Noname manuscript No.
(will be inserted by the editor)

Scalable and Precise Refinement of Cache Timing Analysis via Path-sensitive Verification

Sudipta Chattopadhyay · Abhik Roychoudhury

Received: date / Accepted: date

Abstract Hard real-time systems require absolute guarantees in their execution times. Worst case execution time (WCET) of a program has therefore become an important problem to address. However, performance enhancing features of a processor (*e.g.* cache) make WCET analysis a difficult problem. In this paper, we propose a novel analysis framework by combining abstract interpretation and program verification for different varieties of cache analysis ranging from single to multi-core platforms. Our framework can be instantiated with different program verification techniques, such as model checking and symbolic execution. Our modeling is used to develop a precise yet scalable timing analysis method on top of the Chronos WCET analysis tool. Experimental results demonstrate that we can obtain significant improvement in precision with reasonable analysis time overhead.

1 Introduction

Worst-case execution time (WCET) analysis of real-time embedded software is an important problem. WCET estimates of tasks are used for system level schedulability analysis. WCET estimation usually involves a program level path analysis (to determine the infeasible paths in the program's control flow graph), micro-architectural modeling (to accurately determine the maximum execution time of the basic blocks), and a calculation phase (which combines the results of path analysis and micro-architectural modeling).

Sudipta Chattopadhyay
School of Computing, National University of Singapore, Singapore - 117417
Tel.: +65 6516 6836
Fax: +65 6779 4580
E-mail: sudiptac@comp.nus.edu.sg

Abhik Roychoudhury
School of Computing, National University of Singapore, Singapore - 117417
E-mail: abhik@comp.nus.edu.sg

Micro-architectural modeling usually involves systematically considering the timing effects of performance enhancing processor features such as pipeline and caches. Cache analysis for real-time systems is usually accomplished by abstract interpretation. This involves estimating the cache behavior of a basic block B by considering the incoming flows to B in the control flow graph. The memory accesses of the incoming flows are analyzed to determine the cache hits/misses for the memory accesses in B. Since programs contain loops, such an analysis of memory accesses involves an iterative fixed point computation via a method known as abstract interpretation. Abstract interpretation is usually efficient, but the results are often not precise. This is because the estimation of memory access behaviors are "joined" at the control flow merge points - resulting in an over-estimation of potential cache misses returned by the method.

In this paper, we develop a cache analysis framework which improves the precision of abstract interpretation, without appreciable loss of efficiency. We augment abstract interpretation with a gradual and controlled use of path sensitive program verification methods (*e.g.* model checking and symbolic execution). Because of path sensitivity in the search process, program verification methods are known to be of high complexity. Hence abstract interpretation based analysis cannot be naively replaced with standard program verification methods such as model checking or symbolic execution. Recent works [Lv et al, 2010] which have advocated combination of abstract interpretation and model checking for multicore software analysis - restrict the use of model checking to program path level; cache analysis is still accomplished only by abstract interpretation. Indeed almost all current state-of-the-art WCET analyzers (such as Chronos [Li et al, 2007], aiT [aiT, 2000]) perform cache analysis via some variant of abstract interpretation. Model checking is usually found to be not scalable for micro-architectural analysis because of the huge search space that needs to be traversed [Wilhelm, 2004; Huber and Schoeberl, 2009]. The main novelty of our work lies in integrating path sensitive program verification methods with abstract interpretation for timing analysis of cache behavior.

Our baseline analysis is abstract interpretation. Potential cache conflicts identified by abstract interpretation are then subjected to a path sensitive program verification method. Our goal is to rule out "false" cache conflicts which can occur only on infeasible program paths. Such false conflicts are reported by abstract interpretation since its join operator (which merges the estimates from paths at control flow join points) conservatively considers all possible cache conflicts on any path in the control flow graph. The path sensitive search in program verification naturally rules out the infeasible program paths and the cache conflicts incurred therein.

One appealing nature of our analysis method is that the results are always safe. We start with the results from abstract interpretation and gradually refine the results with repeated runs of program verification. We instantiate our framework with two different program verification methods - model checking and symbolic execution.

Model checking is a property verification method which takes in a system/program P and a temporal logic property φ, where φ is interpreted over the execution traces [1] of P. It checks whether all execution traces of P satisfy φ. Given a potentially con-

[1] We consider only Linear Time Temporal Logic properties here.

flicting pair of memory blocks, we can model check a property that the pair never conflicts in any execution trace of the program. If indeed the conflict pair is introduced due to the over-approximation in abstract interpretation - model checking verifies that the conflict pair can never be realized. We can then rule out the cache misses estimated due to the conflict pair and tighten the estimated time bounds.

Symbolic execution refers to executing a program with symbolic or un-instantiated inputs - as opposed to concrete inputs. Symbolic execution may be static (by which we mean execution of all possible paths in a program) or dynamic (by which we mean execution of a specific program path). In this paper, we use static symbolic execution as embodied in the KLEE toolkit [KLEE, 2008].

Most often, a symbolic execution engine relies on the power of constraint solving. Constraint solving technology has made a significant progress with the advances in *satisfiability modulo theory* (SMT). In symbolic execution, a program is executed with *symbolic* input values (rather than concrete input values in normal execution). Since the input values are *symbolic*, a branch instruction in the program may lead to multiple execution scenarios, as both the *true* and *false* legs of the branch might be satisfiable. Such multiple execution scenarios are reasoned independently by the symbolic execution engine. The feasibility of a path at a branch instruction is checked *on-the-fly* during the execution by sending a query to the SMT-based constraint solver. Given a formula φ to check at a particular program location, the constraint solver is also used to check the satisfiability of φ whenever the same program location is visited by any execution scenario during the symbolic execution.

Due to the inherent path sensitive nature of symbolic execution, the spurious cache conflicts can be eliminated if they are introduced due to the over-approximation of abstract interpretation. As the SMT technology is continuously evolving, we believe that the composition of abstract interpretation and symbolic execution leads to an exciting opportunity for WCET analysis.

Recall that abstract interpretation merges the results from different paths, via the join function. Thus, abstract interpretation is not necessarily path-sensitive. On the other hand, the property checked in a single run of program verification (via model checking or symbolic execution) involves certain cache conflicts identified by abstract interpretation. The path sensitive search by program verification then detects whether these conflicts are indeed realizable. Overall, the scalability of our framework is never in question. Given a time budget T, we can first employ abstract interpretation and then employ as many runs of program verification as we can within time T. Of course, given more time, more precise analysis results (in the form of potential cache misses) are achieved.

Contributions In summary, this paper presents a generic cache analysis framework based on abstract interpretation and path sensitive program verification methods. In our previous work [Chattopadhyay and Roychoudhury, 2011], we have designed and evaluated a generic cache analysis framework based on abstract interpretation (AI) and model checking. In this paper, we additionally show that an AI-based cache analysis framework can be extended in a scalable fashion by a systematic use of symbolic execution - a path sensitive program verification method. As a result, our work in this paper shows that we can instantiate our analysis framework with different path

sensitive program verification techniques, such as model checking and symbolic execution. Moreover, we provide detailed experimental results to evaluate the impact of our approach on different model checking engines - namely bounded model checking and explicit-state search based model checking. Depending on the time budget for analysis and the analysis precision required - the framework can be tuned to analyze cache hit/miss classifications for timing analysis. We further show that the framework can be instantiated with a wide variety of cache analyses - (i) analysis of cache behavior in a single program, (ii) analysis of cache related preemption delay for a multi-tasking system where the tasks are running on a single core, and (iii) analysis of shared caches in multi-cores. Our experimental results on the subject programs chosen from [Gustafsson et al, 2010] show substantial improvements in the precision of timing analysis results with limited time overheads. This yields a parameterizable cache analysis framework for real-time systems which is generic, precise and scalable.

Organization The rest of the paper is organized as follows. The next section discusses the related work. Section 3 introduces a general background on abstract interpretation based cache analyses. Section 4 presents a general description of our compositional analysis framework using abstract interpretation and path sensitive program verification methods. Section 5 describes the instantiation of our analysis framework using model checking, similar to our previous work in [Chattopadhyay and Roychoudhury, 2011]. In Section 6, we discuss the instantiation of our framework using symbolic execution - the key contribution of this paper. Section 7 discusses the implementation details and all the experimental results. We present a few important extensions of our framework in Section 8 and Section 9 concludes the paper.

2 Related work

Since the initiation of WCET analysis research, cache modeling has been an active topic in this area. Initial works used Integer Linear Programming (ILP) [Li et al, 1999] for modeling intra-task cache conflicts. However, ILP-based approach for cache modeling faces scalability concerns in terms of analysis time. Subsequently, a novel WCET analysis approach has been proposed in [Theiling et al, 2000], which efficiently composes abstract interpretation based micro-architectural modeling and ILP-based path analysis. The solution proposed in [Theiling et al, 2000] has been proved scalable and it has also been adopted in commercial tool chain [aiT, 2000].

In multi-tasking system, additional difficulties arise in modeling inter-task cache conflicts. Inter-task cache conflicts are generated by a high priority task when it preempts a low priority task. The bound on additional cache misses due to preemption is called *cache related preemption delay* (CRPD). In last decade, there has been an extensive amount of research to estimate CRPD [Lee et al, 1998; Negi et al, 2003; Tan and Mooney, 2007] using abstract interpretation. Recently, two advancements in CRPD estimation ([Altmeyer and Burguière, 2009] and [Altmeyer et al, 2010]) have improved and generalized the previously proposed approaches for set-associative caches.

