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.

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.

![Test generation framework](image-url)
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_5, m_6, m_7\} in the control flow graph are mapped to different cache sets \{S_1, S_2, S_3, S_4, S_5, S_6\} as follows: \(m_1 \mapsto S_1, m_2 \mapsto S_1, m_3 \mapsto S_2, m_4 \mapsto S_2, m_5 \mapsto S_3, m_6 \mapsto S_4\) and \(m_7 \mapsto S_6\). Therefore, \(m_1\) and \(m_2\), as well as \(m_3\) and \(m_4\) 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.

![Figure 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.](image-url)
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 $m_1$, $m_2$, $m_3$ and $m_4$ are categorized as NC (as $m_1$ conflicts with $m_2$ and $m_3$ conflicts with $m_4$ 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 $m_1$, $m_2$, $m_3$ and $m_4$ 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 $\{m_1, m_2\}$ in Figure 2(b). $C_{m_1}$ ($C_{m_2}$) captures the amount of cache conflicts generated to memory block $m_1$ ($m_2$). Specifically, for least recently used (LRU) cache replacement policy, $C_{m_1}$ ($C_{m_2}$) 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 $m_1$ ($m_2$). Therefore, if $C_{m_1} > 0$ (recall that we assumed a direct-mapped cache) before accessing memory block $m_1$, accessing $m_1$ will result in a cache miss. The instrumented code essentially manipulates the set of variables $\{C_{m_1}, C_{m_2}, C_{m_3}, C_{m_4}\}$ 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 $f_1(C_{m_1}, C_{m_2})$ only manipulates $C_{m_1}$ and $C_{m_2}$ (and neither $C_{m_3}$ nor $C_{m_4}$). In general, a cache miss does not necessarily capture a cache thrashing scenario. For the set of memory blocks $\{m_1, m_2\}$, we informally say that a cache thrashing happens when both $m_1$ and $m_2$ are evicted from the cache at least once. Therefore, the cache thrashing scenario involving memory blocks $m_1$ and $m_2$ is captured by the following formula:

$$\Phi_{12} \equiv C_{m_1} > 0 \land C_{m_2} > 0$$

The placed assertion checks the formula $\neg \Phi_{12} \equiv C_{m_1} \leq 0 \lor C_{m_2} \leq 0$ 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 $\{m_3, m_4\}$ can be captured in a similar fashion using the formula $\Phi_{34} \equiv C_{m_3} > 0 \land C_{m_4} > 0$. Therefore, the violation of $\neg \Phi_{34}$ during the dynamic test generation process will capture a real cache thrashing scenario involving memory blocks $m_3$ and $m_4$.

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 $C_{m_1}$ and $C_{m_2}$ (by the instrumented code $f_1(C_{m_1}, C_{m_2})$ and $f_2(C_{m_1}, C_{m_2})$, respectively), the assertion $assert(C_{m_1} \leq 0 \lor C_{m_2} \leq 0)$ is violated (as shown in Figure 2(c)). Such an assertion violation captures the cache thrashing scenario involving memory blocks $m_1$ and $m_2$.

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 $x \leq 3 \land x \geq 1 \land y \leq 10$. 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
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 $x > 3 \land x < 1$ must be satisfied to execute both $f1(C_{m3}, C_{m4})$ and $f2(C_{m3}, C_{m4})$. However, the formula $x > 3 \land x < 1$ is clearly unsatisfiable. Therefore, $x > 3 \land x < 1$ captures an infeasible path in Figure 2(a) and $f1(C_{m3}, C_{m4})$ and $f2(C_{m3}, C_{m4})$ cannot appear in any execution trace together. As a result, the assertion $assert(C_{m3} \leq 0 \lor C_{m4} \leq 0)$ 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 $x \leq 3 \land x \geq 1$). 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 preemptions 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](image)
Besides, variable \(m\) captures the number of unique cache conflicts between two consecutive accesses of memory block \(m\). 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 behavior 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 \(K\)-way set associative cache. A set of memory blocks \(\mathcal{M} := \{M_1, M_2, \ldots, M_{K+1}\}\) is said to have cache thrashing if and only if, for all \(i \in [1, K+1]\), 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 \(X\)) is a tunable parameter. In our work, we assume \(X = 1\). However, in the following, we show that our technique can be generalized for different values of \(X\).

