate the control conditions at $B0$ (*i.e.* $x \leq 3$) or $B3$ (*i.e.* $x \geq 1$). In general, our method employs a *greedy strategy* to pick a control condition, which can lead to maximum number of unchecked assertions. Assume that branch $x \leq 3$ is chosen for negation and we obtain a test input $x = 4, y = 0$ for the symbolic condition $x > 3$. Executing the program for $x = 4, y = 0$ never violates any assertions. Note that the formula $x > 3 \wedge x < 1$ must be satisfied to execute both $f_1(C\_m3, C\_m4)$ and $f_2(C\_m3, C\_m4)$. However, the formula $x > 3 \wedge x < 1$ is clearly *unsatisfiable*. Therefore, $x > 3 \wedge x < 1$ captures an *infeasible path* in Figure 2(a) and $f_1(C\_m3, C\_m4)$ and $f_2(C\_m3, C\_m4)$ cannot appear in any execution trace together. As a result, the assertion $assert(C\_m3 \leq 0 \vee C\_m4 \leq 0)$ is always validated.

In the end, for the example shown in Figure 2, our framework finds *exactly one* cache thrashing scenario (that involves memory blocks $m1$ and $m2$) and a test input capturing the same thrashing scenario (*i.e.* $x = 3$ for a symbolic formula $x \leq 3 \wedge x \geq 1$). Our framework guarantees to cover *all the assertions* at the end of the test generation method. Note that such a process is in general *undecidable* [12] due to the inherent limitations imposed by constraint solvers. Therefore, the test generation process may go on forever. However, our test generation has the *anytime* property, meaning that the test generation process can be terminated anytime if the time budget is violated. After such a premature termination, the computed test-suite exposes a subset of thrashing scenarios that exist in the program. In fact, due to our directed search via CDG, our experimental results suggest that we can find most thrashing scenarios very early in the test generation process.

System and application model: In this work, we shall assume the traditional configuration of WCET analysis. Therefore, we consider only uninterrupted executions of a program and the computed thrashing scenarios appear solely due to the *intra-task* variant of cache conflicts. However, given a set of preemption points, our technique can be extended to capture thrashing scenarios that may appear only in the presence of preemptions. Such an extension will need to instrument the preempting tasks to compute the inter-task cache conflicts and it will require to shift the execution across tasks during test generation. Moreover, for the sake of simplicity, our framework is shown for separated instruction and data caches. To consider unified caches, the computation of cache conflicts can be combined during instrumentation (*i.e.* the computation of variables $\{C\_m1, \ldots, C\_m4\}$ in Figure 2).

III. Test generation methodologies

In this section, we shall describe our test generation methodologies in detail. Broadly, our test generation methodology contains two substeps: (i) systematically generating assertions to expose cache thrashing behaviour and (ii) a *dynamic* test generation to check the validity of the generated assertions. We shall elaborate these two steps in the following sections. For the sake of simplicity, we shall describe the core methodologies for instruction caches and we shall mention the minor changes required in the instrumentation to handle data caches.

A. Generating assertions

1) Code Instrumentation: Figure 5 shows the instrumented code for our example program in Figure 2. We assume that memory blocks $m1$ and $m2$ conflict in a direct-mapped cache. Therefore, after the static cache analysis, both $m1$ and $m2$ are categorized as *unclassified* (NC). Informally, the instrumented code manipulates the cache conflict faced by a particular memory block. Such an instrumentation depends on the underlying *cache replacement policy*. For the sake of illustration, we shall use *least recently used* (LRU) cache replacement policy. However, such an instrumentation can easily be changed for other cache replacement policies (*e.g.* FIFO) in a similar fashion as in our previous work [9].

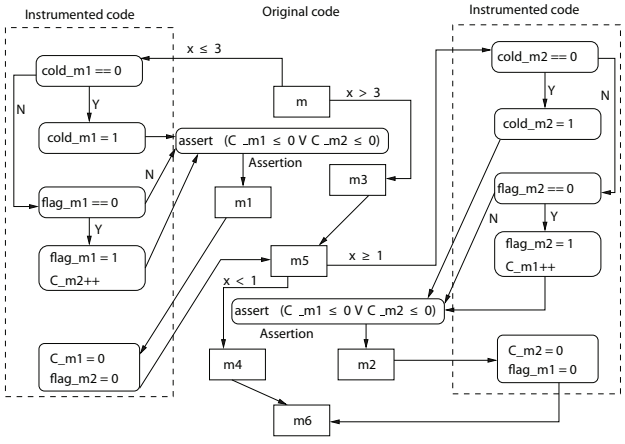


Fig. 5: Instrumented code with assertions

The heart of the instrumentation shown in Figure 5 lies in manipulating the two variables C_m1 and C_m2. For LRU cache replacement policy and a particular memory block

m, C_m *captures the number of unique cache conflicts between two consecutive accesses of memory block* m [1]. While counting such cache conflicts, we do not count the conflicts generated merely due to *cold misses*. Let us consider the instrumented code before memory block $m1$ (as shown in Figure 5). Since memory block $m1$ creates conflicts to only memory block $m2$, such cache conflicts are captured by the increment of variable C_m2. Variable $flag\_m1$ is used to count only *unique cache conflicts* (*i.e.* the number of unique memory blocks conflicting with memory block $m1$). Besides, variable $cold\_m1$ is used to discard the *cold cache miss* for accessing memory block $m1$. The instrumented code introduced for memory block $m2$ is entirely symmetric to the one introduced for block $m1$.

2) Formulation of assertions: The crucial step of the instrumentation is to systematically inserting assertions to expose *cache thrashing*. Cache thrashing behaviour only happens inside program loops. Therefore, for rest of the discussion, we shall only consider memory blocks inside program loops. Moreover, without loss of generality, we shall consider memory blocks mapped to a single cache set. For set-associative caches, the process is identically applied for each cache set.