With the extensive deployment of multi-core architectures, it has also become important to adopt the existing cache analysis techniques for multi-cores. Multi core architectures employ shared resources (*e.g.* shared cache). Therefore, a few research groups have already proposed the modeling of shared caches ([Li et al, 2009], [Yan and Zhang, 2008] and [Hardy et al, 2009]) based on abstract interpretation.

Path sensitive program verification techniques, such as model checking has previously been explored by the research community in the context of WCET estimation. The work in [Metzner, 2004] uses model checking alone for cache and path analysis. However, [Metzner, 2004] does not employ the modeling of other important micro-architectural features (*e.g.* pipeline) and it is unclear whether the employed technique would remain scalable in the presence of pipeline or other micro-architectural features. In contrast, our technique can easily be integrated with the modeling of different micro-architectural features (*e.g.* pipeline). Nevertheless, some recent advances [Dalsgaard et al, 2010] have employed full model checking based approach for software timing analysis in pipelined processor. However, [Dalsgaard et al, 2010] faces some common scalability issues (*e.g.* state space explosion) in the presence of caches. In [Wilhelm, 2004] and [Huber and Schoeberl, 2009], it has also been argued that model checking alone is not suitable for WCET analysis due to the state space explosion problem.

In summary, abstract interpretation based approach is scalable for cache analysis and it is easy to integrate with other micro-architectural features (*e.g.* pipeline). On the other hand, path sensitive program verification such as model checking can give the most accurate result, but it is difficult to scale in terms of analysis time. A recent approach [Lv et al, 2010] has therefore looked at the combination of abstract interpretation and model checking. However, [Lv et al, 2010] uses model checking for path analysis only; cache analysis is performed by conventional abstract interpretation methods.

In the past few years, symbolic execution has widely been adopted for the functionality testing of software [Godefroid et al, 2005; Cadar et al, 2008]. Such research progress in the functionality testing has been made possible due to the recent advances in *satisfiability modulo theory* (SMT). SMT-based constraint solvers have been adopted to explore feasible program paths. However, previous works [Godefroid et al, 2005; Cadar et al, 2008] have studied symbolic execution mainly in the context of functionality testing. On the other hand, our work studies the combination of abstract interpretation and symbolic execution to improve the precision of cache timing analysis in a scalable fashion.

To summarize, we study the combination of abstract interpretation and path sensitive program verification for different cache analysis to design a scalable and precise WCET analysis framework. Our proposed framework tries to best combine the *scalability* advantage offered by abstract interpretation with the *accuracy* offered by path sensitive program verification techniques (*e.g.* model checking, symbolic execution). Our analysis can be stopped after anytime during the program verification phase and the results are always safe. Thus our framework gives the designer a precision-scalability tradeoff which (s)he can choose to use.

3 Background

WCET analysis of a single task WCET analysis of a single task is broadly composed of two different phases: i) micro-architectural modeling and ii) path analysis. Micro-architectural modeling analyzes the timing characteristics of different hardware components (*e.g.* cache, pipeline, branch predictor) and works at the granularity of basic blocks. As an outcome of micro-architectural modeling, we obtain the WCET of each basic block in the examined program. On the other hand, path analysis uses the WCET of each basic block as input and searches for the *longest feasible program path*. Our baseline implementation employs the separated cache and path analysis as proposed in [Theiling et al, 2000]. [Theiling et al, 2000] uses abstract interpretation (AI) for cache analysis and integer linear programming (ILP) for path analysis. We assume *least recently used* (LRU) cache replacement policy. Memory blocks are categorized as *all-hit* (AH) or *all-miss* (AM) or *unclassified* (NC). Cache analysis is used along with the virtual inline and virtual unrolling (VIVU), as discussed in [Theiling et al, 2000]. In VIVU approach, each loop is unrolled once to distinguish the *cold cache misses* at first iteration of the loop. AH categorized memory blocks are always in cache when accessed. On the other hand, AM categorized memory blocks are never in cache when accessed. If a memory block cannot be classified as either of two (AH or AM), it is considered *unclassified* (NC). Cache analysis outcome is used for computing the WCET of each basic block. Finally, longest path search in a program is formulated as an integer linear program. The formulated ILP uses the basic block WCETs and structural constraints imposed by program control flow graph (CFG). Infeasible program path informations are also encoded as separate ILP constraints using the technique explored in [Ju et al, 2008]. The solution of the formulated ILP returns the whole program WCET.

Inter-task cache conflict analysis Inter-task cache conflict analysis is required to find an upper bound on cache misses due to preemption. The bound on cache misses (or additional clock cycles) due to preemption is called cache related preemption delay (CRPD). CRPD analysis revolves around the notion of two basic concepts: *useful cache blocks* (UCB) and *evicting cache blocks* (ECB). UCBs are computed by analyzing the preempted task and ECBs are computed by analyzing the preempting task. A UCB is a block that may be *cached* before preemption and may be *used* later, resulting in a cache hit in the absence of preemption. The number of UCBs imposes a bound on CRPD. On the other hand, the preempting task can cause additional cache misses in a cache set only if it uses the same cache set during its execution. For a particular cache set, the set of cache blocks used by the preempting task during its execution is known as ECB for the corresponding cache set. ECBs are used to check whether a particular UCB could be *evicted* by the preempting task. We can disregard all UCBs from CRPD computation if they could not be evicted by the set of computed ECBs. Therefore, a combined analysis with UCB and ECB may tighten the CRPD estimates obtained with UCB alone [Negi et al, 2003]. Recently, two approaches ([Altmeyer and Burguière, 2009] and [Altmeyer et al, 2010]) have improved and generalized the state-of-the-art CRPD estimation framework ([Lee et al, 1998; Negi et al, 2003]) for set associative caches. We implement both the techniques proposed

in [Altmeyer and Burguière, 2009] and [Altmeyer et al, 2010] in our baseline CRPD estimation framework. Therefore, our baseline implementation captures the current state-of-the-art AI-based CRPD analysis.

Inter-core cache conflict analysis Inter-core cache conflict analysis computes the conflicts generated in shared cache. Conflicts in shared cache, on the other hand, are generated by the tasks running on different cores. Till now, only a few solutions have been proposed for analyzing timing behaviors of shared cache [Li et al, 2009; Hardy et al, 2009; Yan and Zhang, 2008]. However, all of them suffer from over-estimating the inter-core cache conflicts. We use our former work on shared cache analysis [Li et al, 2009], which employs a separate shared cache conflict analysis phase. Shared cache conflict analysis may change the categorization of a memory block m from all-hit (AH) to unclassified (NC). This analysis phase first computes the number of unique conflicting shared cache accesses from different cores. Then it is checked whether the number of conflicts from different cores can potentially replace m from shared cache. More precisely, cache hit/miss categorization (CHMC) of m is changed from all-hit (AH) to unclassified (NC) if and only if the following condition holds:

$$N - age(m) < |\mathcal{M}_c(m)| \tag{1}$$

where $|\mathcal{M}_c(m)|$ represents the number of conflicting memory blocks from different cores which may potentially access the same L2 cache set as m. N represents the associativity of shared L2 cache and $age(m)$ represents the *age* of memory block m in shared L2 cache set in the absence of inter-core conflicts. Therefore, $N - age(m)$ specifically captures the amount of shift that memory block m can tolerate before being replaced from the cache. We call the term $N - age(m)$ as *residual age* of m.

It is worthwhile to mention that our proposed framework only refines the number of estimated shared cache misses. Nevertheless, accessing a shared cache in multi-core system is usually accomplished via a shared communication medium (*e.g.* a shared bus). Previous works [Pellizzoni et al, 2010; Chattopadhyay et al, 2010; Kelter et al, 2011] have shown the technical challenges in modeling the worst-case delay of such shared communication medium. On the contrary, our work in this paper aims to refine the estimation of *worst-case* cache misses and we do not claim any contribution in reducing the worst-case estimation of communication delay (*e.g.* delay arising due to the presence of a shared bus). However, in our previous works [Chattopadhyay et al, 2010; Kelter et al, 2011; Chattopadhyay et al, 2012], we have shown the modeling of both shared caches and shared buses. For shared caches, our previous works have used the AI-based shared cache analysis [Li et al, 2009]. Since our proposed framework in this paper gives a more precise outcome of AI-based shared cache analysis, it can be integrated with our previous works [Chattopadhyay et al, 2010; Kelter et al, 2011; Chattopadhyay et al, 2012] to account the delay arising due to a shared communication medium.