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 \(\mathcal{T}_S\) is defined as follows

\[
\mathcal{T}_S = \{m \mid m \in \mathcal{M}_l \land \text{CHMC}(m) \neq \text{PS} \land \text{CHMC}(m) \neq \text{AH}\}
\]

In Equation 1, \(\text{CHMC}\) captures the cache hit-miss classification obtained via static cache analysis [18]. Note that \(\text{AH}\) (all-hit) and \(\text{PS}\) (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 \(K\), the minimum number of memory blocks to create a cache thrashing is \(K + 1\). Therefore, a thrashing scenario for a thrashing set \(\mathcal{T}_S\) is defined as any \(K + 1\) combination of the thrashing set \(\mathcal{T}_S\). The set of all cache thrashing scenarios \(\Omega_l\) can be defined as follows.

\[
\Omega_l = \{ S \subseteq \mathcal{T}_S \mid |S| = K + 1 \}
\]

Note that a thrashing set \(\mathcal{T}_S\) has a total of \(\binom{|\mathcal{T}_S|}{K+1}\) 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 \(A_\Theta\). Informally, the assertion \(A_\Theta\) captures the property that thrashing scenario \(\Theta\) never happens in any program execution. As a result, any violation of the assertion \(A_\Theta\) during dynamic test generation captures a realization of the thrashing scenario \(\Theta\). Formally, thrashing scenario \(\Theta\) is captured by the following property.

\[
\Phi_\Theta = \bigwedge_{m \in \Theta} (C_m > K)
\]

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.

\[
A_\Theta = \text{assert}(\neg \Phi_\Theta)
\]

The assertion \(A_\Theta\) is placed before each memory block involved in the thrashing set \(\Theta\). For example, in Fig. 5, the set \(\{m1, m2\}\) captures a thrashing scenario and the assertion \(\text{assert}(C_{m1} \leq 0 \lor C_{m2} \leq 0)\) 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 \(A_\Theta\) is checked dynamically before accessing each memory block involved in the thrashing scenario \(\Theta\).

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 $X$ number of violations of an assertion (and thereby $X$ 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 SCP analysis has been performed on 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 as described in preceding paragraphs.

for (i=0; i<10; i++)
{
    sum += Array_X[i] + Array_Y[i];
    //Array_X points to m1, m2
    //Array_Y points to m3, m4
    //m1, m3 are accessed for 0 <= i < 5
    //m2, m4 are accessed for 5 <= i < 10
}

(a) Original Code (b) Instrumented Code

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 $m_1$ or $m_2$, depending upon the loop iteration. Likewise, access to Array_Y might result in fetching of memory block $m_3$ or $m_4$.

Assume that the address analysis has reported that the memory blocks $m_1$ and $m_3$ are accessed for loop iterations $i \in [0, 4]$ and memory blocks $m_2$ and $m_4$ are accessed for loop iterations $i \in [5, 9]$. Also assume that the only sets of memory blocks which conflict in the cache are $\{m_1, m_3\}$ and $\{m_2, m_4\}$. Under these assumptions, memory block $m_1$ and $m_3$ can participate in a thrashing scenario, only during loop iterations $[0, 4]$. Therefore, the instrumented code for $m_1$ and $m_3$ needs to be proceeded by conditional checks on the iteration number ($i \geq 0$ & $i < 5$). Conditional checks for memory block $m_2$ and $m_4$ 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 $(\Theta_i, \Psi_i)$ for each cache thrashing scenario $\Theta_i$. In the output pair, $\Psi_i$ captures a symbolic formula on the input variables, such that any input satisfying the formula leads to the cache thrashing scenario $\Theta_i$. In our example (cf. Figure 2), one such output would be $\langle \{m_1, m_2\}, x \geq 3 \land x \geq 1 \rangle$. 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 $\mathcal{P}_A$ and the set of instrumented assertions $A$ as inputs and generates a set of test cases $T$. Each element in $T$ realizes a unique cache thrashing scenario. To begin with, Algorithm 1 executes the instrumented program $\mathcal{P}_A$ with a random input $I$ and collects the execution trace $X$.

