MESS: Memory Performance Debugging on Embedded Multi-core Systems

Sudipta Chattopadhyay

Linköping University, Sweden
sudipta.chattopadhyay@liu.se

Abstract. Multi-core processors have penetrated the modern computing platforms in several dimensions. Such systems aim to achieve high-performance via running computations in parallel. However, the performance of such systems is often limited due to the congestion in shared resources, such as shared caches and shared buses. In this paper, we propose MESS, a performance debugging framework for embedded, multi-core systems. MESS systematically discovers the order of memory-access operations that expose performance bugs due to shared caches. We leverage both on single-core performance profiling and symbolic constraint solving to reveal the interleaved memory-access-pattern that leads to a performance bug. Our baseline framework does not generate any *false positive*. Besides, its failure to find a solution highlights the absence of performance bugs due to shared caches, for a given input. Finally, we propose an approximate solution that dramatically reduces debugging time, at the cost of a reasonable amount of false positives. Our experiments with several embedded software and a real-life robot controller suggest that we can discover performance bugs in a reasonable time. The implementation of MESS and our experiments are available at `https://bitbucket.org/sudiptac/mess`.

1 Introduction

It is notoriously difficult to understand and discover performance bugs in software. Whereas performance bugs may appear in any application, these bugs are critical for certain class of software, such as embedded and real-time software. Embedded and real-time applications are, in general, constrained via several temporal requirements. For hard real-time applications, violation of such temporal constraints may lead to catastrophic effects, often costing human lives. Apart from hard real-time applications, the existence of performance bugs may substantially impact the quality of soft real-time applications (*e.g.* media players) as well as web applications. As the computing world is moving towards the multi-core era, it has become a critical problem to develop correct and efficient software on multi-core platforms. In this paper, broadly, we concentrate on the efficiency of applications which run on multi-core platforms.

In multi-threaded execution, software functionality might be disrupted due to the non-deterministic order in accessing shared data [11]. Similarly, the performance of multi-core systems may highly vary due to the non-deterministic order in accessing *shared resources*, such as shared caches. Caches are managed at runtime and they store copies of memory blocks from the main memory. In current generation computing platforms, caches are several magnitudes faster than accessing the main memory. As a result, cache memory is a crucial component to bridge the performance gap between the processor and main memory, and to improve the overall performance of applications. However, since caches are managed at runtime, the order of memory-access patterns play a crucial role in deciding the content of caches. For instance, consider a shared cache which can hold only one memory block. If accesses to m_1 and m_2 are interleaved in parallel, then the ordering $(m_1 \cdot m_2)^*$ will *always* lead to cache misses. In contrast, *only the first* accesses of m_1 and m_2 will suffer cache misses, for the ordering $(m_1^* \cdot m_2^*)$. In summary, depending on the memory-access order, there might be a high variation on cache performance, which dramatically impacts the overall performance of software.

In this paper, we propose a novel approach to discover interleaving patterns that violate a given temporal constraint. For a given program input, our framework *automatically* discovers the order of memory accesses that highlights a performance bug. These bugs happen due to the cache sharing between cores and they may lead to serious performance issues at runtime. A typical usage of our framework is the reproduction of performance bugs on multi-core systems and subsequently, improve the overall performance via classic cache management techniques, such as cache locking [18]. We leverage on the recent advances in constraint solving and *satisfiability modulo theory* (SMT) to systematically explore memory-access patterns. We propose a baseline framework which does not generate *false* alarms. Moreover, if our baseline framework terminates without a solution, then we can guarantee the *validity* of given temporal constraints, for the given input. We also propose an approximation that systematically partitions the set of constraints and solve each partition in parallel. Such a strategy dramatically improves the solver performance. Our approximation guarantees *soundness*, meaning that the absence of a solution highlights the absence of performance bugs. Besides, our evaluation reveals that such an approximation exhibits reasonably low false alarms.

The generation of a performance-stressing interleaving pattern involves many technical challenges. Unlike the functionality of an application, its performance is not directly annotated in the code. Moreover, it is infeasible to execute an application for all possible interleaving patterns, due to an exponential number of possibilities. To resolve such challenges, we propose a compositional approach to discover performance bugs. Our framework broadly contains two stages. In the first stage, we monitor the performance of each core *in isolation* and compute a performance-summary for each core. In each performance-summary, the timing to access the shared cache is replaced by a symbolic variable. In the second stage, we formulate constraints that relate the order of memory accesses with the delay to access the shared-cache. In particular, we formulate constraints that *symbolically encode necessary and sufficient conditions for a memory block to be evicted from the shared-cache*. As a result, using these constraints, we could determine whether a given memory block is available in the shared-cache, when it is being accessed. In other words, we can use such constraints to bound the delay to access the shared-cache and thereby, constraining the value of symbolic variables, which were introduced in the first stage of our framework. Finally, the temporal constraint is also provided as a quantifier-free formula. All the constraints, together with the temporal constraint, is given to an SMT solver. If the solver finds a solution, the resulting solution highlights an interleaving pattern that violates the temporal constraint. Since SMT technology is continuously evolving, we believe that such a compositional approach will be appealing to discover performance bugs in multi-core systems.

To tackle the complexity of our systems, we also propose an approximate solution that significantly improves the performance of our proposed framework. For shared caches, we observed that the set of all constraints can be partitioned systematically to solve in parallel. The general intuition is to consider partitions of memory accesses which can contend in the shared-cache and solve the constraints generated for each partition independently. By increasing the size of each partition, the designer can reduce the number of *false positives* at the cost of debugging time. Therefore, our framework gives designer the flexibility to fine tune the precision, with respect to debugging time.

Contribution In summary, we propose a performance debugging framework that exposes performance issues due to shared caches. We leverage on single-core performance profiling and symbolic-constraint solving, in order to discover the interleaving pattern that violates a given temporal constraint. Our baseline framework does not generate any *false positive* and it can also be used to prove the absence of performance bugs for a given input. Moreover, for time-critical code fragments, our baseline framework can be employed to derive the worst-case interleaving pattern (in terms of shared-cache performance), for a given input. To tackle the complexity of our constraint-based framework, we have also proposed an approximation that dramatically increases the solver performance. To show the generality of our approach, we have instantiated our framework for two different caches *(i)* caches with least-recently-used (LRU) replacement policy and *(ii)* caches with first-in-first-out (FIFO) policy. We have implemented our entire framework on top of `simplescalar` [6] – an open-source, cycle-accurate, processor simulator and `Z3` [5] – an open source, SMT solver. Our experiments with several embedded software reveals the effectiveness of our approach. For instance, our baseline framework was able to check a variety of temporal constraints for a real-life robot controller [2] within *3 minutes* and our approximation took only *20 seconds* on average to check the same set of constraints. This makes the idea of constraint-based formulation in performance debugging quite appealing for research in future.

2 Overview

Background on caches Caches are employed between the CPU and the main memory (DRAM) to bridge the performance gap between the CPU and the DRAM. A cache can be described as a three tuple $\langle \mathcal{A}, \mathcal{S}, \mathcal{L} \rangle$, where $\mathcal{A}$ is the associativity of the cache, $\mathcal{S}$ is the number of cache sets and $\mathcal{L}$ is the line size (in bytes). Each cache set can hold $\mathcal{A}$ cache lines, leading to a total cache size of $(\mathcal{A} \cdot \mathcal{S} \cdot \mathcal{L})$ bytes. When $\mathcal{A} = 1$, the respective caches are called to be *directly mapped*. Data is fetched into caches at the granularity of line size ($\mathcal{L}$). Therefore, for an arbitrary memory address x, $\mathcal{L}$ contiguous bytes are fetched into the cache starting from address $\lfloor \frac{x}{\mathcal{L}} \rfloor$ and we say that x belongs to the *memory block* $\lfloor \frac{x}{\mathcal{L}} \rfloor$. The number of cache sets ($\mathcal{S}$) decides the location where a particular memory block would be placed in the cache. For instance, a memory block, starting at address M, is always mapped to the cache set $M \ mod \ \mathcal{S}$. Since each cache set can hold only

$\mathcal{A}$ cache lines, a cache line needs to be replaced when the number of memory blocks mapping to a cache set exceeds $\mathcal{A}$. In order to accomplish this, a replacement policy is employed when $\mathcal{A} \geq 2$. In this paper, we instantiate our framework for two widely used replacement policies – LRU and FIFO. In LRU policy, the memory block, that was not *accessed* for the longest period of time, is replaced from the cache to make room for other memory blocks. In FIFO policy, the memory block, which is *residing* in the cache for the longest period of time, is replaced to make room for other blocks. In general, the performance of a cache may greatly depend on the underlying replacement policy.