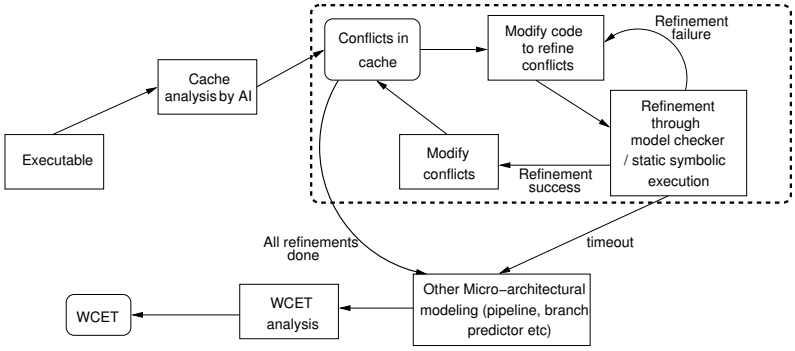


Fig. 1 General framework of our WCET analysis which combines abstract interpretation and model checking / static symbolic execution

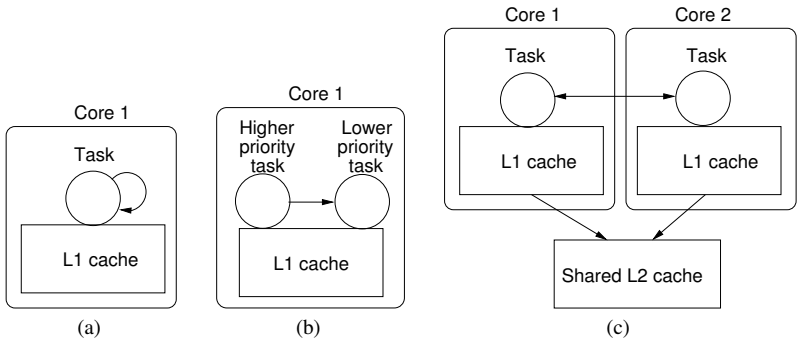


Fig. 2 (a) Intra-task cache conflicts, (b) inter-task cache conflicts, (c) inter-core cache conflicts

4 Overview of the analysis framework

4.1 General framework

Figure 1 demonstrates the general analysis framework. Our goal is to refine different types of abstract interpretation (AI) based cache analysis through program verification techniques. For program verification, we use either static symbolic execution (SE) or model checking (MC). *Cold cache misses* are unavoidable and AI-based cache analysis can accurately predict the set of cold cache misses. However, AI-based cache analysis suffers from overestimating the *conflict misses* in a cache. On the other hand, conflicts in a particular cache set may come from different sources. We focus on all three types of conflicts which may arise in a cache: first, intra-task cache conflicts which is created by different memory blocks accessed by a particular task and mapping into the same cache set. Secondly, inter-task cache conflicts which is created when a high priority task preempts a low priority task. Finally, inter-core cache conflicts which is generated in the shared cache by a task running on a different core. Figure 2 pictorially represents all forms of the above mentioned cache conflicts.

Even though the basic goal of our framework is cache conflict refinement, the notion of cache conflict may vary depending on the outcome of AI-based cache analysis. For example, in inter-task cache conflict refinement, initial CRPD analysis produces a set of ECBs, which can be considered as the set of cache conflicts. On the other hand, during intra-task and inter-core cache conflict refinement, we get the cache hit miss classification (AH, AM or NC) of each memory block. A memory block might be categorized as NC due to its conflicts with more than one memory blocks. Therefore, by refining one NC categorized memory block into AH, we may reduce more than one cache conflict pairs, which may in turn result in an improvement of WCET.

In Figure 1, the dotted boxed portion has different implementations for refining different types of cache conflicts (*i.e.* intra-task, inter-task and inter-core). The refinement of cache conflicts is accomplished iteratively via program verification techniques (symbolic execution or model checking) on a modified code. We rule out the cache accesses for which AI has generated precise information. Therefore, the refinement phase via program verification works on a very small subset of all cache accesses. The iterative refinement through model checking or symbolic execution eliminates several infeasible paths from the candidate program, resulting in the removal of several unnecessary conflicts generated in a particular cache set. The iterative refinement is continued as long as the time budget permits or all possible refinements have been verified.

The iterative refinement via program verification only generates a more precise outcome of the baseline analysis (*i.e.* abstract interpretation). Therefore, our proposed framework does not reduce any of the advantages obtained by using abstract interpretation (AI). AI-based analysis has been widely adopted, as such an analysis can easily be integrated inside a compiler and it can also handle non-compositional architectures (*e.g.* in aiT [aiT, 2000] toolchain). Moreover, AI-based cache analysis can easily be integrated with architectures that exhibit *timing-anomalies*. Our previous works have shown such integration for single-core WCET analysis in [Li et al, 2004] and recently, for multi-core WCET analysis in [Chattopadhyay et al, 2012]. Our proposed framework does not compromise any such advantages offered by AI-based analysis, rather our proposed framework aims to obtain a more precise cache analysis outcome to improve the overall accuracy of WCET estimation.

Recall that the WCET analysis process can broadly be categorized into two phases: micro-architectural modeling and path analysis. The infeasible path exploration by the program verification is only performed for refining cache conflicts (*i.e.* during the micro-architectural modeling phase). For path analysis, our framework encodes the infeasible path information as separate ILP constraints (for details, refer to [Suhendra et al, 2006; Ju et al, 2008]). Infeasible path constraints are finally used in the global ILP formulation for computing WCET. There are two important advantages of our framework: first, the iterative refinement can be terminated at any point if the time budget exceeds. The resulting *cache conflicts*, after a partial refinement, can *safely* be used for estimating the WCET or CRPD. Secondly, our framework can be integrated with other micro-architectural features (*e.g.* pipeline, branch prediction) and thereby, not affecting the flexibility of AI-based cache analysis.

```
void f(int z) {
        int i = 0;
        while (i < 100) {
                if (z >= 0) {
                    x++;
                } else {
                    x−−;
                }
                if (z == −2) {
                    x−−;
                } else {
                    x++;
                }
                i++;
        }
}
```

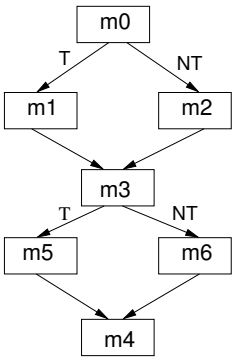


Fig. 3 Example program and its corresponding control flow graph (CFG) without the backedge

4.2 A general code transformation framework

Any code transformation for refining various cache conflicts can be represented by a quintuple $\langle \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} \rangle$ as follows:

- $\mathcal{L}$: Set of conflicting memory blocks in the cache set for which the refinement is being made.
- $\mathcal{A}$: The property which needs be verified. The property is placed in the form of an "assertion" clause, which validates $\mathcal{A}$ for all possible execution traces of the modified code.
- $\mathcal{P}_l$: Set of positions in the code where the conflict count would be incremented. These are the set of positions where some memory block in $\mathcal{L}$ might be accessed.
- $\mathcal{P}_c$: Position in the code where property $\mathcal{A}$ would be placed.
- $\mathcal{I}$: Set of positions in the code to reset conflict count. In case of LRU cache replacement policy, a memory block m becomes the *most recently used* immediately after it is accessed. Therefore, if we are counting cache conflicts with m, the conflict count must be reset after m is accessed.

Any refinement via program verification corresponds to a specific cache set and therefore, conflicts are defined for a specific cache set in each code transformation. Consequently, computation of $\mathcal{L}$ and $\mathcal{P}_l$ depends only on the cache set for which the conflicts are being refined. On the other hand, $\mathcal{A}$, $\mathcal{P}_c$ and $\mathcal{I}$ depends on the type of cache conflict (*i.e.* intra-task, inter-task or inter-core) being refined.

In subsequent sections, we shall describe the instantiation of the framework in Figure 1 for refining different versions of cache conflicts (as shown in Figure 2). We shall also show how $\mathcal{A}$, $\mathcal{P}_c$ and $\mathcal{I}$ are configured depending on the type of cache conflict being refined. We shall primarily discuss the analysis of instruction caches and we shall assume *least-recently-used* (LRU) cache replacement policy. The extension of our framework for non-LRU cache replacement policies and data caches has been discussed in Section 8.

For our subsequent discussions, we shall use the example in Figure 3. Parameter z can be considered as an input to the program. Control flow graph (CFG) of the loop body and the accessed memory blocks are also shown in Figure 3.

5 Instantiation of the general framework via model checking

In the following sections, we shall first give a general background on model checking and subsequently, discuss the instantiation of our general framework (Figure 1) via model checking.

5.1 Recapitulation of Model Checking

Model checking [Clarke et al, 1986] is a state space exploration method for formal verification of program properties. The general formulation of the model checking problem is simple, it checks whether a finite state machine M satisfies a property φ

$$M \models \varphi$$

To explain the use of model checking for program verification we need to explain how we get M, φ and what is means for M to satisfy φ.

The finite state machine M is automatically extracted from the program being verified. Such a finite state machine formally described as a quadruple $\langle S, S_0, \rightarrow, L\rangle$ where S is the set of nodes (also called states) in the finite state machine, $S_0 \subseteq S$ is the set of initial states, $\rightarrow \subseteq S \times S$ is the set of edges (also called transitions) in the finite state machine, and $L : S \rightarrow 2^{AP}$ is a labeling function which maps a given state s to the atomic propositions true in the state s. The atomic propositions true in a given state are drawn from AP, the set of all atomic propositions.

The properties verified are temporal logic properties, which constrain ordering of specific events in program executions. In this work, we are only concerned with Linear-time temporal logic (LTL). The syntax of LTL properties is recursively defined as follows

$$\varphi = true \mid false \mid AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid X\varphi \mid G\varphi \mid F\varphi \mid \varphi U \varphi \mid \varphi R \varphi$$

The formula $true$ is always true and the formula $false$ is never true. Further, the atomic propositions AP form the basic building blocks of the formula. A LTL property is constructed using the following

- Atomic propositions AP
- propositional logic operators
- temporal logic operators X (next), G (globally), F (finally), U (until), R (release).

For this work, the only properties we use are in the form of assertions which should hold in a control location of the program. For example consider the assertion $C\_1 \leq 0$ which should hold in control location $loc2$ in Figure 5. It corresponds to a linear time temporal logic property

$$G(pc == loc2 \Rightarrow C\_1 \leq 0)$$

meaning whenever the program counter variable (denoted pc in the above property) holds the value "$loc2$" (*i.e.*, when control location $loc2$ is reached during program execution), we must have $C\_1 \leq 0$). Given an execution trace π of the program, we can check this property by looking for all the visits to control location $loc2$ in the trace π, and then checking whether for each of these visits $C\_1 \leq 0$ holds true in the corresponding program state.