The formulation of an assertion depends on the definition of *cache thrashing*. Therefore, we first formally define the notion of *cache thrashing* used in this paper.

Definition 3.1: Consider a $\mathcal{K}$-way set associative cache. A set of memory blocks $\mathcal{M} := \{M_1, M_2, \ldots, M_{\mathcal{K}+1}\}$ is said to have cache thrashing if and only if, for all $i \in [1, \mathcal{K}+1]$, access to M_i suffers at least one non-cold miss and all the cache conflicts for this non-cold miss are generated by the set of memory blocks $\mathcal{M} \setminus \{M_i\}$.

In the preceding definition of cache thrashing, the number of *non-cold* misses (say $\mathcal{X}$) is a tunable parameter. In our work, we assume $\mathcal{X} = 1$. However, in the following, we show that our technique can be generalized for different values of $\mathcal{X}$.

The instrumented code in Figure 5 takes the accessed memory blocks in the original program (*i.e.* the set of memory blocks $\{m1, m2, m3, m4\}$ in Figure 2(a)) as input. Therefore, it is worthwhile to note that the instrumented code manipulates cache conflicts in the original program (*i.e.* Figure 2(a)) and not the instrumented program shown in Figure 5. Since the validity of inserted assertions are based on this instrumented code, any violation of an assertion essentially captures a cache thrashing scenario in the original program (*i.e.* the program shown in Figure 2(a)).

To describe the generation of assertions, we shall begin with a few notations and definitions. Let us assume $\mathcal{M}_l = \{M_1, M_2, \ldots, M_N\}$ is the set of memory blocks accessed inside some program loop. We define a *thrashing set* as a subset of $\mathcal{M}_l$ that may be potentially involved in cache thrashing. Formally, a thrashing set $\mathcal{TS}_l$ is defined as follows

$$\mathcal{TS}_l = \{m \mid m \in \mathcal{M}_l \wedge CHMC(m) \neq PS \wedge CHMC(m) \neq AH\} \quad (1)$$

[1]For FIFO cache replacement policy, C_m captures the number of unique cache conflicts faced by m since it is last reloaded into the cache [9]

In Equation 1, $CHMC$ captures the *cache hit-miss classification* obtained via static cache analysis [18]. Note that AH (all-hit) and PS (persistent) categorized memory blocks can never be evicted from the cache (due to the inherent guarantee provided by static analysis). Therefore, we do not include such memory blocks as the potential cause of cache thrashing.

From a *thrashing set*, we define a number of *thrashing scenarios*. Informally, a cache thrashing scenario contains *just enough* memory blocks from a thrashing set to create a potential cache thrashing. If we assume that the associativity of the cache is $\mathcal{K}$, the *minimum* number of memory blocks to create a cache thrashing is $\mathcal{K}+1$. Therefore, a *thrashing scenario* for a thrashing set $\mathcal{TS}_l$ is defined as any $\mathcal{K}+1$ combination of the thrashing set $\mathcal{TS}_l$. The set of all cache thrashing scenarios Ω_l can be defined as follows.

$$\Omega_l = \{\ S \subseteq \mathcal{TS}_l \mid |S| = \mathcal{K}+1\} \quad (2)$$

Note that a thrashing set $\mathcal{TS}_l$ has a total of $\binom{|\mathcal{TS}_l|}{\mathcal{K}+1}$ different cache thrashing scenarios.

Finally, we generate exactly one assertion for each cache thrashing scenario. Let us assume one such cache thrashing scenario $\Theta \in \Omega_l$ and its respective assertion $\mathcal{A}_\Theta$. Informally, the assertion $\mathcal{A}_\Theta$ captures the property that thrashing scenario Θ *never happens in any program execution*. As a result, any violation of the assertion $\mathcal{A}_\Theta$ during dynamic test generation captures a realization of the thrashing scenario Θ. Formally, thrashing scenario Θ is captured by the following property.

$$\Phi_\Theta \equiv \bigwedge_{m \in \Theta} (C_m > \mathcal{K}) \quad (3)$$

In Equation 3, C_m captures the amount of unique cache conflicts faced by two consecutive accesses of memory block m. Since the assertion checks the negation of thrashing scenario, it can be formalized as follows.

$$\mathcal{A}_\Theta \equiv assert(\neg\Phi_\Theta) \quad (4)$$

The assertion $\mathcal{A}_\Theta$ is placed before each memory block involved in the thrashing set Θ. For example, in Fig. 5, the set $\{m1, m2\}$ captures a thrashing scenario and the assertion $assert(C\_m1 \leq 0 \vee C\_m2 \leq 0)$ was placed before accessing memory blocks $m1$ and $m2$.

The purpose of the preceding assertion (Eq. 4) is to validate that *at least one of the memory block from the thrashing set is never evicted from the cache*. Therefore, if all of the memory blocks in a thrashing set are evicted *at least once*, an assertion violation will be triggered and a cache performance issue will be reported. The assertion $\mathcal{A}_\Theta$ is checked dynamically before accessing each memory block involved in the thrashing scenario Θ.

It is worthwhile to mention that our formalization to capture cache thrashing (*i.e.* Definition 3.1 and Equation 3) is independent of cache replacement policy. Therefore, such a formalization can be applied to a wide variety of cache architectures. The effect of different cache architectures (*e.g.* caches with different replacement policies) will be reflected via the variable C_m (cf. Section III-A1). Finally, we can also

generalize the notion of reporting a cache thrashing scenario at runtime. Specifically, a cache thrashing scenario can be reported for $\mathcal{X}$ number of violations of an assertion (and thereby $\mathcal{X}$ number of evictions for each memory block in the respective thrashing set) instead of only one violation. Such a generalization also corresponds to the reconfiguration of the number of non-cold misses, as described in Definition 3.1.