Terminologies We use the following terminologies on caches throughout the paper.

1. *memory block:* For an arbitrary memory reference to address x, we say that it belongs to memory block $\lfloor \frac{x}{\mathcal{L}} \rfloor$ ($\mathcal{L}$ is the line size of cache, in bytes), in order to distinguish different cache lines.
2. *cache hit/miss:* For an arbitrary memory reference, we say that it is a cache hit (miss) if the referenced memory block is found (not found) in the cache.
3. *cache conflict:* Two memory blocks M_1 and M_2 conflict in the cache if they map to the same cache set. In other words, M_1 is conflicting to M_2 (and vice versa). These conflicting memory blocks might be accessed within the same core (intra-core) or across different cores (inter-core).
4. *cache-set state:* Ordered $\mathcal{A}$-tuple capturing the content of a cache set. For instance, $\langle m_1, m_2 \rangle$ captures such a tuple for caches with associativity 2. The relative position of a memory block in the tuple decides the number of unique cache conflicts required to evict the same from the cache. For instance, in $\langle m_1, m_2 \rangle$, m_1 requires two unique cache conflicts to be evicted from the cache, whereas m_2 requires only one. The generation of cache conflicts critically depends on the replacement policy and the order of memory accesses.

Motivation and challenges Figure 1 captures an example where two programs are executing in parallel on different processor cores and sharing a cache. For the sake of simplicity, let us assume that all the instructions in both `Program x` and `Program y` access the same shared-cache set. In Figure 1(a), the memory block accessed by each instruction is shown within the brackets. In the following discussion, we shall capture the location i of `Program x` via x^i and the same of `Program y` via y^i.

Let us assume a cache with associativity ($\mathcal{A}$) two and employing FIFO replacement policy. Further assume that we want to check whether *all instructions in both programs can face cache misses*. Figure 1(b) captures an interleaving pattern which leads to 100% cache misses in both programs. The progression of the cache content for this interleaving pattern is captured via Figure 1(c). It is worthwhile to note that many interleaving patterns will fail to generate 100% cache misses in both programs. Figure 1(d) captures one such interleaving pattern. As a result, if the set of memory accesses (*cf.* Figure 1(a)) appears within a loop, the memory-access delay might change dramatically depending on the interleaving pattern. The respective cache contents for the interleaving pattern in Figure 1(d) are shown via Figure 1(e). In general, it is infeasible to perform an exhaustive search over the set of all possible interleaving patterns, due to an exponential number of possibilities. As a result, a systematic method is required to check performance-related constraints, in the context of multi-core systems.

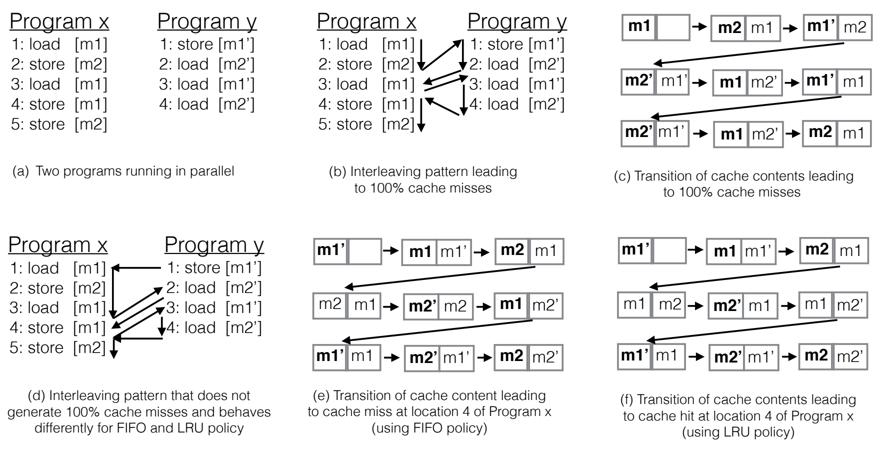

Program x
1: load [m1]
2: store [m2]
3: load [m1]
4: store [m1]
5: store [m2]
Program y
1: store [m1']
2: load [m2']
3: load [m1']
4: load [m2']
(a) Two programs running in parallel
(b) Interleaving pattern leading to 100% cache misses
(c) Transition of cache contents leading to 100% cache misses
(d) Interleaving pattern that does not generate 100% cache misses and behaves differently for FIFO and LRU policy
(e) Transition of cache content leading to cache miss at location 4 of Program x (using FIFO policy)
(f) Transition of cache contents leading to cache hit at location 4 of Program x (using LRU policy)

Fig. 1. An example showing the impact of interleaving pattern on shared-cache performance. The direction of an arrow captures the *happens-before* relation. Cache misses are highlighted in bold.

Let us now assume that we want to check whether location x^4 can face a *cache miss*. Such a behaviour can also take place only for a few interleaving patterns. Figure 1(d) captures an interleaving pattern which lead to a cache miss at location x^4 (*cf.* Figure 1(e) for the transition of cache contents). Unfortunately, if we replay the same interleaving pattern for LRU replacement policy, it will not lead to a cache miss at location x^4. This behaviour is captured via Figure 1(f), which demonstrates the modification of cache contents in the presence of LRU policy. This shows the influence of the *cache replacement policy* to check or invalidate temporal constraints.

To summarize, due to the presence of shared caches in multi-core systems, it is challenging to check the validity of temporal constraints or reproduce any violation of temporal constraints in a production run. This phenomenon occurs due to the non-determinism in the order of interleaved memory-accesses, which, in turn leads to non-determinism in cache contention and variability in memory-access delay. In the following, we shall give an outline of our performance debugging framework.

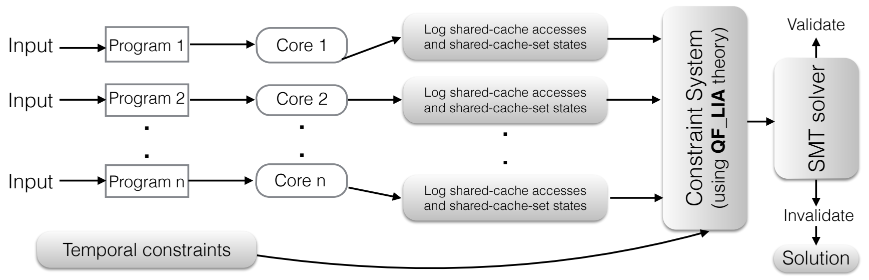

Input
Program 1
Core 1
Input
Program 2
Core 2
Input
Program n
Core n
Log shared-cache accesses and shared-cache-set states
Log shared-cache accesses and shared-cache-set states
Log shared-cache accesses and shared-cache-set states
Temporal constraints
Constraint System (using QF_LIA theory)
SMT solver
Validate
Invalidate
Solution

Fig. 2. Performance debugging framework for multi-core systems

Overall framework Figure 2 outlines the overall design of MESS. For a given input to each program running in parallel, our framework is used to check the temporal constraints. We first monitor the execution on each core *in isolation*, ignoring any interference from other cores. At the end of this monitoring phase, we obtain a sequence of shared-cache accesses $\langle i^1, i^2, \ldots, i^{\mathcal{V}_i - 1}, i^{\mathcal{V}_i} \rangle$ for each core i, where $\mathcal{V}_i$ is the total number of shared-cache accesses by core i. We also collect the shared-cache-set states at these access points. Using the information obtained from the monitoring phase, we build a constraint system via the theory of quantifier-free linear integer arithmetic (QF_LIA). Intuitively, this constraint system relates the order of memory accesses with the delay to access the shared cache. The size of our constraint system is *polynomial*, with respect to the number of accesses to the shared cache. Finally, the temporal constraint can be provided to the constraint system via quantifier-free predicates. The entire constraint system, along with the temporal constraints, is provided to an SMT solver. If the constraint system is *satisfiable*, then the solution returned by the SMT solver captures an interleaving pattern that *violates* certain temporal constraints. This solution can further be used for debugging performance on multi-core systems.