Finally, we explain what it means for a finite state machine M to satisfy a given LTL property φ. The semantics of LTL dictates that M satisfies φ if and only if *all* the execution traces of M satisfy φ. In the context of our example property $G(pc == loc2 \Rightarrow C\_1 \leq 0)$ — even if one single trace of state machine M is such that it has a visit to control location $loc2$ when $C\_1 \leq 0$ does not hold – we will say that M does not satisfy the property $G(pc == loc2 \Rightarrow C\_1 \leq 0)$. Such an execution trace π will then be considered as a *counter-example* trace of the property.

5.2 Refinement of intra-task cache conflicts

In this section we describe the refinement of cache conflicts shown in Figure 2(a). Recall that the memory blocks are classified as AH (*all-hit*), AM (*all-miss*) or NC (*unclassified*) by [Theiling et al, 2000]. AH and AM are guaranteed categorizations by AI based cache analysis. Therefore, AH and AM categorized memory blocks do not have any scope for refinement. On the other hand, AI based cache analysis fails to give guaranteed information (in this case cache hit or cache miss) for NC categorized memory blocks. Consequently, we use the model checker to refine the set of NC categorized memory blocks.

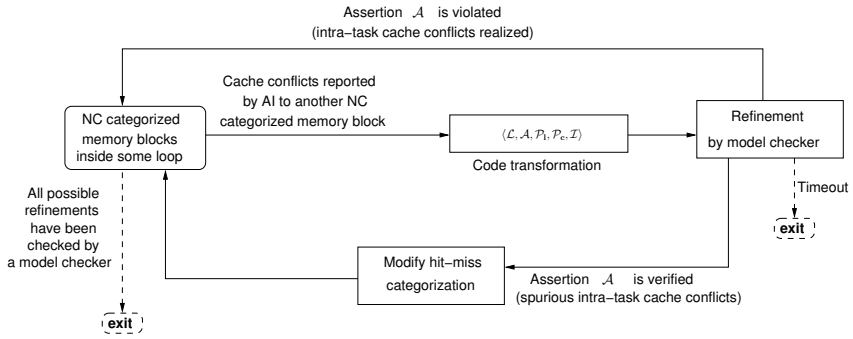


Fig. 4 Refinement of intra-task conflict analysis

Figure 4 demonstrates the instantiation of our general framework for reducing the over-estimation in intra-core WCET analysis. As shown in Figure 4, we only target the NC categorized memory blocks inside some loop. Therefore, we concentrate only on a few memory blocks whose successful refinement may lead to a reasonable WCET improvement. For each of the NC categorized memory blocks under consideration, we call our general code transformation framework (as shown in Figure 4).

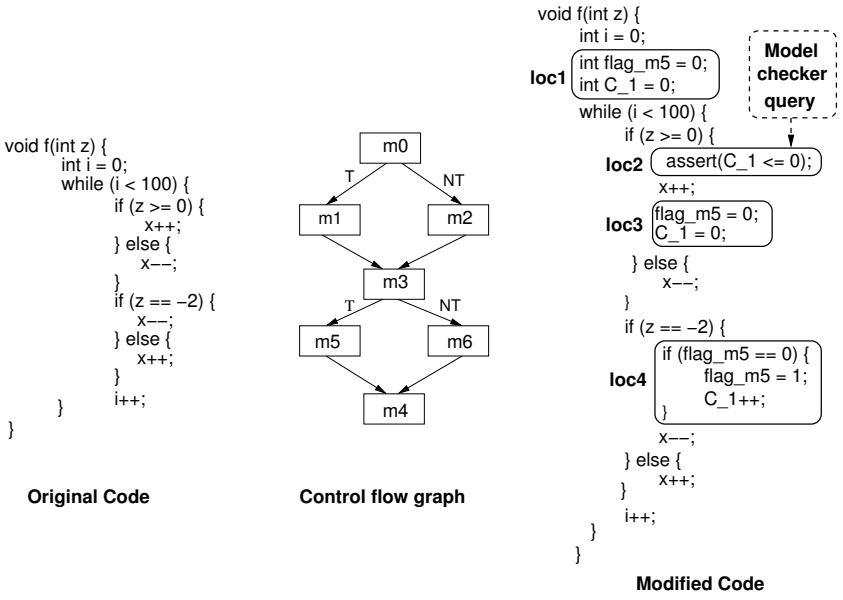


Fig. 5 Intra-task cache conflict refinement

The transformed code has an assertion property $\mathcal{A}$ which needs to be verified by the model checker. A successful verification of the assertion property $\mathcal{A}$ captures the fact that certain cache conflicts reported by AI were spurious and we can conclude that the NC categorized memory block can never be evicted from the cache (as pointed by the "Modify hit-miss categorization" block in Figure 4). On the other hand, the assertion property $\mathcal{A}$ is violated when certain cache conflicts reported by AI were realized and such cache conflicts may lead to the eviction of the NC categorized memory block. This process is continued until we have checked all the NC categorized memory blocks inside loops or the time budget exceeds.

Let us assume that we want to refine the categorization of a particular memory reference m mapping to a cache set i. m becomes the *most recently used* cache block immediately after it is accessed. We would like to check whether m could be evicted from the cache between any two of its consecutive references. Let us assume C_i counts the number of unique conflicts in cache set i and N is the associativity of the cache. Since we use LRU cache replacement policy, it would require at least N unique conflicts to replace m (from cache set i) after m is referenced. Therefore, if C_i is less than or equal to $N-1$, we can guarantee that m cannot be *evicted* from the cache. The model checker is used to check an "assertion" property $C_i \leq N-1$ just before m is referenced. More over, as m is the most recently used cache block immediately after it is accessed, we need to reset the conflict count C_i after m is referenced.

We demonstrate our technique through an example in Figure 5. Parameter z can be considered as a user input. Corresponding control flow graph (CFG) of the loop body and the accessed memory blocks are shown in Figure 5. For illustration pur-

poses, assume a direct-mapped L1 cache where $m1$ and $m5$ are mapped to the same cache set and rest of the memory blocks do not conflict in L1 cache with $m1$ or $m5$. A correct AI-based cache analysis will classify both $m1$ and $m5$ accesses as NC. Figure 5 shows the transformation to refine the NC categorization of $m1$. Since the cache is direct mapped, the refinement of $m1$ is possible only if there are no other conflicting cache accesses between any two consecutive accesses of $m1$. Variable C_1 serves the purpose of counting the number of conflicts. Since $m5$ is the only conflicting memory block, C_1 is incremented before the access of $m5$. Increment of C_1 is guarded by condition ($flag\_m5$ serves the purpose of guard), so that we count only unique cache conflicts. The above transformation of code is *fully automated* and we pass the transformed code to a software model checker. As $m1$-$m3$-$m5$ is an infeasible path (due to the conflicting conditions $z \geq 0$ and $z = -2$), a software model checker satisfies the assertion clause "P1" in Figure 5. Therefore, we conclude that $m1$ cannot be evicted from cache. Similarly, using a separate model checker refinement pass, we can also conclude that $m5$ cannot be evicted from cache. As marked in Figure 5, our code transformation framework $\langle \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} \rangle$ is configured as follows: $\mathcal{L} = \{m5\}$, $\mathcal{A}$ is the "assertion" clause checking the property $C\_1 \leq 0$, $\mathcal{P}_l = \{loc4\}$, $\mathcal{P}_c = \{loc2\}$ and $\mathcal{I} = \{loc1, loc3\}$.

5.3 Refinement of inter-task cache conflicts

We now show the refinement of inter-task cache conflicts (as shown in Figure 3(b)) and thereby reduce the over-estimation introduced by CRPD analysis. A major source of over-estimation in CRPD analysis may come from the computation of evicting cache blocks (ECB). ECB denotes the set of cache blocks possibly touched by the preempting task. Recall that CRPD depends on the set of useful cache blocks (UCBs) replaced from cache due to preemption. Whether a UCB could be replaced due to preemption, on the other hand, depends on the set of ECBs conflicting with it. Therefore, more precise the set of ECBs, more precise the CRPD we get. If ECB computation does not take into account the infeasible paths in preempting task, set of ECBs might be over-approximated. Therefore, over-estimation in CRPD analysis will also increase. Consequently, we use a model checker for refining the number of ECBs by eliminating infeasible paths found in the preempting task.