3) Handling data caches: For data caches, the memory block classification is obtained using the *scope-aware persistent* (SCP) analysis [13]. SCP analysis can be used to classify data memory blocks as persistent or non-persistent. Unlike the instruction cache analysis, determining the set of data memory blocks accessed at a program point, can be challenging. Existing address analysis techniques such as the one used in [13] can be used to obtain the set of memory blocks, which may be accessed at a given program point. Once the SCP analysis has been performed on the set of memory blocks generated by address analysis, the assertions can be generated as described in preceding paragraphs.

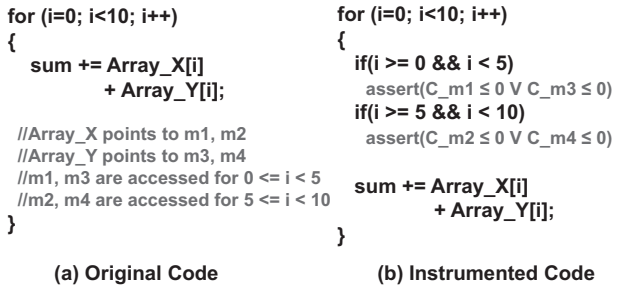


Fig. 6: Instrumentation scenarios for data caches

The basic structure for instrumentation is similar to what was described for instruction caches. However, unlike an instruction access, a data access may correspond to multiple memory blocks. For example, in Figure 6, access to Array_X might result in fetching of memory block $m1$ or $m2$, depending upon the loop iteration. Likewise, access to Array_Y might result in fetching of memory block $m3$ or $m4$.

Assume that the address analysis has reported that the memory blocks $m1$ and $m3$ are accessed for loop iterations i $\in$ [0..4] and memory blocks $m2$ and $m4$ are accessed for loop iterations i $\in$ [5..9]. Also assume that the only sets of memory blocks which conflict in the cache are $\{m1,m3\}$ and $\{m2,m4\}$. Under these assumptions, memory block $m1$ and $m3$ can participate in a thrashing scenario, only during loop iterations [0..4]. Therefore, the instrumented code for $m1$ and $m3$ needs to be proceeded by conditional checks on the iteration number ($i >= 0$ && $i < 5$). Conditional checks for memory block $m2$ and $m4$ can be placed in a similar fashion. Figure 6(b) shows the instrumented code for the example program shown in Figure 6(a).

B. Dynamic test generation

Dynamic test generation tries to find violations of the instrumented assertions (cf. Section III-A). Our dynamic test generation process is inspired by recent advances in *constraint solving* and *concolic* testing [12]. As an output of the dynamic test generation process, we obtain a pair $\langle \Theta_i, \Psi_i \rangle$ for each cache thrashing scenario Θ_i. In the output pair, Ψ_i captures a symbolic formula on the input variables, such that any input satisfying the formula leads to the cache thrashing scenario Θ_i. In our example (cf. Figure 2), one such output would be $\langle \{m1, m2\}, x \leq 3 \wedge x \geq 1 \rangle$. This implies that any input value of x $\in$ [1,3], would lead to a thrashing scenario, involving memory blocks m_1 and m_2.

Algorithm 1: The primary goal of Algorithm 1 is to check the validity of instrumented assertions (cf. Section III-A). It takes the instrumented program $\mathcal{P}_\mathcal{A}$ and the set of instrumented assertions $\mathcal{A}$ as inputs and generates a set of test cases $\mathcal{T}$. Each element in $\mathcal{T}$ realizes a unique cache thrashing scenario. To begin with, Algorithm 1 executes the instrumented program $\mathcal{P}_\mathcal{A}$ with a random input $\mathcal{I}$ and collects the execution trace $\mathbb{X}$. The exploration of different assertions is performed by systematically manipulating the *path condition* of this execution trace. Formally, *path condition* is defined as follows.

Definition 3.2: For a particular execution trace $\mathbb{X}$, path condition is a quantifier free first order logic formula that captures exactly the set of inputs which drives the program execution through the execution trace $\mathbb{X}$.

For example, in Figure 2(c), the symbolic formula $x \leq 3 \wedge x \geq 1 \wedge y \leq 10$ captures the *path condition* for the execution trace on input values $x = 3$ and $y = 0$.

The variable *unchecked* represents the set of unexplored partial path conditions in the instrumented program $\mathcal{P}_\mathcal{A}$. Each partial path condition φ_i is associated with a metric $\mathcal{F}_{\varphi_i}$. This metric measures the maximum number of non-violated assertions reachable from φ_i. We employ a *greedy strategy* based on the value of $\mathcal{F}_{\varphi_i}$ to continue exploration. More precisely, we generate a test input τ_θ from an unexplored, partial path condition φ_i that has the maximum value for $\mathcal{F}_{\varphi_i}$. Subsequently, we invoke the procedure *ExecuteAndReport* with input τ_θ.

Procedure ExecuteAndReport executes the instrumented program $\mathcal{P}_\mathcal{A}$ for an input τ to obtain the execution trace $\mathbb{X}$. For a particular execution, assume that $\varphi \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$ captures the path condition for the execution trace $\mathbb{X}$. Further assume that $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$ is the set of violated assertions and π is the shortest path-prefix in the $\mathbb{X}$ which captures *at least one violation* of each assertion in the set $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$. Therefore, an assertion, if violated beyond path-prefix π, must belong to the set $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$. If φ is the respective path condition, say $\hat{\varphi}$ be the prefix of the path condition corresponding to path-prefix π. Note that $\varphi \vDash \hat{\varphi}$, however, $\hat{\varphi}$ is a compact yet *lossless* formula to manifest the set of thrashing scenarios (*i.e.* the set of thrashing scenarios exposed by violations of the set of assertions $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$). Therefore, for each