System model We assume a sequentially-consistent, multi-core system where *each core may have several levels of private caches and only the last-level cache is shared across cores*. Therefore, a shared-cache miss will lead to an access to the slow DRAM. Such a design of memory-hierarchy is typical in embedded multi-core processors [1]. In this paper, we do not address the problem of cache coherency and any cache misses resulting from the same. Such cache misses might appear due to the invalidation of cache lines that hold outdated data. In summary, we first assume that programs, running on different cores, have disjoint memory spaces. We argue that, even in the absence of cache coherency, debugging shared-cache performance is sufficiently complex. In Section 4, we discuss the required modifications in our framework in the presence of data sharing.

3 Methodologies

In this section, we shall introduce the formal foundation of our framework. Recall that the outcome of our framework is to compute a memory-access ordering, leading to a performance bug. This ordering is captured among all accesses to the shared cache.

Let us assume that we have a total of $\mathcal{N}$ cores, each of which might exhibit a different sequence of shared-cache accesses. We use the notation i^j to capture the j-th shared-cache access by i-th core and $\mathcal{V}_i$ to capture the total number of shared-cache accesses by core i. We also use the following notations in our framework:

- σ_i^j : The memory block accessed by the shared-cache access i^j.
- $\pi(m)$: Cache set where memory block m is mapped.
- ζ_i^j : Shared-cache-set state for cache set $\pi(\sigma_i^j)$, immediately before the access i^j.
- $\mathcal{C}_i^j$: The set of memory blocks, other than σ_i^j, mapping to the same cache set as σ_i^j in the shared cache. Therefore, for any $m' \in \mathcal{C}_i^j$, we have $m' \neq \sigma_i^j$ and $\pi(m') = \pi(\sigma_i^j)$.
- $\mathcal{O}_i^j$: The position of the shared-cache access i^j in the ordering among all accesses to the shared cache.
- δ_i^j : The delay suffered by the shared-cache access i^j.

For instance, in Figure 1(b), $\sigma_x^1 = m1$, $\sigma_y^1 = m1'$, $\zeta_y^1 = \langle m2, m1 \rangle$ and the interleaving pattern is captured as follows: $O_x^1 < O_x^2 < O_y^1 < O_y^2 < O_x^3 < O_y^3 < O_y^4 < O_x^4 < O_x^5$. The outcome of our framework is such an interleaving pattern.

Profiling each core in isolation As outlined in the preceding section, our framework initially records the performance of each core *in isolation*. The primary purpose of this recording phase is to accurately identify accesses to the shared cache, for each core. Therefore, while profiling each core in isolation, ζ_i^j contains memory blocks accessed *only* within core i and ignores all memory blocks accessed within core $\bar{i} \neq i$.

Let us assume age_i^j denotes the relative position of σ_i^j within ζ_i^j, while profiling each core in isolation. If $\sigma_i^j \notin \zeta_i^j$ (*i.e.* i^j suffers a shared-cache miss), we assign $\mathcal{A}+1$ to age_i^j, where $\mathcal{A}$ is the associativity of the shared-cache. Subsequently, for each core i, we encode a performance-summary α_i as a sequence of pairs. Each such pair captures a shared-cache access i^j, along with age_i^j as follows:

$$\alpha_i \equiv \langle (i^1, age_i^1), (i^2, age_i^2), \ldots, (i^{\mathcal{V}_i - 1}, age_i^{\mathcal{V}_i - 1}), (i^{\mathcal{V}_i}, age_i^{\mathcal{V}_i}) \rangle \tag{1}$$

For any shared-cache access i^j, it is a shared-cache miss if and only if $\sigma_i^j \notin \zeta_i^j$, leading age_i^j being set to $\mathcal{A}+1$. Such a cache miss can happen because of the following reasons: ***(i)*** σ_i^j was accessed for the first time, or ***(ii)*** σ_i^j was evicted from the shared-cache by some other memory block. Recall that programs running on different cores have disjoint memory spaces. As a result, while profiling each core in isolation, we can accurately identify shared-cache misses when σ_i^j was accessed for the first time. This is because, σ_i^j was not accessed by any other core except core i. Next, we describe our constraint system (using QF_LIA theory), which formulates necessary and sufficient conditions for evicting memory blocks from the shared-cache, leading to shared-cache misses.

Program order constraints These constraints are generated to capture the program order on each core. Note that $\langle i^1, i^2, \ldots, i^{\mathcal{V}_i - 1}, i^{\mathcal{V}_i} \rangle$ captures the sequence of shared-cache accesses by core i. Therefore, the following constraints are generated to capture the program order semantics (note that any partial ordering between shared-cache accesses across cores, if exists, can be captured in a similar fashion).

$$\Theta_{order} \equiv \bigwedge_{i \in [1, \mathcal{N}]} \; \bigwedge_{j \in [2, \mathcal{V}_i]} \left(\mathcal{O}_i^j > \mathcal{O}_i^{j-1} \right) \tag{2}$$

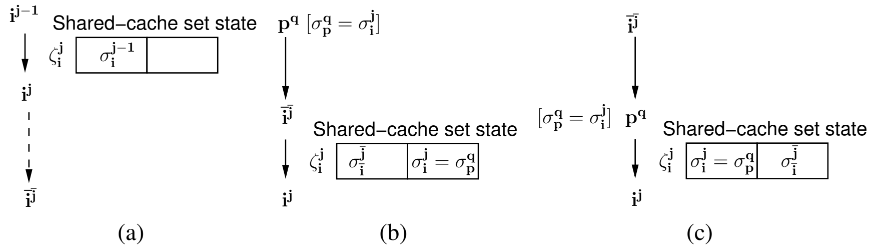
Program-order constraints are generated irrespective of the cache replacement policy. In the following, we now instantiate the constraint formulation for LRU and FIFO policies.

3.1 Constraint system for LRU caches

A shared-cache access i^j is a cache hit if and only if ζ_i^j contains σ_i^j. Otherwise, i^j suffers a shared-cache miss. Therefore, to accurately determine the shared-cache performance, it is crucial to track all feasible states of ζ_i^j. We accomplish this by relating the order of memory accesses with the changes in cache-set states. In order to understand the relationship between the memory-access order and cache-set states, we first define the notion of cache-conflict generation between two shared-cache accesses.

Definition 1 *(Cache Conflict Generation) Consider a shared-cache access $\bar{i}^{\bar{j}}$, which requests memory block $\bar{m}$ (*i.e. $\sigma_{\bar{i}}^{\bar{j}} = \bar{m}$). A shared-cache access $\bar{i}^{\bar{j}}$ generates (cache) conflict to i^j, only if accessing $\bar{m}$ at $\bar{i}^{\bar{j}}$ can affect the relative position of σ_i^j within ζ_i^j. For instance, in Figure 1(d), accesses to $m1'$ and $m2'$ do not generate cache conflict to x^3, but an access to $m2$ does (at x^2).*

We introduce a variable $\Psi_i^j(\bar{m})$ to capture whether any access to memory block $\bar{m}$ generates conflict to the shared-cache access i^j. As stated in Definition 1, the memory block $\bar{m}$ might be accessed more than once and therefore, the formulation of $\Psi_i^j(\bar{m})$ must consider all possible places where $\bar{m}$ was accessed. Consider one such place $\bar{i}^{\bar{j}}$, where $\bar{m}$ was accessed. Therefore, $\sigma_{\bar{i}}^{\bar{j}} = \bar{m}$. Figure 3 illustrates different scenarios in LRU policy, with respect to the generation of cache conflicts.



i^{j-1}
Shared–cache set state
ζ_i^j
σ_i^{j-1}
i^j
$\bar{i}^{\bar{j}}$
p^q $[\sigma_p^q = \sigma_i^j]$
Shared–cache set state
$\sigma_{\bar{i}}^{\bar{j}}$
$\sigma_i^j = \sigma_p^q$
$[\sigma_p^q = \sigma_i^j]$
(a)
(b)
(c)

Fig. 3. The direction of arrow captures the total order between accesses to the shared cache. The left-most position in ζ_i^j captures the *most recently used* memory block. (a) $\bar{i}^{\bar{j}}$ cannot affect shared-cache set state ζ_i^j and therefore, it cannot generate cache conflict to i^j, if $\bar{i}^{\bar{j}}$ *happens after* i^j, (b) $\bar{i}^{\bar{j}}$ can affect ζ_i^j only if $\bar{i}^{\bar{j}}$ *happens before* i^j, (c) shared-cache access p^q accesses the same memory block as that of i^j (*i.e.* $\sigma_p^q = \sigma_i^j$) and therefore, access $\bar{i}^{\bar{j}}$ cannot affect the relative position of σ_i^j within ζ_i^j.