The refinement of ECBs can be represented in Figure 6. Let us assume $ECB(i)$ represents the number of ECBs computed for cache set i and C_i counts the unique conflicts (in our transformed code) in cache set i generated by the preempting task . The refinement of ECBs are performed in an iterative manner. In each iteration, for all non-zero $ECB(i)$, we try to refine $ECB(i)$ with an immediately smaller value. More precisely, if $ECB(i) = N$, we use the model checker to verify an "assertion" property $C_i \leq N - 1$. Note that we need to count the cache conflicts generated by the entire preempting task. Therefore, the conflict count C_i need to be initialized only once, before any cache blocks accessed by the preempting task and the "assertion" property $C_i \leq N - 1$ is placed immediately before the exit point of the preempting task. If the model checker successfully verifies the "assertion" property $C_i \leq N - 1$, we can guarantee that the preempting task cannot generate more than $N - 1$ conflicts

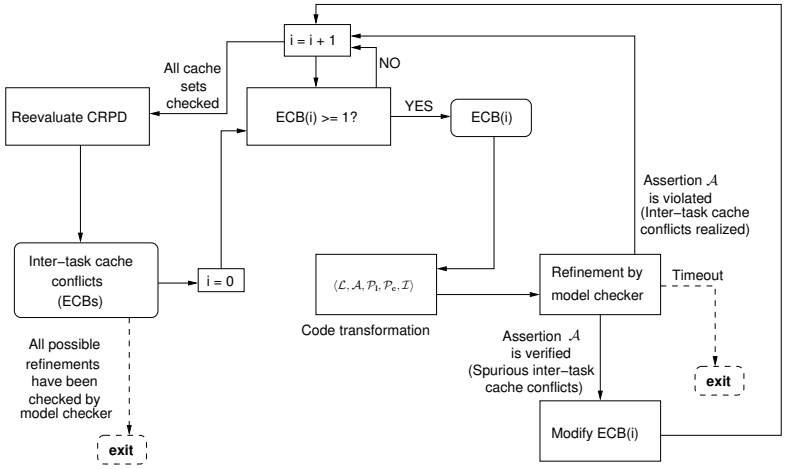


Fig. 6 Refinement of inter-task conflict analysis, ECB(i) denotes the ECB of cache set i

in cache set i. Therefore, we can update $ECB(i)$ with $N - 1$. After each iteration (*i.e.* after checking refinement of $ECB(i)$ with an immediately smaller value for all cache set i), we re-evaluate the CRPD.

The workflow of refinement, as shown in Figure 6, is not restrictive. More specifically, CRPD need not be evaluated after each iteration. The designer may choose to wait for all possible refinements and evaluate the CRPD once all refinements by the MC has been completed. However, evaluation of CRPD after each iteration gives the designer two advantages: first, (s)he can choose to terminate the MC refinement process when a tolerable value of CRPD has been obtained and secondly, (s)he can choose to terminate the MC refinement process if the value of CRPD has not been changed for a reasonably long number of iterations.

We demonstrate the idea using our example in Figure 7. Suppose, the task in Figure 7 is a high priority task which may potentially preempt some low priority task. For sake of illustration, assume a 2-way set associative cache where $m1$ and $m5$ map to the same cache set i. Therefore, $ECB(i) = 2$ and the immediate refinement of $ECB(i)$ would check whether the number of unique conflicts in cache set i is less than or equal to 1. The transformed code is shown in Figure 7. It checks a property $C\_1 \leq 1$ where C_1 counts the number of unique conflicts generated by the preempting task in cache set i. $flag\_m1$ and $flag\_m5$ are used as guards, so that C_1 counts only unique cache conflicts. When the modified code is passed to a software model checker, it can find out the infeasible path $m1$-$m3$-$m5$ and satisfy the property (*i.e.* $C\_1 \leq 1$). Consequently, we can refine the value of $ECB(i)$ as 1. Since $ECB(i)$ is refined to a smaller value, some UCBs in the preempted task may not be evicted from cache set i after preemption, which might in turn lead to a smaller CRPD. As marked in Figure 7, our code transformation framework $\langle \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} \rangle$ is configured as follows: $\mathcal{L} = \{m1, m5\}$, $\mathcal{P}_l = \{loc2, loc3\}$, $\mathcal{A}$ is the the property $C\_1 \leq 1$, $\mathcal{P}_c = \{loc4\}$ and $\mathcal{I} = \{loc1\}$.

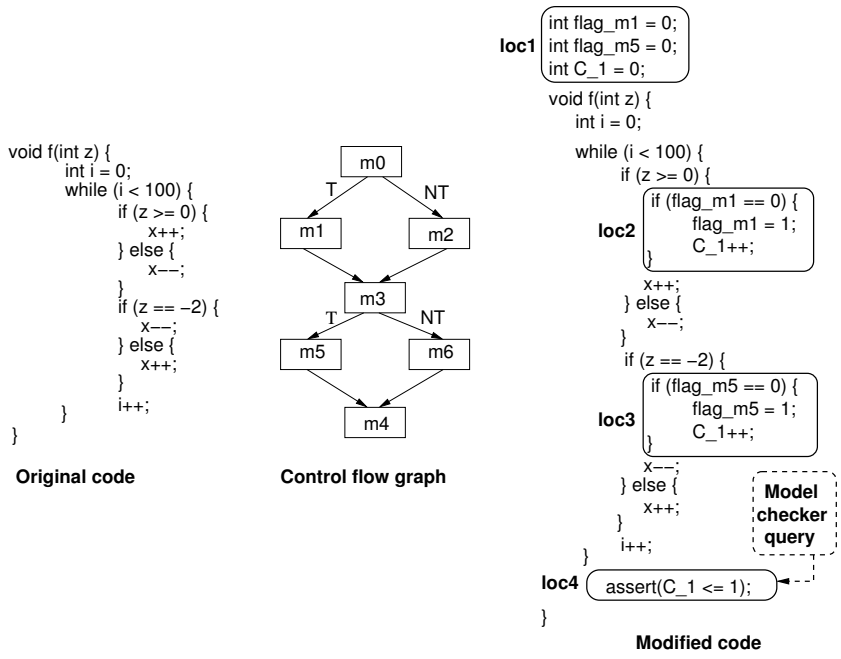


Fig. 7 Inter-task cache conflict refinement

5.4 Refinement of inter-core cache conflicts

Finally, we describe the refinement of inter-core conflicts generated in a shared cache (as shown in Figure 2(c)). Recall from Equation 1 that the precision of shared L2 cache analysis largely depends on the accuracy of estimating the term $|\mathcal{M}_c(m)|$. The model checking pass in our framework refines the set $\mathcal{M}_c(m)$ by exploiting infeasible paths in the conflicting task.

Figure 8 demonstrates the instantiation of our general framework for inter-core conflict refinement. We only target the memory blocks whose categorizations are changed from AH to NC in a shared cache conflict analysis phase. The code transformation is performed on the task which generates inter-core cache conflicts. In this case, the assertion property $\mathcal{A}$ aims to refine the number of inter-core cache conflicts. If the assertion property $\mathcal{A}$ is verified by he model checker, the categorization of a memory block can be reverted back from NC to AH due to the presence of spurious inter-core cache conflicts reported by AI (as shown by the block "Modify categorization from NC to AH" in Figure 8). On the other hand, if the assertion property $\mathcal{A}$ is violated, the refinement process continues for other NC categorized memory blocks in the shared cache.

Consider a memory block m mapping to an N-way associative shared L2 cache set i. Assume that the categorization of memory block m was changed from AH to NC during the shared cache conflict analysis phase. Disregarding the inter-core conflicts, assume the *maximum LRU age* of m in cache set i is denoted by $age(m)$.

Therefore, if the amount of inter-core conflicts (in cache set i) is bounded by $N - age(m)$, we can guarantee that m will remain a shared L2 cache hit, despite inter-core conflicts. Recall that $N - age(m)$ is called the *residual age* of m. Further assume t_c is a task which may generate inter-core cache conflicts and C_i serves the purpose of counting inter-core conflicts in shared L2 cache set i generated by t_c. Therefore, we use the model checker to verify an "assertion" property $C_i \leq N - age(m)$. Identical to inter-task cache conflict refinement, we need to check the total amount of cache conflicts generated by task t_c. Therefore, in our transformed code, we initialize C_i only once, before any cache blocks accessed by t_c and we check the "assertion" property just before the exit point of t_c.

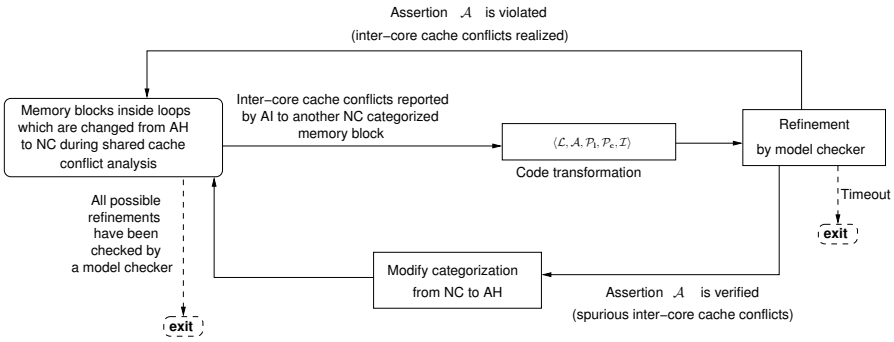


Fig. 8 Refinement of shared cache conflict analysis

Coming back to the example in Figure 3, assume that $m1$ and $m5$ map to the same cache set of a 2-way set associative L2 cache. Further assume that we are trying to refine the inter-core cache conflicts generated to a task t' and t' is running in parallel on a different core with the task in Figure 3. Consider t' accesses a memory block m', which map into the same shared L2 cache set as $m1$ and $m5$. Finally assume that m' is an all-miss (AM) or unclassified (NC) in L1 cache, but an all-hit (AH) in L2 cache *with residual age one*, in the absence of inter-core cache conflicts. Previous analysis will compute $|\mathcal{M}_c(m')|$ as 2 (due to $m1$ and $m5$ in the conflicting task). Since the residual age of m' is one, the categorization of m' will be changed to NC (Equation 1), leading to unnecessary conflict misses. We modify the code to check whether the number of unique inter-core conflicts is less than or equal to the residual age of m'. The transformation is similar to Figure 7 where C_1 serves the purpose of counting unique cache conflicts with m' in shared L2 cache. The model checker will satisfy the assertion P2 in Figure 7 due to the infeasible path $m1$-$m3$-$m5$. Consequently, we shall be able to derive that the amount of inter-core conflicts with m' never exceeds the residual age of m'. Therefore, the categorization of m' is kept all-hit (AH). Configuration of our code transformation framework $\langle \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} \rangle$ is identical to the inter-task cache conflict refinement as follows: $\mathcal{L} = \{m1, m5\}$, $\mathcal{P}_l = \{loc2, loc3\}$, $\mathcal{A}$ is the the property $C\_1 \leq 1$, $\mathcal{P}_c = \{loc4\}$ and $\mathcal{I} = \{loc1\}$.