Algorithm 1 Dynamic exploration of instrumented assertions

Input:
$\mathcal{P}_{\mathcal{A}}$: instrumented program with assertions
$\mathcal{A}$: set of instrumented assertions
Output:
$\mathcal{T}$: a set of test cases, each of which realizes a unique cache thrashing scenario
$AllPathConditions = unchecked = \mathcal{T} =$ empty
/*build the control dependency graph (CDG) of $\mathcal{P}_{\mathcal{A}}$ */
$CDG_{\mathcal{P}_{\mathcal{A}}} \leftarrow$ `BuildCDG`($\mathcal{P}_{\mathcal{A}}$)
select a random input $\mathcal{I}$
`ExecuteAndReport`($\mathcal{P}_{\mathcal{A}}$,$\mathcal{A}$, $\mathcal{I}$,$CDG_{\mathcal{P}_{\mathcal{A}}}$)
while $unchecked \neq$ empty $\wedge$ $\mathcal{A} \neq$ NULL **do**
    /*pick a partial path with maximum number of reachable and non-violated assertions */
    select $\langle \varphi, \mathcal{F}_{\varphi} \rangle \in unchecked$ with maximum $\mathcal{F}_{\varphi}$
    $unchecked := unchecked \setminus \{\langle \varphi, \mathcal{F}_{\varphi} \rangle\}$
    let $\varphi \leftarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{r-1} \wedge \psi_r$
    /* execute an unexplored path */
    if φ is satisfiable **then**
        $\tau_{\varphi} \leftarrow$ some concrete inputs satisfying φ
        `ExecuteAndReport`($\mathcal{P}_{\mathcal{A}}$, $\mathcal{A}$, τ_{φ},$CDG_{\mathcal{P}_{\mathcal{A}}}$)
    end if
end while
procedure EXECUTEANDREPORT($\mathcal{P}_{\mathcal{A}}$,$\mathcal{A}$,$\tau$,$CDG_{\mathcal{P}_{\mathcal{A}}}$)
    execute $\mathcal{P}_{\mathcal{A}}$ on input τ
    let $\mathbb{X}$ be the execution trace on input τ
    let $\varphi \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k$ be the path condition
    let $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$ be the set of *violated* assertions, on input τ
    $\mathcal{A} = \mathcal{A} \setminus \{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$
    let π be the *shortest* path-prefix along the execution trace $\mathbb{X}$ that captures at least one violation of each assertion in the set $\{\mathcal{A}_{\theta_1}, \mathcal{A}_{\theta_2}, \ldots, \mathcal{A}_{\theta_r}\}$
    let $\hat{\varphi}$ captures the partial path condition corresponding to the path-prefix π
    /* augment the test suite with witnesses for cache thrashing scenarios */
    $\mathcal{T} \bigcup = \{\langle \theta_1, \hat{\varphi} \rangle, \langle \theta_2, \hat{\varphi} \rangle, \ldots, \langle \theta_r, \hat{\varphi} \rangle\}$
    /*build all partial path conditions */
    for $i \leftarrow 1, k$ **do**
        let $\varphi_i \leftarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$
        if $\varphi_i \notin AllPathConditions$ **then**
            $AllPathConditions \bigcup = \varphi_i$
            let b_{end} be the control dependency edge in $CDG_{\mathcal{P}_{\mathcal{A}}}$ w.r.t. the branch condition $\neg\psi_i$
            /* compute the number of non-violated assertions ($\in \mathcal{A}$) reachable from b_{end} */
            $\mathcal{F}_{\varphi_i}$:= `GuidanceFunction`(b_{end})
            if $\mathcal{F}_{\varphi_i} \neq 0$ **then**
                $unchecked \bigcup = \{\langle \varphi_i, \mathcal{F}_{\varphi_i} \rangle\}$
            end if
        end if
    end for
end procedure

thrashing scenario θ_i, we construct a test pair $\langle \hat{\varphi}, \theta_i \rangle$ and add such test pairs to the existing test suite $\mathcal{T}$. To avoid any redundant computation, we manipulate $\hat{\varphi}$ *on-the-fly* during the execution.

To continue exploration, we must deviate from the present path. Such a deviation is performed by negating a branch conditions along the execution trace $\mathbb{X}$. Assume that we pick a partial path condition $\varphi_i \equiv \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$. Let us also assume that b_{end} is the control dependency edge in the CDG (of the instrumented program) that captures the negated branch condition $\neg\psi_i$. We rank each partial path condition φ_i with a metric $\mathcal{F}_{\varphi_i}$ and add it to the set of $unchecked$ partial path conditions. $\mathcal{F}_{\varphi_i}$ captures the *maximum* number of assertions in $\mathcal{A}$ that are reachable, if the program is executed with a test input satisfying φ_i.

GuidanceFunction captures the computation of $\mathcal{F}_{\varphi_i}$. Formally, $\mathcal{F}_{\varphi_i}$ is defined as follows.

$$\mathcal{F}_{\varphi_i} = |\{\mathcal{A}_{\theta} \in \mathcal{A} \mid b_{end} \rightsquigarrow \mathcal{A}_{\theta}\}| \tag{5}$$

Where $b_{end} \rightsquigarrow \mathcal{A}_{\theta}$ captures that the assertion $\mathcal{A}_{\theta}$ is reachable from the control dependency edge b_{end} (*i.e.* the control dependency edge corresponding to the negated branch condition ψ_i). Therefore, $\mathcal{F}_{\varphi_i}$ accounts for all the assertions in $\mathcal{A}$ that are reachable from b_{end}. It is worthwhile to note that the definition of $\mathcal{F}_{\varphi_i}$ (in Equation 5) can be changed very easily depending on the *criticality* of different assertions. In Equation 5, we have only considered the reachability of assertions, giving each assertion equal priority. However, the definition $\mathcal{F}_{\varphi_i}$ can be easily changed to incorporate other priorities (*e.g.* assertions in an innermost loop can be given higher priorities than assertions in an outermost loop).