In particular, Figures 3(a)-(b) capture the *happens-before* relationship between accesses $\bar{i}^{\bar{j}}$ and i^j. It is impossible for $\bar{i}^{\bar{j}}$ to affect the cache-set state ζ_i^j, if i^j happens before $\bar{i}^{\bar{j}}$. Moreover, if the memory block σ_i^j is accessed after $\bar{i}^{\bar{j}}$ and before i^j, then such an access will hide the cache conflict between $\bar{i}^{\bar{j}}$ and i^j. Figure 3(c) captures one such situation, where shared-cache access p^q accesses the memory block σ_i^j and prevents $\bar{i}^{\bar{j}}$ to affect the relative position of σ_i^j within cache-set state ζ_i^j.

In the following, we describe the formulation of constraints for an arbitrary shared-cache access i^j. The primary purpose of these constraints is to compute the delay δ_i^j. Considering the intuition provided in Figure 3, we can state that a shared-cache access $\bar{i}^{\bar{j}}$ generates conflict to the shared-cache access i^j, only if the following conditions hold:

– $\psi_{cft}^{lru}\left(\bar{i}^{\bar{j}}, i^j\right)$: Shared-cache access $\bar{i}^{\bar{j}}$ *happens before* the shared-cache access i^j. Therefore, $\mathcal{O}_{\bar{i}}^{\bar{j}} < \mathcal{O}_i^j$. This is illustrated via Figures 3(a)-(b).
– $\psi_{ref}^{lru}\left(\bar{i}^{\bar{j}}, i^j\right)$: There does not exist any shared-cache access p^q, such that p^q accesses memory block σ_i^j from the shared-cache, p^q *happens before* i^j and $\bar{i}^{\bar{j}}$ *happens before* p^q. Therefore, for any shared-cache access p^q, where $\sigma_p^q = \sigma_i^j$, conditions $\mathcal{O}_p^q < \mathcal{O}_i^j$ and $\mathcal{O}_{\bar{i}}^{\bar{j}} < \mathcal{O}_p^q$ cannot be satisfiable together. Otherwise, note that p^q will hide the cache conflict between $\bar{i}^{\bar{j}}$ and i^j, as illustrated via Figure 3(c).

$\psi_{cft}^{lru}\left(\bar{i}^{\bar{j}}, i^j\right)$ and $\psi_{ref}^{lru}\left(\bar{i}^{\bar{j}}, i^j\right)$ can be formalized via the following constraints:

$$\psi_{cft}^{lru}\left(\bar{i}^{\bar{j}}, i^j\right) \equiv \mathcal{O}_{\bar{i}}^{\bar{j}} < \mathcal{O}_i^j \tag{3}$$

$$\psi_{ref}^{lru}\left(\bar{i}^{\bar{j}}, i^j\right) \equiv \bigwedge_{p,q:\ \sigma_p^q = \sigma_i^j} \neg\left(\mathcal{O}_{\bar{i}}^{\bar{j}} < \mathcal{O}_p^q \wedge \mathcal{O}_p^q < \mathcal{O}_i^j\right) \tag{4}$$

We combine Constraint (3) and Constraint (4) to formulate the generation of shared-cache conflict. Recall that $\mathcal{C}_i^j$ captures the set of memory blocks that map to the same shared-cache set as σ_i^j. Therefore, Constraints (3)-(4) need to be generated for each memory block in $\mathcal{C}_i^j$. Formally, for each shared-cache access i^j, we generate the following constraints to capture cache conflicts generated across cores.

$$\Theta_1^{lru}(i,j) \equiv \bigwedge_{\bar{i} \neq i : \sigma_{\bar{i}}^{\bar{j}} \in \mathcal{C}_i^j} \left(\left(\psi_{cft}^{lru}\left(\bar{i}^{\bar{j}}, i^j\right) \wedge \psi_{ref}^{lru}\left(\bar{i}^{\bar{j}}, i^j\right) \right) \Rightarrow \left(\Psi_i^j\left(\sigma_{\bar{i}}^{\bar{j}}\right) = 1 \right) \right) \tag{5}$$

The absence of inter-core cache conflict is captured via the negation of Constraint (5). In particular, for any memory block $\bar{m} \in \mathcal{C}_i^j$, we need to consider the set of locations $\bar{i}^{\bar{j}}$ where $\bar{m}$ is accessed (*i.e.* $\sigma_{\bar{i}}^{\bar{j}} = \bar{m}$). If none of these locations satisfy either Constraint (3) or Constraint (4), we can conclude that accesses to memory block $\bar{m}$ do not generate any cache conflict to shared-cache access i^j. This behaviour can be captured via the following constraints:

$$\Theta_0^{lru}(i,j) \equiv \bigwedge_{\bar{m} \in \mathcal{C}_i^j} \left(\bigwedge_{\bar{i} \neq i : \sigma_{\bar{i}}^{\bar{j}} = \bar{m}} \left(\neg\psi_{cft}^{lru}\left(\bar{i}^{\bar{j}}, i^j\right) \vee \neg\psi_{ref}^{lru}\left(\bar{i}^{\bar{j}}, i^j\right) \right) \Rightarrow \left(\Psi_i^j\left(\bar{m}\right) = 0 \right) \right) \tag{6}$$

Finally, we need to link Constraints (5)-(6) to the absolute latency suffered by shared-cache access i^j (*i.e.* δ_i^j). Let us assume HIT and MISS capture the shared-cache hit latency and miss penalty, respectively. To compute the latency, we need to check whether the set of cache conflicts generated at i^j could evict the memory block σ_i^j. Therefore, we generate the following constraints to formulate the delay suffered at location i^j.

$$\Theta_{miss}^{lru}(i,j) \equiv \left(\sum_{\bar{i} \neq i:\ \sigma_{\bar{i}}^{\bar{j}} \in \mathcal{C}_i^j} \Psi_i^j(\sigma_{\bar{i}}^{\bar{j}}) \geq \mathcal{A} - age_i^j + 1 \right) \Rightarrow (\delta_i^j = \mathit{MISS}) \tag{7}$$

$$\Theta_{hit}^{lru}(i,j) \equiv \left(\sum_{\bar{i} \neq i:\ \sigma_{\bar{i}}^{\bar{j}} \in \mathcal{C}_i^j} \Psi_i^j(\sigma_{\bar{i}}^{\bar{j}}) \leq \mathcal{A} - age_i^j \right) \Rightarrow (\delta_i^j = \mathit{HIT}) \tag{8}$$

age_i^j denotes the relative position of σ_i^j within ζ_i^j and age_i^j=$\mathcal{A}$+1, if $\sigma_i^j \notin \zeta_i^j$. The value age_i^j was collected while profiling each core *in isolation* (*cf.* Equation (1)). Therefore, age_i^j already captures cache conflicts generated within core i and the quantity $\left(\mathcal{A} - age_i^j + 1\right)$ captures the minimum number of unique, inter-core cache conflicts (as formulated via Constraint (5)) to evict σ_i^j from the shared cache.