Although we show the transformation for a two core system, our framework does not have the strict limitation of working only for two cores. However, one model

checker invocation can verify only one task. Therefore, to refine conflicts from X different tasks $t_1, t_2, \ldots, t_X$ running on X different cores, we first employ an additional *compose* phase in transformation. The *compose* phase sequentially composes $t_1, t_2, \ldots, t_X$ (in any order) into a single task T. The infeasible paths in any task $t_1, t_2, \ldots, t_X$ are preserved in task T. Consequently, our code transformation technique can be applied to T in exactly same manner as described in the preceding to refine conflicts from $t_1, t_2, \ldots, t_X$. Since the composition is sequential, number of conflicts are accumulated from all X cores. Model checker refinement passes can then be carried out on task T.

5.5 Optimizations

To reduce the number of calls to model checker, we cache the verification results. Recall that the "assertion" property verified by the model checker was always placed at the end of conflicting task during inter-task and inter-core cache conflict refinement. However, during intra-task cache conflict refinement, the position of "assertion" property may vary and depends on the position of NC categorized memory block being refined. Therefore, the following optimization can be applied only during inter-task and inter-core conflict refinement but *not* during intra-task conflict refinement.

Model checker results are stored as a triple $(set,\ result_{mc},\ conflicts)$. The triple has the following meaning:

- set : Cache set for which the refinement is being made.
- $result_{mc}$: Returned result by the model checker. Assume $result_{mc}$ is one for a successful verification and zero otherwise.
- $conflicts$: Number of conflicts in the assertion property. If we verify an assertion property $C_i \leq N$, value of $conflicts$ is N.

In Figure 3(e), we store $(1, 1, 1)$ after the successful refinement (assuming $m1$ and $m5$ map to cache set 1). Assume any other assertion of form $C_{set'} \leq N'$ is needed to be verified, where set' is the cache set for which the conflicts are being refined. We search the cached results of form $(set,\ result_{mc},\ conflicts)$ and take an action as follows:

- $set = set' \wedge result_{mc} = 0 \wedge N' \geq conflicts$: Assertion failure is returned. If the refinement previously failed for a less number of conflicts, it will definitely fail for more conflicts.
- $set = set' \wedge result_{mc} = 1 \wedge N' \leq conflicts$: Assertion success is returned. If the refinement was previously satisfied for more number of conflicts, it must be satisfied for less number of conflicts.

If none of the entries satisfy the above two conditions, a new call to the model checker is made. Depending on the outcome, the new result is cached accordingly for future use.

6 Instantiation of the general framework via symbolic execution

In Section 5, we have shown the instantiation of our framework using model checking, as also discussed in our previous work [Chattopadhyay and Roychoudhury, 2011]. In this section, we shall discuss the extension of our compositional analysis framework with a symbolic execution engine. Such an extension is built on our previous work [Chattopadhyay and Roychoudhury, 2011] and it forms the key contribution of our work in this paper. As before, we use the abstract interpretation (AI) as a base analysis. We rule out the set of cache conflicts which are accurately analyzed by AI. Rest of the cache conflicts are iteratively refined using our code transformation framework and a symbolic execution engine.

In the following, we first briefly describe the operation of a static symbolic execution engine.

6.1 Static symbolic execution

Symbolic execution [King, 1976] interprets a program with *symbolic input values* (rather than concrete input values). Any expression, whose value depends directly or indirectly on these symbolic input variables, are treated as symbolic expressions throughout the execution. At any point of interpreting the program, symbolic execution maintains a set of execution states. Each such execution state is associated with a *constraint store*. The *constraint store* is a symbolic formula capturing the set of inputs along which the respective execution state is reached. Let us consider an execution state which has to interpret a branch instruction. At a branch location, the symbolic execution must decide which branch to take. If the branch instruction contains a symbolic expression, such a decision making involves constraint solving. If the constraint solver can decide which branch to take, the execution state proceeds along the respective branch (without creating any additional execution state). Such an interpretation of branch instruction is usually called a "nonforking" execution. The more complex scenario appears when the outcome of a branch instruction cannot be decided – which means that there is at least one input which satisfy the *true* leg of the branch and there is also at least one input which satisfy the *false* leg of the branch. In such a scenario, symbolic execution creates two parallel execution states (called "forking" execution), one for the *true* leg of the branch (say *true state*) and the other for the *false* leg of the branch (say *false state*). Assuming that the branch instruction checks a condition θ and the constraint store of the execution state before branch was Φ, the constraint store of the *true state* is updated as $\Phi \wedge \theta$ and the constraint store of the *false state* is updated as $\Phi \wedge \neg\theta$. Both the *true state* and *false state* inherit the same computation state before the branch location, but after the branch location, the two execution states proceed independently.

We shall illustrate the work flow of a symbolic execution engine with the example in Figure 9. Let us assume that z is an input to the program and therefore, z is marked as *symbolic*. If the value of an expression does not depend on any of the symbolic variables, the expression value is treated as *concrete* (*i.e.* input independent). In Fig-

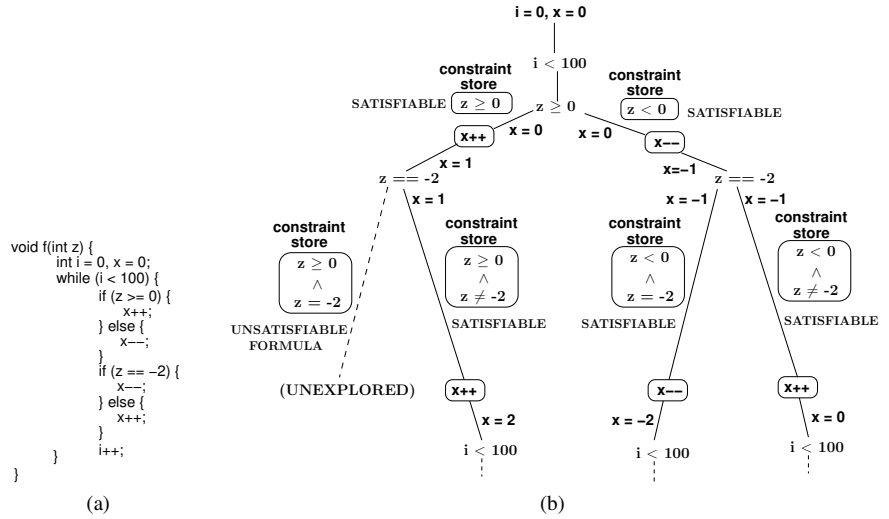


Fig. 9 (a) Example program, (b) symbolic execution

ure 9, any update on variable i and x are interpreted as *concrete* values, as the updates on i and x are not data dependent on the value of z.

Recall that a constraint store is maintained for each execution state created during symbolic execution. The constraint store is a symbolic formula on the input variables which *must be satisfied to reach the respective execution state*. The constraint store is *the logical formula true* at the beginning of the program and is adjusted at each branch instruction. In example 9(b), the program hits the $i < 100$ branch instruction first. Since i is not an input and is initialized 0, only the *true* leg of the branch instruction is interpreted.

However, consider the branch instruction $z \geq 0$, when being hit for the first time. At this point, the constraint store is *the logical formula true*. This branch condition is sent as a query to the constraint solver to decide the condition outcome (*i.e.* true or false). The constraint solver consults the constraint store to decide the outcome of the branch condition. Since the constraint store is *the logical formula true*, the outcome of $z \geq 0$ could be both *true or false* depending on the value of input z. Therefore, the symbolic execution forks two different execution states for each leg of the branch instruction. The constraint store at the true leg is updated as $z \geq 0$ and the same at the false leg is updated as $z < 0$. The content of the constraint store is shown beside the control flow edges.

Now consider the execution state with constraint store $z \geq 0$. When this execution state hits the branch instruction $z == -2$, the constraint solver checks the satisfiability of the formula $z \geq 0 \wedge z = -2$, which is clearly *unsatisfiable*. The unsatisfiability of such formula can be checked very fast by an SMT solver with the theory of linear integer arithmetic. Therefore, the symbolic execution does not create any execution state which corresponds to the unsatisfiable constraint store $z \geq 0 \wedge z = -2$ (as marked "UNEXPLORED" in Figure 9(b)).

When the execution state with constraint store $z < 0$ hits the branch location $z == -2$, both the formulae $z < 0 \wedge z = -2$ and $z < 0 \wedge z \neq -2$ are *satisfiable* for some input. Therefore, the symbolic execution forks two execution states accordingly. As shown in Figure 9(b), both these execution states inherit the value of $x = -1$ before the branch location $z == -2$, however, proceeds independently thereafter to update $x = -2$ (for the execution state with constraint store $z < 0 \wedge z = -2$) and update $x = 0$ (for the execution state $z < 0 \wedge z \neq -2$).

Eventually, only three different execution states are created (as shown in Figure 9(b)) with their respective constraint stores as follows:

- $z \geq 0 \wedge z \neq -2$,
- $z < 0 \wedge z = -2$, and
- $z < 0 \wedge z \neq -2$

The symbolic execution is terminated when it finishes interpreting all the instructions in all the three execution states (as shown in the preceding).