Termination: Algorithm 1 terminates as soon as we obtain a witness (*i.e.* test case) for each cache thrashing scenario (captured by the condition $\mathcal{A} \neq$ NULL in the outermost loop of Algorithm 1). However, some thrashing scenarios might not be manifested due to the presence of *infeasible paths* in a program (*e.g.* the assertion $assert(C\_m3 \leq 0 \vee C\_m4 \leq 0)$ in Figure 2(c)). In such cases, the test generation process may go on forever in the presence of *unbounded* (*e.g.* input-dependent) loop iterations and due to the inherent *incompleteness* of any constraint solver. However, one appealing nature of our test generation process is that it can be terminated *anytime*. The resulting test-suite might be *incomplete*, but it can still be used for investigating cache performance issues in the program. Our experiments suggest that we can find most of the cache performance stressing test inputs in very early phase of Algorithm 1. This is primarily due to the directed search strategy along the control dependency chain (via the CDG) to reach the set of instrumented assertions.

C. Salient features of generated test suites

Our generated suite has several important properties. In the following description, we shall formally capture the properties of the generated test suite.

Property 3.1: At any point in time during the execution of Algorithm 1, for any cache thrashing scenario Θ_i, if no path

witnessing Θ_i has been explored - no test case containing Θ_i appears in the test-suite $\mathcal{T}$ computed so far. Otherwise, the entry $\langle \Theta_i, \Psi_i \rangle$ appears in the test-suite $\mathcal{T}$ where Ψ_i captures the set of all inputs which witness Θ_i, and whose paths have been explored already by Algorithm 1.

Property 3.2: If Algorithm 1 terminates, we can guarantee to find all *feasible cache thrashing scenarios in any uninterrupted execution, that is, all cache thrashing scenarios witnessed by at least one program input. Each such feasible cache thrashing scenario will appear as an entry $\langle \Theta_i, \Psi_i \rangle$ in the generated test suite $\mathcal{T}$ reported at the end of Algorithm 1. Any solution for the formula Ψ_i is a test input witnessing cache thrashing scenario Θ_i.*

IV. Evaluation

A. Experimental Set-up

Figure 7 shows an outline of our implementation framework. To generate cache hit-miss classifications (CHMC), we use the abstract interpretation (AI) based cache analyses (using [18] for instruction caches and using [13] for data caches) implemented in Chronos [11]. Outcomes of AI-based cache analyses are used by the instrumentation engine to compute thrashing sets and to insert assertions at appropriate program points (as explained in Section III-A). This instrumented program is passed to the dynamic test generation process.

Dynamic test generation process is implemented on top of LLVM compiler infrastructure [2]. The instrumented program is compiled into LLVM bitcode format and its control flow graph (CFG) is extracted from the LLVM bitcode. We also implement a module inside LLVM to compute the control dependency graph (CDG) of a given program. This CDG is used to guide our test generation, as explained in Algorithm 1. To generate path conditions for different execution traces, we use KLEE symbolic execution engine [1]. To solve and manipulate path conditions along an execution trace, we use the STP constraint solver [3].

We have performed all the experiments on a machine having an Intel Core-i5 processor, with 4 GB RAM and running Ubuntu 9.04 OS.

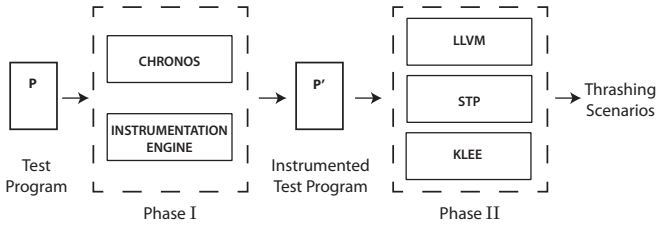


Fig. 7: Key phases in the framework

Subject Programs: Table I shows the subject programs, used in our experiments. *Nsichneu*[4] is an automatically generated code, which simulates an extended Petri Net. It was taken from the Mälardalen WCET benchmarks suite. It has a code size much larger than other programs used in our experiments. Also it contains a large amount of if-statements. *Papabench*[16] is an Unmanned Aerial Vehicle (UAV) control application. In our experiments, we used the auto-navigation utility from *papabench*. The auto-navigation utility contains a lot of input dependent paths, therefore it can potentially show different thrashing scenarios for different symbolic input formulas. *Jetbench*[17] is a real-time, Jet engine performance calculator. It uses Jet engine parameters and thermodynamic equations from the NASA's EngineSim program to perform real-time thermodynamic calculations. We use a single-threaded version of *Jetbench* for our experiments.

S. No	Test Program	Lines of Code
1	Nsichneu	4253
2	Papabench	1097
3	Jetbench	770

TABLE I: Subject Programs

B. Experimental Results

In following paragraphs, we shall describe some of the experiments which were performed to measure the efficacy of our framework. Also we shall discuss the applications of our framework for answering some of the issues related to design space exploration and performance optimization.

Efficacy of our framework, in exposing cache performance issues: We performed experiments with the three real-time programs listed in Table I. The results of which are discussed subsequently. But first we shall describe a few metrics which are used to present the experimental results.

Assertion Coverage : Our framework aims to find all thrashing scenarios due to *intra-task cache conflicts*. However, our test generation framework may not terminate (in general, this problem is undecidable [12]). Therefore, we define a metric named *Assertion coverage* which measures the percentage of unique assertion checked, within a given amount of time.