3.2 Constraint system for FIFO caches

Unlike LRU policy, cache-set state remains unchanged for all *cache hits* in FIFO policy (*cf.* Figure 1(e)). As a result, the necessary conditions to generate cache conflicts (*cf.* Constraints (3)-(4)) need to be modified for FIFO policy.

To illustrate the difference between LRU and FIFO policy, let us consider the scenarios in Figure 4. For instance, in Figure 4(a), shared-cache access $\bar{i}^{\bar{j}}$ *happens before*

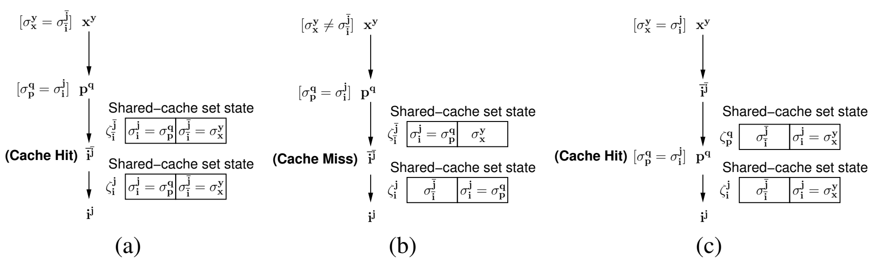


Fig. 4. The direction of arrow captures the total order between accesses to the shared cache. The left-most position in ζ_i^j captures the *most recent memory block inserted* into ζ_i^j. (a) $\bar{i}^{\bar{j}}$ cannot affect shared-cache set state ζ_i^j as $\bar{i}^{\bar{j}}$ is a cache hit. Therefore, $\bar{i}^{\bar{j}}$ cannot generate cache conflict to i^j, (b) $\bar{i}^{\bar{j}}$ can affect ζ_i^j only if $\bar{i}^{\bar{j}}$ *happens before* i^j and it is a cache miss, (c) shared-cache access p^q accesses the same memory block as that of i^j (*i.e.* $\sigma_p^q = \sigma_i^j$), however, p^q is a cache hit. Therefore, p^q cannot hide the cache conflict generated between $\bar{i}^{\bar{j}}$ and i^j.

the access i^j. However, $\bar{i}^{\bar{j}}$ cannot affect the relative position of σ_i^j within ζ_i^j and therefore, $\bar{i}^{\bar{j}}$ cannot generate cache conflict to i^j (*cf.* Definition 1). It is worthwhile to note that, $\bar{i}^{\bar{j}}$ would have generated conflict to i^j, in the presence of LRU policy. Figure 4(b) captures a scenario, where $\bar{i}^{\bar{j}}$ was a cache miss, leading to the generation of cache conflict to i^j. Recall that, for LRU policy, if the memory block σ_i^j was accessed between $\bar{i}^{\bar{j}}$ and i^j, then $\bar{i}^{\bar{j}}$ could not generate cache conflict to i^j (*cf.* Constraint (4)). However in FIFO policy, as shown in Figure 4(c), even though access p^q references σ_i^j and it occurs between $\bar{i}^{\bar{j}}$ and i^j, p^q cannot hide the cache conflict between $\bar{i}^{\bar{j}}$ and i^j. This is because p^q was a cache hit and therefore, it does not affect the relative position of σ_i^j within ζ_i^j.

In summary, a shared-cache access must be a cache miss if it affects the cache-set state ζ_i^j. In order to realize this intuition, we formulate the following constraints, which capture the necessary conditions for $\bar{i}^{\bar{j}}$ generating cache conflict to i^j.

$$\psi_{cft}^{fifo}\left(\bar{i}^{\bar{j}}, i^j\right) \equiv \left(\mathcal{O}_{\bar{i}}^{\bar{j}} < \mathcal{O}_i^j\right) \wedge \left(\delta_{\bar{i}}^{\bar{j}} = MISS\right) \tag{9}$$

$$\psi_{ref}^{fifo}\left(\bar{i}^{\bar{j}}, i^j\right) \equiv \bigwedge_{p,q:\ \sigma_p^q = \sigma_i^j} \neg\left(\left(\mathcal{O}_{\bar{i}}^{\bar{j}} < \mathcal{O}_p^q\right) \wedge \left(\mathcal{O}_p^q < \mathcal{O}_i^j\right) \wedge \left(\delta_p^q = MISS\right)\right) \tag{10}$$

Constraint (9) ensures that $\bar{i}^{\bar{j}}$ incurs a cache miss, in order to generate cache conflict to i^j (*cf.* Figures 4(a)-(b)). Similarly, Constraint (10) ensures that access p^q needs to be a cache miss to hide the cache conflict between $\bar{i}^{\bar{j}}$ and i^j (*cf.* Figure 4(c)).

The outcome of Constraints (9)-(10) may depend on the interleaving pattern, even within a single core (*i.e.* $\bar{i} = i$). This is because, values of $\delta_{\bar{i}}^{\bar{j}}$ and δ_p^q may depend on the interleaving pattern. As a result, the generation of cache conflicts, even within a core, may be affected with FIFO policy. Hence, unlike LRU policy, we need to formulate cache conflict both within a core and across cores. This is accomplished by modifying Constraints (5)-(6), so that the resulting constraints also consider cache conflicts within cores. In particular, we remove the condition $\bar{i} \neq i$ from Constraints (5)-(6) as follows.

$$\Theta_1^{fifo}(i,j) \equiv \bigwedge_{\bar{i},\bar{j}:\sigma_{\bar{i}}^{\bar{j}} \in \mathcal{C}_i^j} \left(\left(\psi_{cft}^{fifo}\left(\bar{i}^{\bar{j}}, i^j\right) \wedge \psi_{ref}^{fifo}\left(\bar{i}^{\bar{j}}, i^j\right)\right) \Rightarrow \left(\Psi_i^j\left(\sigma_{\bar{i}}^{\bar{j}}\right) = 1\right)\right) \tag{11}$$

$$\Theta_0^{fifo}(i,j) \equiv \bigwedge_{\bar{m} \in \mathcal{C}_i^j} \left(\bigwedge_{\bar{i},\bar{j}:\sigma_{\bar{i}}^{\bar{j}}=\bar{m}} \left(\neg\psi_{cft}^{fifo}\left(\bar{i}^{\bar{j}}, i^j\right) \vee \neg\psi_{ref}^{fifo}\left(\bar{i}^{\bar{j}}, i^j\right)\right) \Rightarrow \left(\Psi_i^j\left(\bar{m}\right) = 0\right) \right) \tag{12}$$

Finally, we link Constraints (11)-(12) to compute the memory-access latency. Intuitively, we check whether the total amount of cache conflict can evict the memory block accessed by i^j. This can be formalized via the following constraints.

$$\Theta_{miss}^{fifo}(i,j) \equiv \left(\left(\sum_{\bar{m} \in \mathcal{C}_i^j} \Psi_i^j(\bar{m}) \geq \mathcal{A} \right) \vee \left(age_i^j = \mathcal{A}+1 \right) \right) \Rightarrow (\delta_i^j = \mathit{MISS}) \tag{13}$$

$$\Theta_{hit}^{fifo}(i,j) \equiv \left(\left(\sum_{\bar{m} \in \mathcal{C}_i^j} \Psi_i^j(\bar{m}) < \mathcal{A} \right) \wedge \left(age_i^j \neq \mathcal{A}+1 \right) \right) \Rightarrow (\delta_i^j = \mathit{HIT}) \tag{14}$$

$\mathcal{A}$ is the associativity of the cache. Recall that age_i^j=$\mathcal{A}$+1, if $\sigma_i^j \notin \zeta_i^j$ and age_i^j was measured while investigating each core in isolation (*cf.* Equation 1). Therefore, the condition age_i^j=$\mathcal{A}$+1 guarantees to include the first-ever cache miss of σ_i^j. Once σ_i^j enters the cache, it takes at least $\mathcal{A}$ unique cache-conflicts to evict it from the cache. $\sum_{\bar{m} \in \mathcal{C}_i^j} \Psi_i^j(\bar{m})$ accounts all unique cache-conflicts faced by σ_i^j, since it enters the cache and till i^j. Therefore, Constraint (13) precisely captures all possibilities of a cache miss at i^j. The violation of Constraint (13) will result in a cache hit at i^j, as shown in Constraint (14).