6.2 Cache conflict refinement

Symbolic execution has successfully been applied to discover many critical functionality bugs [Cadar et al, 2008]. At a high level, our code transformation framework can be viewed as reducing the problem of cache timing checking to functionality checking. Recall that our code transformation framework contains an assertion property $\mathcal{A}$ to check whether certain cache conflicts in the program are *spurious*. This assertion property can be checked for validity using symbolic execution. If the assertion property $\mathcal{A}$ is violated at any execution state created by the symbolic execution, the entire symbolic execution is *aborted*. Such an abnormal termination of the program captures the fact that certain cache conflicts (captured by $\mathcal{A}$) can be realized for some execution of the program and therefore, such cache conflicts are not *spurious*. On the other hand, if the symbolic execution is not aborted, we can prove that our introduced assertion holds over all possible executions of the program. Consequently, the cache conflict captured by the assertion property is *spurious*.

We shall demonstrate the refinement process through the example in Figure 10. Figure 10(a) shows the instrumented code for intra-task cache conflict refinement (we use the same example from Figure 5). Figure 10(b) shows the cache conflict refinement process via symbolic execution. Figure 10(b) shows that only one execution state (among all three) can execute the assertion property involving the variable C_1. As evidenced by Figure 10(b), the execution state interpreting the assertion property captures an input condition $z \geq 0$. Since symbolic execution interprets the program, at each program point it holds the value of all the registers and memory locations. At the assertion location, the respective execution state checks whether the currently stored values satisfy the assertion. Since C_1 has a value of zero initially, a formula of the form $C\_1 = 0 \wedge C\_1 \leq 0$ is sent to the constraint solver as a query. If the constraint solver returns a *satisfiable* formula, we can conclude that the assertion property holds for the corresponding execution. Note that C_1 is incremented only for the execution state which satisfy input condition $z = -2$. On the other hand, the assertion property

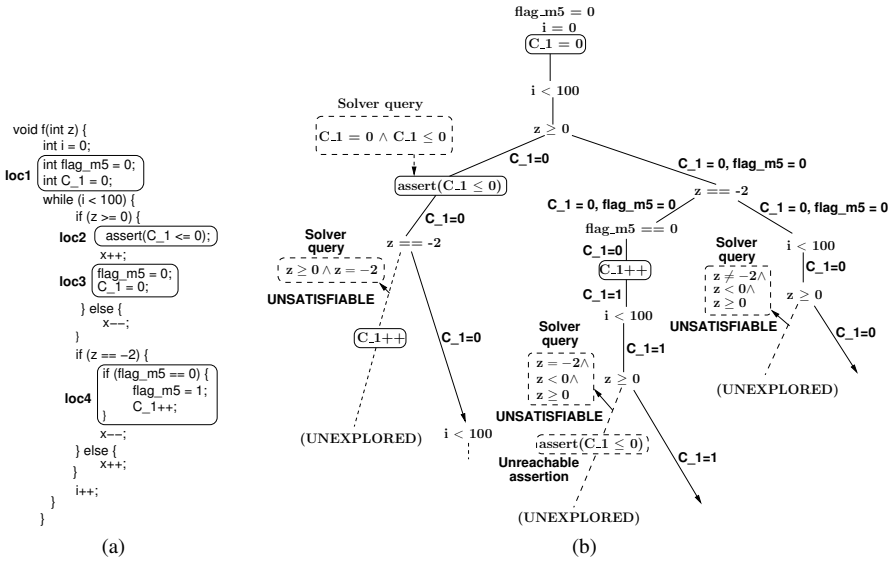


Fig. 10 (a) Transformed code for checking cache conflict, (b) checking the assertion during symbolic execution

is reachable only if the input condition $z \geq 0$ is satisfied. As a result, none of the execution states which increment the variable C_1 can reach the assertion property (as marked "Unreachable assertion" in Figure 10(b)). Consequently, whenever the assertion property is reached, the same formula (*i.e.* $C\_1 = 0 \wedge C\_1 \leq 0$) is sent to the constraint solver. Therefore, symbolic execution is never aborted for the example and we can conclude that the cache conflicts captured by the instrumented code in Figure 10(b) cannot appear in *any real execution*. Inter-task and inter-core cache conflicts are also refined in a similar fashion as shown in Figure 10(b).

Note that the symbolic execution engine tries to reason about a program *path-by-path*. Due to this *path sensitive* reasoning process, such a symbolic execution may generate very precise result compared to an equivalent abstract interpretation based analysis. Since the sole purpose of our refinement process is to check the inserted assertion property, the symbolic execution can be aborted as soon as a violation of the assertion property is reached. As a result, a violation of the assertion is likely to be checked much more quickly than the *validity* of the same assertion.

7 Implementation and evaluation

7.1 Implementation using CBMC/SPIN

We have used the Chronos timing analysis tool [Li et al, 2007] in which we have already integrated the AI-based cache analysis proposed in [Theiling et al, 2000] (for single core) and [Li et al, 2009] (for multiple cores). Chronos employs detailed micro-

architectural modeling (superscalar, out-of-order pipeline and branch prediction). We have also integrated the recently proposed CRPD analysis ([Altmeyer et al, 2010] and [Altmeyer and Burguière, 2009]) into Chronos. For model checking purposes, we use C bounded model checker (CBMC) [Clarke et al, 2004] or SPIN model checker [SPIN, 1991].

SPIN is a linear time temporal logic (LTL) based model checker targeted for efficient software verification. SPIN can also be used as an exhaustive verifier to prove several correctness properties of a system. Recall that our proposed code transformation framework introduces an assertion property to check whether certain cache conflicts in the code are realizable. Such assertions can be considered as the correctness properties of the transformed code. As a result, we can use the SPIN model checker for refining cache conflicts in our proposed framework. SPIN provides direct support for using embedded C code in the specification. As a result, we can use the SPIN model checker for verifying C programs.

CBMC formally verifies ANSI-C programs through bounded model checking (BMC) [Clarke et al, 2001]. In the context of our proposed approach, bounded model checking through CBMC requires some explanation as follows. For a given system/program P, BMC unwinds P to a certain depth. After unwinding, a Boolean formula is obtained that is satisfiable if and only if there exists a counter example trace. The formula is checked by a SAT procedure. If the formula is satisfiable, a counter example is produced from the output of SAT procedure. Technically, for a C program, the unwinding is achieved by unrolling the program loops to a certain depth. For a given unwinding depth n, CBMC unwinds a loop by duplicating the code of loop body n times. Each copy is guarded by the loop entry condition and hence, covering the cases where the loop executes for less than n iterations. The main advantage of CBMC is that the tool also checks whether sufficient unwinding has been done and thereby ensures that no *longer* counterexample can exist. Technically, CBMC achieves the same by putting an "assertion" (called *unwinding assertion*) after the last copy of the unrolled loop. The assertion uses the negated loop entry condition and therefore, it ensures that the program never requires more iterations. *In summary, if no counterexample is produced by CBMC, it ensures the absence of error in the program for any execution*.

As described in the preceding, CBMC requires unwinding depth (bound) of each loop. If user does not specify any unwinding depth (loop bound), CBMC tries to determine the depth automatically. In most of our experiments, CBMC was able to determine the loop bound automatically. For the cases where CBMC failed to determine the loop bound, we passed sufficient loop bound for each loop as an input to CBMC. Recall that CBMC automatically put an "assertion" clause (called an *unwinding assertion*) after the last unwound copy of a loop. The assertion clause verifies the negated loop entry condition. Therefore, if insufficient loop bound is provided by the user, CBMC generates an unwinding assertion violation and the verification process returns a failure. Consequently, user can give a larger loop bound and rerun CBMC. However, in our experiments, we initially provided sufficient loop bounds, so that no unwinding assertion is violated. In our current implementation, CBMC is called as an external module. Therefore, for each different call of CBMC, the loop unwinding needs to be performed. Running time of our analysis can certainly improve if we can

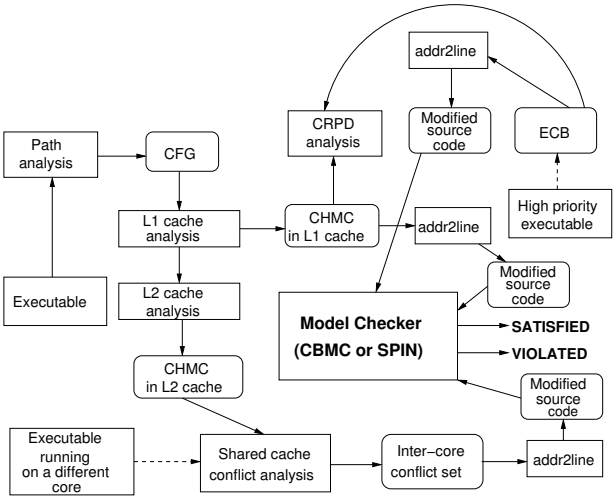


Fig. 11 Implementation framework using model checker (CBMC or SPIN)

restrict the number of loop unwindings. This will require us to make use of CBMC and Chronos in a single binary executable, which could be explored in future.