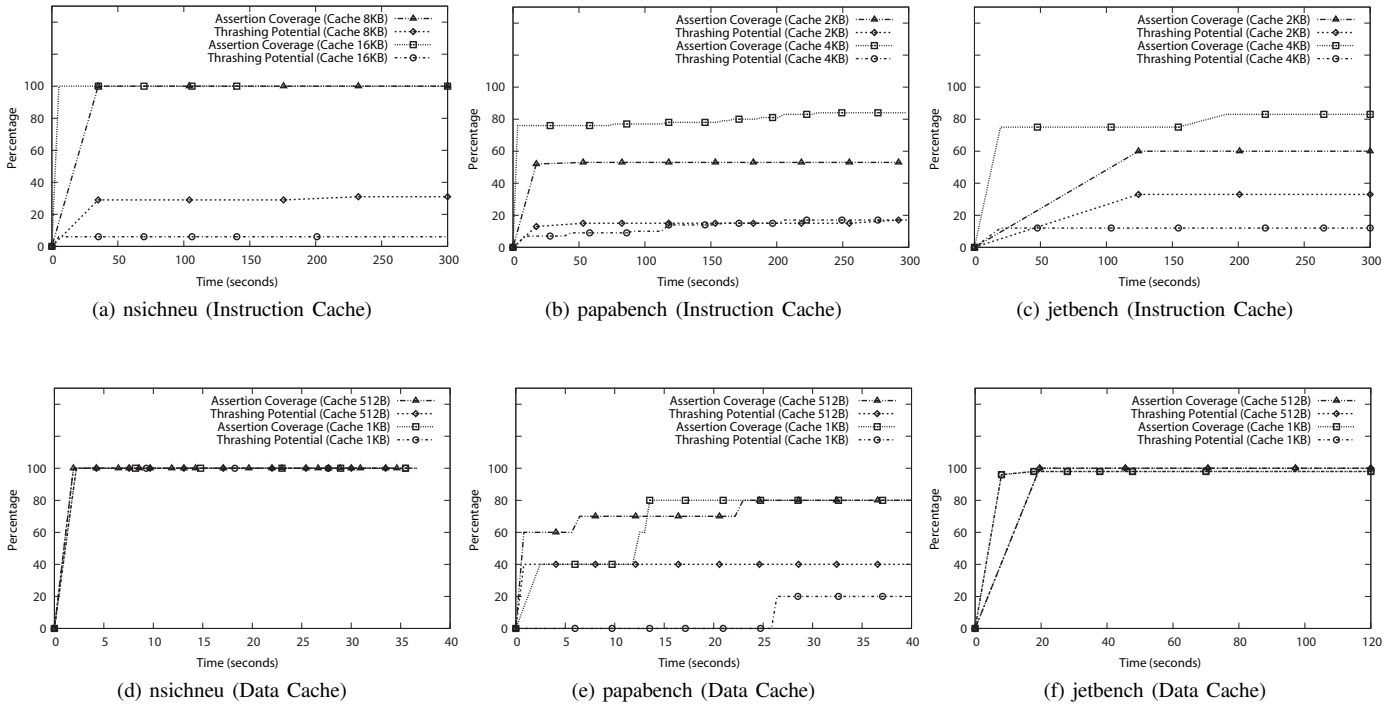
$$\text{Assertion Coverage} = \frac{\text{unique assertion checked}}{\text{unique assertion instrumented}} \times 100$$

A 100% assertion coverage implies that all unique assertions have been checked at least once.

Thrashing Potential : It is not necessary that all the checked assertion will be violated. However, the number of unique assertions violated, tells us about the potential for cache thrashing, for a program, on a given cache-configuration. Therefore, we define *Thrashing Potential* as follows

$$\text{Thrashing Potential} = \frac{\text{unique assertion violated}}{\text{unique assertion instrumented}} \times 100$$

Through our experiments we investigated the assertion coverage and the thrashing potential for all the program listed in Table I, for various cache-configurations. Some of the results from our experiments are shown in Figure 8. The plots in Figure 8 show the assertion coverage (and thrashing potential) on the y-axis and the exploration time on the x-axis. Since our framework looks for all possible thrashing scenarios due to intra-task cache conflicts, it is possible that the test generation will not terminate (refer to section III-B). Therefore, we tested



(a) nsichneu (Instruction Cache) (b) papabench (Instruction Cache) (c) jetbench (Instruction Cache)

(d) nsichneu (Data Cache) (e) papabench (Data Cache) (f) jetbench (Data Cache)

Fig. 8: Assertion Coverage and Thrashing Potential for different cache configurations

the subject programs, with an exploration budget of 5 minutes. We performed the experiments for instruction caches as well as data caches. Overall, we observed an assertion coverage ranging from 53% to 100% for different experimental set-ups (within a exploration budget of 5 minutes). Essentially, programs which had lesser number of input dependent paths (such as *nsichneu*) were explored much faster than program which had more number of input dependent paths (such as *papabench*). We also observed that for most of the experiments with instruction caches, only a small fraction of instrumented assertion were actually violated.

The figures presented in the first row (Fig. 8 (a), (b) and (c)) show the results, for instruction caches. On one hand, our chosen cache sizes are sufficient to avoid *capacity misses*. On the other hand, cache sizes are also small enough to generate *conflict misses*. Since *nsichneu* has a large code within a loop, we choose a relatively bigger cache for *nsichneu*, compared to the other two subject programs. Figure 8(a) shows the percentage coverage for *nsichneu*, on a 2-way, set-associative, LRU, instruction cache. The results reported here are for cache configuration of 8 KB and 16 KB. For both the configuration, the framework achieved a 100% assertion coverage, in less than 5 minutes. The thrashing potential for *nsichneu*, was observed to be less than 31% for both the experiments. Note that since the framework achieved a 100% assertion coverage for *nsichneu*, therefore the recorded thrashing potential is accurate. We performed experiments with *papabench* and *Jetbench* on a 2 KB and 4 KB for 2-way, set-associative, LRU cache. Neither of these experiments, resulted in a 100% assertion coverage, within the exploration budget of 5 minutes. However, this doesn't imply that the greedy exploration strategy is inefficient. This observation is supported by the fact that most of the explored assertions in Figure 8 were discovered early in the exploration. Additionally, some of the instrumented assertions may be present along infeasible paths (such as $x = 0 \wedge x = 1$), therefore they might not be checked throughout the exploration.