Providing temporal constraints For embedded software, temporal constraints can be provided in the form of an assertion. Therefore, our framework will search for an ordering on symbolic variables $\mathcal{O}_i^j$ that violates such assertions. In particular, we consider assertions that check the execution time against a threshold τ. In our framework, the non-determinism in timing behaviour appears due to the accesses to shared caches. Therefore, in our evaluation, we search for a solution that satisfy the following constraint: $\left(\sum_{i,j} \delta_i^j \geq \tau\right)$. Recall that δ_i^j symbolically captures the delay suffered by shared-cache access i^j. It is worthwhile to note that we can also check the timing behaviour of a code fragment, instead of checking the same for the entire system. In such cases, we consider only a subset of δ_i^j variables relating to the code fragment.

Putting it all together Our formulated constraints, along with the temporal constraint, is provided to an off-the-shelf SMT solver. As a result, any ongoing and future improvements in the solver technology will directly boost the efficiency of our approach. The SMT solver searches for a satisfying solution of the following constraints:

$$\Phi \equiv \Theta_{order} \wedge \bigwedge_{i,j} \left(\Theta_1^x(i,j) \wedge \Theta_0^x(i,j) \wedge \Theta_{miss}^x(i,j) \wedge \Theta_{hit}^x(i,j)\right) \wedge \left(\sum_{i,j} \delta_i^j \geq \tau \right) \tag{15}$$

where $x \in \{lru, fifo\}$, depending on the cache replacement policy. The solution of the solver captures concrete values of symbolic variables $\mathcal{O}_i^j$ that satisfy Φ. Such concrete values can be used to derive the total order among all accesses to the shared-cache.

Complexity of constraints The complexity of our constraints Φ (*cf.* Constraint (15)) is dominated by the number of constraints to formulate cache conflicts. For instance, in LRU policy, Constraints (5)-(6) dominate the total number of constraints. Let us assume that the total number of shared-cache accesses across all cores is $\mathcal{K}$. Therefore, the size of Constraints (2) has a complexity of $O(\mathcal{K})$. Similarly, the total size of Constraints (7)-(8), for LRU policy (respectively, the total size of Constraints (13)-(14) for FIFO policy) has a size of $O(\mathcal{K})$. Finally, during the formulation of cache conflict, each shared-cache access can be compared with all conflicting shared-cache accesses. Therefore, $\Theta_1^{lru}(i,j)$, $\Theta_0^{lru}(i,j)$, $\Theta_1^{fifo}(i,j)$ and $\Theta_0^{fifo}(i,j)$ have a worst-case size-complexity $O(\mathcal{K}^2)$. Since there exists a total of $\mathcal{K}$ shared-cache accesses, the total size of Constraints (5)-(6) has a complexity of $O(\mathcal{K}^3)$. Putting everything together, our constraint system has a worst-case size-complexity $O(\mathcal{K}^3)$. However, our evaluation reveals that the size of our constraint system is substantially lower than the worst-case complexity.

3.3 Approximate solution

Our approximation scheme aims to reduce the pressure on the constraint solver by reducing the number of constraints to be solved together. The general intuition of our approximation is based on the design principle of caches. In particular, we leverage the fact that two different cache sets never interfere with each other, in terms of cache conflict. Therefore, we model the constraints for each cache set separately and solve them in parallel. In the following, we shall formalize the concept.

Finding a slice of constraints The key idea for the approximation is to find a slice of constraints that could be solved *independently*. Recall that the symbolic variable δ_i^j captures the delay suffered by shared-cache access i^j. It is worthwhile to note that the memory block accessed at i^j (*i.e.* σ_i^j) can be evicted from the shared-cache only by memory blocks conflicting to σ_i^j. A memory block $\bar{m}$ conflicts to σ_i^j in the cache if and only if $\bar{m}$ and σ_i^j map to the same cache set. Therefore, we first group shared-cache accesses with respect to different cache sets and generate the respective constraints. For instance, consider that we are generating constraints with respect to cache set s. We shall use $\pi(m)$ to capture the cache set in which memory block m is mapped.

We slice out the program-order constraints by considering only the memory blocks which map to cache set s. Therefore, the set of program-order constraints, with respect to cache set s, can be defined as follows.

$$\Gamma_{order}(s) \equiv \bigwedge_{i \in [1,\mathcal{N}]} \left(\bigwedge_{j,k \in [1,\mathcal{V}_i]:\ j<k \wedge \left(\pi(\sigma_i^j)=\pi(\sigma_i^k)=s\right) \wedge (\forall m \in [j+1,k):\ \pi(\sigma_i^j) \neq \pi(\sigma_i^m))} \mathcal{O}_i^k > \mathcal{O}_i^j \right) \tag{16}$$

Let us now consider LRU cache replacement policy. The set of constraints, with respect to cache set s, considers constraints that only influence the memory blocks mapped to cache set s. Therefore, for cache set s, we extract the constraints formulated in Equations (5)-(8) as follows.

$$\Gamma_1^{lru}(s) \equiv \bigwedge_{i,j:\pi(\sigma_i^j)=s} \Theta_1^{lru}(i,j); \quad \Gamma_0^{lru}(s) \equiv \bigwedge_{i,j:\pi(\sigma_i^j)=s} \Theta_0^{lru}(i,j) \tag{17}$$

$$\Gamma_{miss}^{lru}(s) \equiv \bigwedge_{i,j:\pi(\sigma_i^j)=s} \Theta_{miss}^{lru}(i,j); \quad \Gamma_{hit}^{lru}(s) \equiv \bigwedge_{i,j:\pi(\sigma_i^j)=s} \Theta_{hit}^{lru}(i,j) \tag{18}$$

Finally. we gather all constraints with respect to cache set s. Our goal is to maximize the delay faced by accessing memory blocks mapped to s. This is performed via the following constraints and objective function.

$$\Gamma(s) \equiv \Gamma_{order}(s) \wedge \Gamma_1^{lru}(s) \wedge \Gamma_0^{lru}(s) \wedge \Gamma_{miss}^{lru}(s) \wedge \Gamma_{hit}^{lru}(s) \tag{19}$$

$$\Delta(s) = maximize \sum_{i,j:\ \pi(\sigma_i^j)=s} \delta_i^j \tag{20}$$

Note that $\Gamma(s)$ includes *all* constraints that could influence $\Delta(s)$. We can use recent development in SMT solving [17] to maximize the objective function captured via Equation (20). It is also worthwhile to mention that the preceding process can be carried out in an exactly same fashion for FIFO policy. As a result, our approximation strategy is generic, with respect to the replacement policy employed in a cache.

For each cache set s, we formulate $\Gamma(s)$ and obtain the value of $\Delta(s)$ using [17]. If $s_1, s_2, \ldots, s_q$ are all different sets in the shared cache, $\sum_{r\in[1,q]} \Delta(s_r)$ over-approximates the total delay in accessing the shared cache. More precisely, we state the crucial property of our approximation scheme as follows (see [3] for the proof).

Property 2 *Let us assume* $\{s_1, s_2, \ldots, s_q\}$ *are different sets in the shared cache. For a given temporal constraint* $\sum_{i,j} \delta_i^j < \tau$, *if our baseline constraint system* Φ (cf. *Constraint (15)) is satisfiable, then* $\sum_{r\in[1,q]} \Delta(s_r) \geq \tau$. *In other words, our approximation scheme will never miss the violation of any temporal constraint.*

However, it is worthwhile to mention that our approximation scheme may generate *false positives*. In particular, $\sum_{r\in[1,q]} \Delta(s_r)$ might over-approximate the maximum value of $\sum_{i,j} \delta_i^j$. This is due to the reason that interleaving patterns, which lead to the maximum delay for individual cache sets, may not be feasible together. In our evaluation, we empirically evaluate the amount of pessimism in our approximation scheme.