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 $X$, 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 $X$.

For example, in Figure 2(c), the symbolic formula $x \leq 3 \land x \geq 1 \land y \leq 10$ 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 $\mathcal{P}_A$. Each partial path condition $\varphi_1$ is associated with a metric $F_{\varphi_1}$. This metric measures the maximum number of non-violated assertions reachable from $\varphi_1$. We employ a greedy strategy based on the value of $F_{\varphi_1}$ to continue exploration. More precisely, we generate a test input $\tau_0$ from an unexplored, partial path condition $\varphi_1$ that has the maximum value for $F_{\varphi_1}$. Subsequently, we invoke the procedure $\text{ExecuteAndReport}$ with input $\tau_0$.

Procedure $\text{ExecuteAndReport}$ executes the instrumented program $\mathcal{P}_A$ for an input $\tau$ to obtain the execution trace $X$. For a particular execution, assume that $\varphi \equiv \psi_1 \land \psi_2 \land \ldots \land \psi_{k-1} \land \psi_k$ captures the path condition for the execution trace $X$. Further assume that $\{A_0, A_1, \ldots, A_6\}$ is the set of violated assertions and $\pi$ is the shortest path-prefix in $X$ which captures at least one violation of each assertion in the set $\{A_0, A_1, \ldots, A_6\}$. Therefore, an assertion, if violated beyond path-prefix $\pi$, must belong to the set $\{A_0, A_1, \ldots, A_6\}$. If $\varphi$ is the respective path condition, say $\hat{\varphi}$ be the prefix of the path condition corresponding to path-prefix $\pi$. Note that $\varphi \equiv \hat{\varphi}$, 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 $\{A_0, A_1, \ldots, A_6\}$). Therefore, for each
Algorithm 1 Dynamic exploration of instrumented assertions

1: Input:
2: \( P_A \): instrumented program with assertions
3: \( A \): set of instrumented assertions
4: Output:
5: \( T \): a set of test cases, each of which realizes a unique cache thrashing scenario
6: AllPathConditions = unchecked = \( T = \emptyset \)
7: /* build the control dependency graph (CDG) of \( P_A \) */
8: \( \text{CDG}_P_A \leftarrow \text{BuildCDG}(P_A) \)
9: select a random input \( I \)
10: ExecuteAndReport(\( P_A, A, I, \text{CDG}_P_A \))
11: while unchecked \( \neq \emptyset \) \&\& \( A \neq \text{NULL} \) do
12: /* pick a partial path with maximum number */
13: of reachable and non-violated assertions */
14: select \( \langle \varphi, F_{\varphi} \rangle \in \text{unchecked} \) with maximum \( F_{\varphi} \)
15: unchecked := unchecked \( \setminus \{\langle \varphi, F_{\varphi} \rangle\} \)
16: let \( \varphi \leftarrow \psi_1 \land \psi_2 \land \ldots \land \psi_k \land \neg \psi_i \)
17: /* execute an unexplored path */
18: if \( \varphi \) is satisfiable then
19: \( \tau_\varphi \leftarrow \) some concrete inputs satisfying \( \varphi \)
20: ExecuteAndReport(\( P_A, A, \tau_\varphi, \text{CDG}_P_A \))
21: end if
22: end while
23: procedure ExecuteAndReport(\( P_A, A, \tau, \text{CDG}_P_A \))
24: execute \( P_A \) on input \( \tau \)
25: let \( X \) be the execution trace on input \( \tau \)
26: let \( \varphi \equiv \psi_1 \land \psi_2 \land \ldots \land \psi_k \) be the path condition
27: let \( \{A_{\theta_1}, A_{\theta_2}, \ldots, A_{\theta_m}\} \) be the set of violated assertions, on input \( \tau \)
28: \( A = A \setminus \{A_{\theta_1}, A_{\theta_2}, \ldots, A_{\theta_m}\} \)
29: let \( \tau \) be the shortest path-prefix along the execution trace \( X \) that captures at least one violation of each assertion in the set \( \{A_{\theta_1}, A_{\theta_2}, \ldots, A_{\theta_m}\} \)
30: let \( \varphi \) captures the partial path condition corresponding to the path-prefix \( \pi \)
31: /* augment the test suite with witnesses for cache thrashing scenarios */
32: \( T \cup = \{\langle \theta_1, \varphi \rangle, \langle \theta_2, \varphi \rangle, \ldots, \langle \theta_r, \varphi \rangle\} \)
33: /* build all partial path conditions */
34: for \( i \leftarrow 1, k \) do
35: if \( \varphi_i \notin \text{AllPathConditions} \) then
36: AllPathConditions \( \cup = \varphi_i \)
37: let \( b_{\text{end}} \) be the control dependency edge in \( \text{CDG}_P_A \) w.r.t. the branch condition \( \neg \psi_i \)
38: /* compute the number of non-violated assertions (\( \in A \)) reachable from \( b_{\text{end}} */
39: \( F_{\varphi_i} := \text{GuidanceFunction}(b_{\text{end}}) \)
40: if \( F_{\varphi_i} \neq 0 \) then
41: unchecked \( \cup = \{\langle \varphi_i, F_{\varphi_i}\rangle\} \)
42: end if
43: end if
44: end for
45: end procedure