Figure 8 (d), (e) and (f) show the analysis results for data caches. For all the experimental results reported in this paragraph we used a direct-mapped, data caches of size of 1KB and 512B. We used small caches for this set of experiments, so as to create sufficient number of thrashing scenarios. For *nsichneu* and *Jetbench* we observed an assertion coverage of almost 100%. In fact, for *nsichneu* only one cache thrashing scenario was reported for both the cache configurations, which was covered (and violated) during exploration. Also, for *Jetbench* (see Figure 8(f)) most of the checked assertions were violated during exploration. However, for *papabench* (see Figure 8(e)), we observed an assertion coverage of 80% and a thrashing potential of less than 40%, for both the cache-configurations.

Applications of our framework for design space exploration: The process of embedded system design can be quite challenging due to sheer size of the design space that needs to be explored. While choosing a design for an embedded application, the designer has to consider various constraints such as timing and energy consumption. For instance, while choosing a cache-configuration, a designer can choose from a large, highly-associative cache or smaller, less-associative cache. A large, highly-associative cache might have lesser number of cache-thrashing scenarios however it will consume more power and possibly slower than the smaller cache. Therefore, determining the ideal cache size for a given application

might be tricky. Our framework can provide a suitable way to choose the appropriate cache configuration for an application.

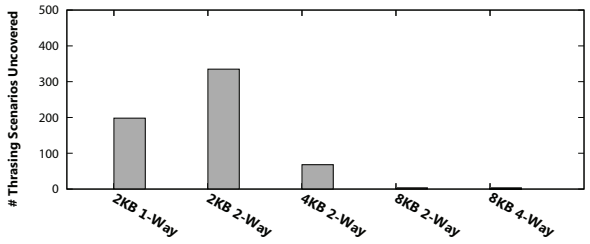


Fig. 9: Number of cache thrashing scenarios discovered for *papabench*, for various cache configurations

Essentially, our framework can be used to compare the number of thrashing scenarios for different cache configurations, for a given application. For example Figure 9 shows the number of thrashing scenarios discovered for different cache configurations, for *papabench*. It is worthwhile to note that our framework pinpoints the real thrashing scenarios, witnessed by a feasible execution. Existing techniques, which are purely based on static analysis (*e.g.* [18], [13]) may include *false thrashing scenarios* that never appear in any execution (cf. Figure 8). As a result, we can choose a more appropriate cache configuration using our framework, compared to the techniques based purely on static analysis.

In Figure 9, it might be interesting to know that a 2KB, 1-way (direct-mapped) cache has lesser number of cache thrashing scenarios then a 2KB, 2-way set-associative cache. Also, the experiments suggest that the number of cache thrashing scenarios for a 8KB, 2-way set-associative cache and a 8KB, 4-way set-associative cache are the same. So for this program, a 8KB, 2-way set-associative cache will be sufficient, to avoid cache-thrashing.

Applications of our framework for performance optimization: In this section, we shall discuss the application of our framework for input sensitive optimization, specifically for cache locking. The main intuition is explained via Figure 10(a). Assume a direct-mapped cache and memory blocks $m1$, $m2$, $m3$ and $m4$ all map to the same cache set. Clearly, this would result in a cache-thrashing scenario (for thrashing sets $\{m1, m2\}$ and $\{m3, m4\}$) and our test generation framework computes the following test cases: $\langle\{m1, m2\}, z \leq 5\rangle$ and $\langle\{m3, m4\}, z > 5\rangle$. In a way, therefore, our dynamic test generation framework can also be viewed as partitioning the input domain, where all inputs constituting a partition realizes the same set of cache thrashing scenarios. In our example, there are two such partitions - Δ_1 and Δ_2 (cf. Fig.10(b)).

Assume that we want to selectively lock memory blocks so that such memory blocks are never evicted from the cache. Traditional cache locking techniques, such as [15] can be used for such purposes. The work in [15] requires a memory trace (sequence of memory blocks) to determine the set of memory blocks that should be locked in the cache. However, a program might have different memory traces for different sets of inputs. If we use a memory trace generated for an input $I_1 \in \Delta_1$,

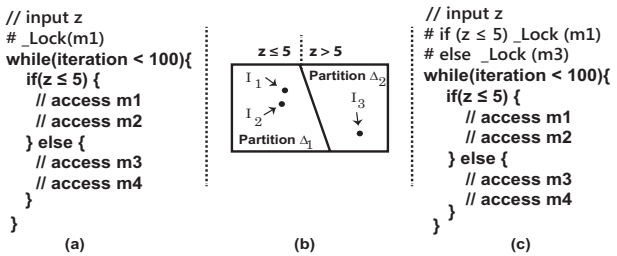


Fig. 10: Illustration of conditional cache locking (a) Program with unconditional cache locking (lock instructions are preceded by #) (b) Input partitions (c) Conditional cache locking

either $m1$ or $m2$ will be locked in the cache (as shown in Fig 10(a)). However, it can be observed that when the program is executed for any input $I_2 \in \Delta_2$, locking $m1$ or $m2$ (as shown in 10(a)) will not improve the cache performance. This is due to the fact that $m3$ and $m4$ will encounter *cache thrashing*.