4 Extension

Applications with shared variables Our framework handles interferences in the shared resources, but, not in the shared variables. As a result, we do not catch the scenario when the program control-flow changes due to updates to shared variables. However, many embedded applications are designed by a number of independent components and the communication occurs in terms of reading sensor inputs or writing to output ports. In our evaluation, we show a real-life robot controller which operates via two independent tasks – balance and navigation. Moreover, shared memory-space across cores often bypass caches, to avoid power consumption due to the coherence traffic [14]. If accessing the shared memory-space bypasses cache, our framework can be easily extended for general applications with shared variables. In order to accomplish this, we need to generate additional constraints, which encode the program control-flow observed during a failure run (*i.e.* an execution scenario violating certain temporal constraints). This can be achieved in an exactly same fashion as shown in [15].

It is slightly more involved when accessing the shared memory-space goes through caches. In particular, we need to add constraints that capture cache misses due to data coherency and false sharing. This can be accomplished by correlating writes and reads to the same memory block. Besides, we need to distinguish the *first-ever shared-cache miss* for a memory block via Constraint (21), for any cache replacement policy. Without data sharing, such cache misses can be detected during the inspection of each core in isolation.

$$\bigwedge_{i,j} \left(\bigwedge_{p,q:\ \sigma_p^q = \sigma_i^j} \left(\mathcal{O}_p^q > \mathcal{O}_i^j \right) \Rightarrow \left(\delta_i^j = MISS \right) \right) \tag{21}$$

Constraint (21) encodes the scenario of i^j being the first shared-cache access to request memory block σ_i^j. This, in turn, leads to a shared-cache miss. We are currently extending MESS to handle data sharing and cache coherency.

Performance debugging for a class of inputs With minor changes, our framework can be extended for performance debugging on a class of inputs. The key to such extension is to collect *path conditions* [12], while monitoring the performance of each core *in isolation*. For each core, such a *path condition* captures the set of all inputs which lead to the respective execution scenario. However, depending on the value of input x, the statement $a[x]$ might access different memory blocks, for the same path condition. Therefore, we need to generate constraints for each such memory block, satisfying the respective path constraint. Let us assume that array a might access memory block m_1 if $0 \leq x \leq 2$ and it accesses memory block m_2 if $2 < x \leq 5$. Subsequently, to formulate cache conflicts generated by accesses (*i.e.* Constraints (5)-(6) for LRU policy and Constraints (11)-(12) for FIFO policy) to m_1 and m_2, we additionally constrain via conditions $(0 \leq x \leq 2)$ and $(2 < x \leq 5)$, respectively. For instance, we modify $\Theta_1^{lru}(i,j)$ to $\Theta_1^{lru}(i,j) \wedge (0 \leq x \leq 2)$ for memory block m_1 and to $\Theta_1^{lru}(i,j) \wedge (2 < x \leq 5)$ for memory block m_2. In future, we aim to build such extension to instantiate performance debugging on a set of inputs, which are captured symbolically by path conditions.

5 Evaluation

We have implemented MESS using simplescalar [6] and Z3 constraint solver [5]. In our evaluation, we configure a multi-core system with dual-core processor, where each core has a private level-one cache and all the cores share a level-two cache. This is a typical design in many embedded systems, such as devices using Exynos 5250 [4], which, in turn, contains a dual-core, ARM Cortex-A15 [1] chip. We configure 1 KB level-one caches with associativity 2 and 2 KB level-two cache with associativity 4. All caches have a line size of 32 bytes. Cache sizes are chosen in a fashion such that we obtain enough accesses to the shared cache and therefore, generate a reasonable number of constraints in our framework (see [3] for experiments with different cache configurations). To evaluate our framework, we have chosen medium to large size programs from [13], which are generally used to validate timing analyzers. We have also used a robot controller from [2]. which contains two tasks — `balance` (to help the robot to keep it in upright position) and `navigation` (to drive the robot through rough terrain). These two tasks are assigned to different cores in our configured dual-core system.

Experimental setup For our evaluation with programs from [13], we run `jfdctint` on one core and choose different programs to run on the other core. We use such a setup in order to check the influence of the same inter-core cache conflicts on different programs. For the robot controller, we run `balance` and `navigation` on two different cores. The first two columns in Table 1 list the set of programs and the respective size of source code. We monitor the execution on each core by instrumenting memory accesses in Simplescalar. At the end of the execution, we generate a summary of memory performance for each core, which, in turn are used to generate constraints. The generated constraints are solved via Z3. All evaluations have been performed on an Intel I7 machine, having 8 GB of RAM and running ubuntu 14.04 operating systems.

Table 1. Evaluation of our baseline framework: "lines of C code" considers the sum of source code of two programs running on two different cores, "#violations" captures the number of violations within the set of 30 temporal constraints $\{\sum_{i,j} \delta_i^j < 200, \ldots, \sum_{i,j} \delta_i^j < 3100\}$.

Program	Total lines of C code	Shared-cache repl. policy	#shared-cache access	Size of constraints	#violations	Time to generate constraints (secs)	Solver time (secs) Max. / Geo. Mean
`cnt`	642	LRU	432	2111	22	1.17	25.01 / 1.84
`+jfdctint`		FIFO	432	6586	22	9.52	161.83 / 15.73
`expint`	532	LRU	433	2166	23	1.22	10.84 / 2.16
`+jfdctint`		FIFO	433	6643	23	9.62	576.56 / 20.02
`qurt`	541	LRU	448	2817	30	1.88	24.81 / 3.16
`+jfdctint`		FIFO	448	7272	30	9.38	31.77 / 11.59
`matmult`	538	LRU	436	2283	28	1.31	244.39 / 1.91
`+jfdctint`		FIFO	436	6758	28	9.69	15495.83 / 12.82
`fdct`	614	LRU	479	3943	30	2.99	17.49 / 5.01
`+jfdctint`		FIFO	479	8418	30	11.85	44.31 / 21.44
`nsichneu`	4628	LRU	1679	40087	30	49.2	17120.46 / 7904.08
`+jfdctint`		FIFO	1679	44562	30	15.35	27534.20 / 15174.8
`balance`	2098	LRU	772	3881	30	0.23	155.17 / 63.94
`+navigation`		FIFO	773	6770	30	0.56	389.68 / 184.32

Basic results Table 1 outlines the basic evaluation of our framework. We set the shared-cache miss-penalty (hit-latency) to be 100 (1) cycles. Recall that we aim to check the validity of temporal constraints $\sum_{i,j} \delta_i^j < \tau$. We generate a number of temporal constraints by varying τ from 200 to 3100 cycles, at a step of 100 cycles and for each such temporal constraint, we invoke our framework. Note that τ captures all possibilities between two to thirty one shared-cache misses. Besides, in $\sum_{i,j} \delta_i^j$, we only consider shared-cache accesses i^j, whose latency were unknown during the investigation of each core in isolation (*cf.* Column 4 in Table 1). Therefore, any shared-cache access i^j, which incurs the first-ever cache miss of the respective memory block σ_i^j, is not included in $\sum_{i,j} \delta_i^j$. In Table 1, we report the maximum and geometric mean over the time to check all temporal constraints. For several cases, this maximum time was recorded for a *valid* temporal constraint, meaning that the solver failed to find a violation. We can observe that, for many scenarios, the solver returns a solution in reasonable time. However, with large number of constraints, the solver takes long time to find a solution. For instance, with program `nsichneu`, such a scenario happens due to its large size and a substantial number of accesses to the shared-cache. In general, finding a solution for FIFO policy takes longer time compared to LRU policy, due to a larger constraint-size.

Evaluation of the approximate solution Table 2 compares our approximation and the baseline framework. As clearly observed, our approximation dramatically reduces the

Table 2. Efficiency and precision of our approximation. TO denotes timeout (>5 hours). "Max. #constraints" capture the maximum number of constraints solved by Z3 over all invocations.