Termination: Algorithm 1 terminates as soon as we obtain a witness (i.e. test case) for each cache thrashing scenario (captured by the condition \( A \neq \text{NULL} \) 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 \text{assert}(C.m3 \leq 0 \lor 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 \( \Theta_i \), if no path
witnessing $\Theta_i$ has been explored - no test case containing $\Theta_i$ appears in the test-suite $T$ computed so far. Otherwise, the entry $<\Theta_i, \Psi_i>$ appears in the test-suite $T$ where $\Psi_i$ captures the set of all inputs which witness $\Theta_i$, and whose paths have been explored already by Algorithm I.

Property 3.2: If Algorithm I terminates, we can guarantee to find all feasible cache thrashing scenarios in any un-interrupted 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 $<\Theta_i, \Psi_i>$ in the generated test suite $T$ reported at the end of Algorithm I. Any solution for the formula $\Psi_i$ is a test input witnessing cache thrashing scenario $\Theta_i$.

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 I. 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.

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.

### Table I: Subject Programs

<table>
<thead>
<tr>
<th>S. No</th>
<th>Test Program</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nsichneu</td>
<td>4253</td>
</tr>
<tr>
<td>2</td>
<td>Papabench</td>
<td>1097</td>
</tr>
<tr>
<td>3</td>
<td>Jetbench</td>
<td>770</td>
</tr>
</tbody>
</table>

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...
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 *nischneu*) 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 *nischneu* has a large code within a loop, we choose a relatively bigger cache for *nischneu*, compared to the other two subject programs. Figure 8(a) shows the percentage coverage for *nischneu*, 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 *nischneu*, was observed to be less than 31% for both the experiments. Note that since the framework achieved a 100% assertion coverage for *nischneu*, 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 $x = 0 \land x = 1$), 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 *nischneu* and *Jetbench* we observed an assertion coverage of almost 100%. In fact, for *nischneu* 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.

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 than 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 \( m_1, m_2, m_3 \) and \( m_4 \) all map to the same cache set. Clearly, this would result in a cache-thrashing scenario (for thrashing sets \( \{m_1, m_2\} \) and \( \{m_3, m_4\} \)) and our test generation framework computes the following test cases: \( \{\{m_1, m_2\}, z \leq 5\} \) and \( \{\{m_3, m_4\}, z > 5\} \). 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 - \( \Delta_1 \) and \( \Delta_2 \) (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 \( I_1 \in \Delta_1 \), either \( m_1 \) or \( m_2 \) 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 \( m_1 \) or \( m_2 \) (as shown in 10(a)) will not improve the cache performance. This is due to the fact that \( m_3 \) and \( m_4 \) 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 \( m_1 \) (or \( m_2 \)) for all inputs satisfying \( z \leq 5 \) and lock \( m_3 \) (or \( m_4 \)) 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 \( \Delta_1 \) and \( \Delta_2 \) (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 ~ 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