Based on the discussion in the preceding paragraph, we argue the potential of performance optimization (*e.g.* cache locking) techniques that is sensitive to inputs. In particular, for cache locking optimization in Figure 10, we could lock $m1$ (or $m2$) for all inputs satisfying $z \leq 5$ and lock $m3$ (or $m4$) for all inputs satisfying $z > 5$. Such a conditional cache locking (as shown in Figure 10(c)), will improve the program performance for both the input partitions Δ_1 and Δ_2 (cf. Figure 10(b)).

To validate our argument, we have studied the feasibility of conditional cache locking technique on the subject program `nsichneu`. For baseline cache locking optimization, we use [15], that locks a set of memory blocks from a given memory trace. We conduct several experiments for two arbitrary inputs I_1 and I_2; where I_1 is used to generate a memory trace based on which we decide which memory blocks to lock in the cache, using the technique of [15], and I_2 is used to run `nsichneu` after the cache locking optimization is performed for the memory trace on input I_1. We have made the following crucial observations.

- If I_1 and I_2 belong to the same input partition produced by our framework, the performance improvement from cache locking observed in nsichneu is significantly greater than the situation where I_1 and I_2 belong to different input partitions. These results seem to motivate the use of conditional locking instructions.
- For the situation where I_1 and I_2 belong to the same partition, we also observed the performance improvement from locking varies across input partitions. On average, we observed a variation from $\sim$ 10% to 20% in performance improvement across different input partitions in *nsichneu*. Note that inputs from different partitions have different memory traces and so, they lead to different set of locked memory blocks using [15].

The preceding observations motivate the need for conditional cache locking, which can be studied at length in the future. Specifically, our observations conclude that memory

blocks should be locked differently across different input partitions computed by our framework.

V. Related work

Over the past two decades, a significant research effort has been put forward for the performance validation of embedded software. Such efforts include abstract interpretation (AI) based method, such as [18], which was proposed to analyze the cache behaviour of a program. Our previous work [9] improves the precision of such AI-based cache analyses via a gradual and controlled use of model checking. These works [18], [9] analyze the cache behaviour of a program irrespective of its inputs. On the contrary, our primary goal is to build a connection between the set of inputs and anomalous cache behaviours (*e.g.* cache thrashing). Our test generation methodology is inspired by the recent advances in constraint solving and *concolic* testing [12], [7]. These works aim to detect software functionality bugs. In contrast, we aim to detect software performance problems due to memory subsystems.

Different techniques used for *program profiling* [5], [14] also aim to find performance problems in a program. Such profiling techniques work on full or compressed execution traces to derive useful information about program performance. It is assumed that the relevant inputs for obtaining an execution trace are known *a priori*. Our approach is complementary to these profiling techniques, as our aim is to systematically find test inputs that lead to poor cache performance. Once such test inputs are found, they can be fed back to a traditional profiler for further analysis.

Recent advances in profiling [19], [10] have extended the traditional profiling technique to compute a *performance trend* of a program. Such a performance trend is captured by an approximate cost function. The cost function relates program inputs with the overall cost of the program. However, such cost functions are approximations and they do not necessarily capture the actual cost. Besides, these works do not introduce any notion of test coverage. On the contrary, any cache thrashing scenario reported by our framework is indeed a cache thrashing scenario, witnessed by a concrete input. Besides, our framework also reports the coverage of cache thrashing scenarios via the set of dynamically checked assertions.

The work proposed in [6] automatically finds test inputs for the worst-case computational complexity. Our work differs from [6] on several aspects: first, our notion of performance is based on the execution time rather than computational complexity. Secondly, the primary goal of our work is to compute test inputs for possible anomalous cache behaviour in a single program.

A recent work [8] uses constraint-based test generation [12], [7] to partition the input domain of a program with respect to cache performance. Once all the partitions are computed, some manual interventions are required to locate the set of program locations that may exhibit issues related to cache performance. Besides, the work proposed in [8] computes a cache performance range for each partition. The cache performance range in [8] is computed via *static invariant generation* methods. As a result, the computed cache-performance range might be over-approximated, leading to *false positives*. Our approach, on the contrary, directly relates a cache thrashing scenario with the set of inputs (without any manual intervention). Moreover, since we generate test inputs based on dynamic analysis, our generated test-suite does not contain any *false positives*.

VI. Conclusion

In this paper, we have proposed a test generation framework that stresses the cache performance of a program. The key novelty in our work is a systematic combination of static cache analysis and dynamic test generation via a set of instrumented assertions. Violation of any such assertion exposes a unique cache performance issue, specifically, a cache thrashing scenario in the program. As an output, our framework reports a test-suite where each test case in the test-suite points to a unique cache thrashing scenario along with a set of program inputs that leads to the same. Due to the use of dynamic test generation, our generated test-suite does not contain any spurious test cases. We have shown the application of our test generation framework in design space exploration and in cache performance optimization via cache locking.

Acknowledgement

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] STP constraint solver. https://sites.google.com/site/stpfastprover/.
[4] WCET benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
[5] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, 1996.
[6] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, 2009.
[7] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
[8] S. Chattopadhyay, L. K. Chong, and A. Roychoudhury. Program performance spectrum. In *LCTES*, 2013.
[9] S. Chattopadhyay and A. Roychoudhury. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real-Time Systems*, 2013.
[10] E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *PLDI*, 2012.
[11] X. Li et al. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007. http://www.comp.nus.edu.sg/~rpembed/chronos.
[12] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.
[13] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *RTAS*, 2011.
[14] J. R. Larus. Whole program paths. In *PLDI*, 1999.
[15] Y. Liang and T. Mitra. Instruction cache locking using temporal reuse profile. DAC, 2010.
[16] F. Nemer, H. Cassé, and P. Sainrat. Papabench: a free real-time benchmark. *WCET Workshop*, 2006.
[17] M. Qadri, D. Matichard, and K. M. Maier. JetBench: an open source real-time multiprocessor benchmark. *ARCS*, 2010.
[18] 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.
[19] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI*, 2012.