Program	Replacement policy of the shared-cache	Max. #constraints		Solver time (in seconds)		Max. delay $\left(\max \sum_{i,j} \delta_i^j\right)$ (in CPU cycles)	
		baseline	approx	baseline	approx	baseline	approx
`cnt+jfdctint`	LRU	2111	154	23.58	4.39	2394	3285
	FIFO	6586	513	116.49	14.35	2300≤X<2400	3285
`expint+jfdctint`	LRU	2166	207	10.84	4.77	2494	3385
	FIFO	6643	526	409.39	14.58	2400≤X<2500	3385
`qurt+jfdctint`	LRU	2817	305	565.91	9.2	3884	6161
	FIFO	7272	631	TO	29.03	≥3900	6061
`matmult+jfdctint`	LRU	2283	154	244.39	5.23	2988	4473
	FIFO	6758	513	15495.83	15.98	2900≤X<3000	4473
`fdct+jfdctint`	LRU	3943	304	TO	22.31	≥6200	10116
	FIFO	8418	599	TO	66.4	≥6200	10116
`nsichneu+jfdctint`	LRU	40087	2862	TO	764.56	≥10000	31500
	FIFO	44562	3137	TO	926.45	≥10000	31500
`balance+navigation`	LRU	3881	442	93.32	12.81	12800≤X<12900	13200
	FIFO	6770	818	182.68	25.08	12200	12200

debugging time, compared to the baseline framework. This is due to the partitioning of constraints with respect to different cache sets. Such constraint partitioning drastically reduces the number of constraints to be solved together, leading to a substantial reduction of pressure to Z3. As our approximation may generate false positives, we also compare the precision of our approximation compared to the baseline framework. In order to do this, we compare the maximum delay computed by our approximation with the maximum delay computed by the baseline framework. This maximum delay captures the sum of all delays to access the shared-cache. For our baseline framework, obtaining such maximum delay may incur large overhead (we used `symba` [17] to compute the maximum delay). In such cases, we use the time taken by the solver to validate a temporal constraint $\sum_{i,j} \delta_i^j < \tau$. This means that the maximum delay cannot exceed $\tau - 1$. For instance, in Table 2, 2300≤X<2400 indicates that the solver found a solution for $\sum_{i,j} \delta_i^j \geq 2300$, but not for $\sum_{i,j} \delta_i^j \geq 2400$. The respective debugging-time captures the time taken by the solver for $\sum_{i,j} \delta_i^j \geq 2400$. Finally, we use a timeout of five hours for the solver. For instance, the timeout event happens for the program `fdct`. From Table 2, we also observe that the precision of our approximation scheme is reasonable, in the context of validating embedded software. Finally, we note that with the current state-of-the-art solutions (*e.g.* using [17]), discovering the exact worst-case ordering among memory accesses (in terms of performance), is not very efficient.

Notes on scalability We have implemented a *proof-of-concept* of MESS. We have also shown an approximation, which dramatically improves the solver performance, with a reasonable loss of precision. We believe that several optimizations are still possible. In particular, as shown in [15], other optimizations for parallel constraint-solving is feasible. We are exploring such techniques to further improve the efficiency of MESS.

6 Related work

Testing and debugging of multi-threaded applications has been an active topic of research for the last few years [20,15,23,19,24]. Unlike these approaches, our work concentrates on resource sharing in parallel architectures, rather than data sharing in parallel applications. However, to consider shared data in our framework, an approach similar to [15] can be integrated easily into our constraint system. Our work is also orthogonal to efforts in program synthesis, such as the approach taken in [8]. Instead of generating correct and optimal programs from their specification [8], we aim to discover performance bugs in the original implementation of software.

Modeling shared-cache performance has been an active topic of research in the past few years [21,22]. Unlike our approach, these works do not provide strong guarantees on the presence or absence of performance bugs due to shared caches. Such guarantees are crucial for time-critical code fragments. Recent works on performance testing aim to generate performance-stressing execution in sequential [7] or parallel applications [16,10]. These works are not directly applicable to reproduce or debug performance bugs. Besides, in this paper, we provide strong guarantees on the absence of performance bugs, when a given temporal constraint is not invalidated by the solver.

Works on worst case execution time (WCET) analysis, for multi-core systems [9], predicts the maximum execution time of an application over all possible inputs and interleavings. In this paper, our goal is orthogonal and we aim to discover, for a given input, the interleaving pattern that causes the violation of temporal constraints. Therefore, our work has a significant testing and debugging flavour compared to WCET analysis.

In summary, previous works on automated debugging have mostly concentrated on functionality bugs or performance bugs on single-core systems. In this paper, we propose a systematic debugging approach that highlights performance bugs on multi-core systems, with a specific focus on shared caches.

7 Conclusion

In this paper, we have proposed MESS, a constraint-based framework to debug memory performance in multi-core systems. MESS systematically finds the interleaving pattern that causes the violation of temporal constraints. An appealing feature of our framework is its ability to provide guarantees on the absence of performance bugs, such as the validity of temporal constraints, for a given input. We have also integrated an approximation scheme, which, with a reasonable loss of precision, improves the debugging time by several magnitudes. In general, this opens up several opportunities to improve the debugging time enforced by MESS. Our evaluation with several embedded software and also with a real-life robot controller shows the effectiveness of our approach. Finally, since the performance of constraint solvers is continuously improving, we believe that MESS proposes a promising approach for performance debugging on multi-core systems. In future, we aim to build on our approach to consider shared data and other crucial shared resources in multi-core systems, such as shared buses. We also aim to use MESS to automatically synthesize fixes of performance bugs. One possible approach would be to synthesize barriers. The primary purpose of such barriers will be to satisfy a given temporal constraint, via restricting certain interleaving patterns.

Acknowledgement We thank the anonymous reviewers for their insightful comments and feedback. This work is partially supported by the Swedish National Graduate School on Computer Science (CUGS). This support is gratefully acknowledged.

References

1. ARM Cortex-A5 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a5.php.
2. Ballybot balancing robots. http://robotics.ee.uwa.edu.au/eyebot/doc/robots/ballybot.html.
3. MESS: Memory Performance Checker for Embedded Multi-core Systems. http://sudiptac.bitbucket.org/papers/mess-extended.pdf.
4. Samsung Exynos Processor. http://www.samsung.com/global/business/semiconductor/file/product/Exynos_5_Dual_User_Manaul_Public_REV100-0.pdf.
5. Z3 Constraint Solver. http://z3.codeplex.com/.
6. T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2), 2002.
7. A. Banerjee, S. Chattopadhyay, and A. Roychoudhury. Static analysis driven cache performance testing. In *RTSS*, 2013.
8. P. Cerný, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh. Quantitative synthesis for concurrent programs. In *CAV*, 2011.
9. S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A unified WCET analysis framework for multi-core platforms. *TECS*, 13(4s), 2014.
10. S. Chattopadhyay, P. Eles, and Z. Peng. Automated software testing of memory performance in embedded gpus. In *EMSOFT*, 2014.
11. J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.
12. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.
13. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen WCET benchmarks: Past, present and future. In *WCET*, 2010.
14. B. Holton, K. Bai, A. Shrivastava, and H. Ramaprasad. Construction of GCCFG for interprocedural optimizations in software managed manycore (SMM) architectures. In *CASES*, 2014.
15. J. Huang, C. Zhang, and J. Dolby. CLAP: recording local executions to reproduce concurrency failures. In *PLDI*, 2013.
16. G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan. GKLEE: Concolic verification and test generation for GPUs. In *PPoPP*, 2012. http://www.cs.utah.edu/formal_verification/GKLEE/.
17. Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik. Symbolic optimization with SMT solvers. In *POPL*, 2014.
18. Y. Liang and T. Mitra. Instruction cache locking using temporal reuse profile. In *DAC*, 2010.
19. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.
20. S. Nagarakatte, S. Burckhardt, M. M. K. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI*, 2012.
21. A. Sandberg, D. Black-Schaffer, and E. Hagersten. Efficient techniques for predicting cache sharing and throughput. In *PACT*, 2012.
22. A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *HPCA*, 2013.
23. K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.
24. D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *ASPLOS*, 2010.