Figure 11 gives an overall picture of our implementation framework. The figure demonstrates one refinement for each type of conflicts. Chronos employs AI-based cache analysis directly on the executable. We use a utility `addr2line` which converts an instruction address to corresponding source code line number. The information generated by `addr2line` is used to generate the transformed code. The transformation of code is *entirely automatic*. Note that the sole purpose of the transformed code is to prove that certain cache conflicts in the original code are *infeasible*. Therefore, the timing effects generated by the original code is entirely independent of the additional code introduced in transformation. The transformed code contains an "assertion" property to be verified by the model checker (*i.e.* CBMC or SPIN). The model checker either successfully verifies the assertion property or generates a counter example. We would finally like to point out that the central contribution of this paper is an efficient composition of abstract interpretation and model checking. Therefore, even though we have used CBMC and SPIN for model checking, our proposed framework (Figure 1) remains unchanged if we use a model checker that directly works on the executable (*e.g.* [Balakrishnan et al, 2005]). Nevertheless, there are certain advantages of using a model checker like [Balakrishnan et al, 2005]. Since [Balakrishnan et al, 2005] directly works on the executables, it can capture the effect of all compiler optimizations. Our technique can be integrated with [Balakrishnan et al, 2005] to make a more *robust* WCET analysis framework.

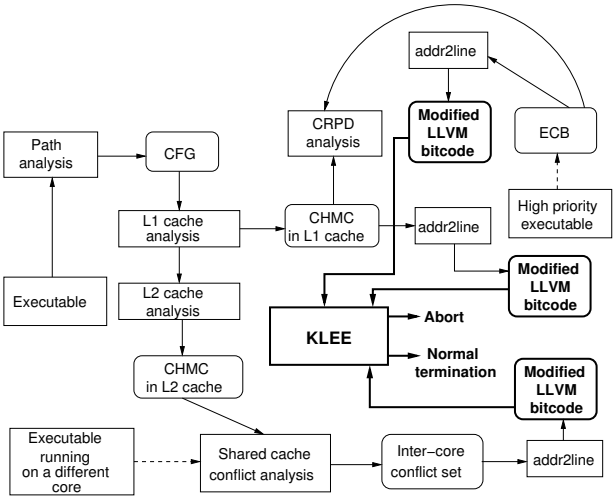


Fig. 12 Implementation framework using KLEE

7.2 Implementation using KLEE

KLEE [KLEE, 2008] is a symbolic execution engine based on LLVM [LLVM, 2003] compiler infrastructure. KLEE uses the power of satisfiability modulo theory (SMT) and the SMT-based solvers to explore different paths in a program.

Figure 12 shows our implementation framework using KLEE. The basic structure of the implementation is same as in Figure 11. The modifications made to use KLEE have been highlighted in Figure 12. KLEE is a symbolic execution engine based on the LLVM bitcode format. Therefore, our transformation is made at the level of LLVM bitcode. Recall that the instantiation of our code transformation framework depends on the type of cache conflict being refined (*i.e.* intra-task, inter-task or inter-core). Therefore, as evidenced by Figure 12, the modified LLVM bitcode depends on the type of cache analysis being refined via KLEE. KLEE allows to specify assertions, which are checked during the symbolic execution using STP [STP, 2007] constraint solver. Originally, KLEE ignores the assertions with a warning and continues symbolic execution. We modify the source code of KLEE to terminate the symbolic execution as soon as it reaches the violation of some assertion property.

It is important to note that the above checking procedure is entirely different from CBMC. In CBMC, the checking of an assertion property is captured by a single SAT formula. The SAT formula takes care of all the different program paths that may reach the assertion. Therefore, in general, the SAT formula created by CBMC is very large. On the other hand, KLEE does not check the assertion by a single formula. KLEE checks the assertion property while interpreting the program. Therefore, each time the assertion is interpreted, an SMT solver is asked to check the satisfiability of the assertion. The symbolic and concrete values at the assertion location are used to validate the assertion property. Note that each interpretation of the assertion captures a single program path and therefore, the formula checked by the SMT solver is usu-

ally much simpler than the single SAT formula generated by CBMC. Nevertheless, an SMT solver is called many times to check the assertion property, whereas CBMC calls a SAT solver only once.

Since our sole purpose is to check the assertions introduced in the modified code, we do not need to continue execution if the assertion is violated in some execution state. As a result, KLEE can usually check the violation of an assertion property much faster than CBMC.

7.3 Experimental setup

We have chosen programs from [Gustafsson et al, 2010] which are generally used for timing analysis. Note that the main motivation of our work is to remove *spurious cache conflicts*, which were introduced due to the infeasible paths. Infeasible paths are often introduced when auto generating code from a high level modeling language (*e.g.* `esterel` as shown in [Ju et al, 2008]). For evaluation of our framework, therefore, we need a set of tasks which potentially exhibit many paths. Table 1 demonstrates a set of subject programs having multiple paths. Let us call the set of tasks in Table 1 as *conflicting task set*. All the program verification passes (*i.e.* using CBMC model checker, SPIN model checker or KLEE symbolic execution engine) are used to refine the three different types of cache conflicts (*i.e.* intra-task, inter-task and inter-core) generated by the conflicting task set. We use another set of subject programs from [Gustafsson et al, 2010] as shown in Table 2 during inter-task and inter-core conflict refinement. We call the tasks in Table 2 as *standard task set*. During inter-task and inter-core conflict refinement, we refine the conflicts generated by conflicting task set on the standard task set. We report our experiences for each possible combinations of standard and conflicting task set.

Table 1 Conflicting task set

Task	Description	Lines of C code	code size (bytes)
statemate	Automatically generated code from Real-time-Code generator STARC	1230	52618
compress	Data compression program	506	13411
nsichneu	Simulate an extended petri-net	4225	118351

Table 2 Standard task set

Task	Description	Lines of C code	code size (bytes)
cnt	Counts non-negative numbers in a matrix	128	2880
fir	Finite impulse response filter	275	11965
fdct	Fast discrete cosign transform	214	8863
jfdctint	discrete cosign transform on 8×8 block	324	16028
edn	signal processing application	283	10563
ndes	complex embedded code	235	7345

We use the following terminology in presenting the experimental data:

- $WCET_{base}$: WCET before any refinement by program verification.
- $WCET_{refined}$: WCET after refinement by program verification.
- $CRPD_{base}$: CRPD before any refinement by program verification.
- $CRPD_{refined}$: CRPD after refinement by program verification.

WCET improvement is computed as $\frac{WCET_{base}-WCET_{refined}}{WCET_{base}} \times 100\%$. CRPD improvement is computed similarly as $\frac{CRPD_{base}-CRPD_{refined}}{CRPD_{base}} \times 100\%$.

To refer to different verification techniques used in our framework, we shall additionally use the following terminologies:

- $AI + CBMC$: Framework using abstract interpretation (AI) and CBMC model checker.
- $AI + SPIN$: Framework using abstract interpretation (AI) and SPIN model checker.
- $AI + KLEE$: Framework using abstract interpretation (AI) and KLEE symbolic execution engine.

Our framework uses the usual 5-stage pipeline (IF-ID-EX-MEM-WB) implemented by Chronos when predicting the WCET value. We fix the L1 cache miss latency as 6 cycles and L2 cache miss latency as 30 cycles for all the experiments. For the experiments which do not have an L2 cache (*e.g.* inter-task and intra-task conflict refinement), we simply take the L1 cache miss penalty as 36 cycles. All reported experiments have been performed in an Intel Core i7 processor having 4 GB of RAM and running ubuntu 10.04 operating system. The reported total time captures the entire time taken during the analysis — including the base analysis through abstract interpretation and repeated invocation of path sensitive program verification steps.

7.4 Evaluation

Key result Before going into the details of each experiment, let us first demonstrate the key result of this paper via Figures 13-15. Figures 13-15 show the average WCET and CRPD improvements using different path sensitive program verification methods. The improvement of WCET is demonstrated in case of intra-task and inter-core cache conflict refinement (Figure 13 and Figure 15). On the other hand, CRPD improvement is shown for inter-task cache conflict refinement (Figure 14). We observe an almost linear improvement in WCET and CRPD estimation with respect to time. Inter-task and inter-core cache conflict refinement is much faster compared to intra-task cache conflict refinement, as evidenced by Figure 14 and Figure 15. Recall that we can store and reuse the results returned by program verification methods during inter-task and inter-core cache conflict refinement phases (Section 5.5). Such reuses of program verification results reduces a significant number of calls to model checkers or the symbolic execution engine. As a result, inter-core and inter-task cache conflict refinement are comparably much faster than the intra-task cache conflict refinement.

Figures 13-15 also capture the efficacy and performance of different standard verification tools on the proposed approach. As evidenced by Figures 13-15, the performance of KLEE symbolic execution engine is the best among the three (*i.e.* SPIN,

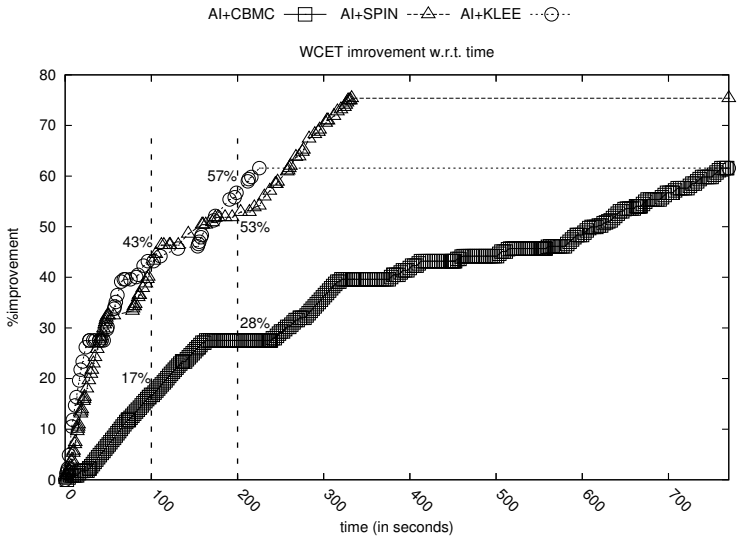


Fig. 13 WCET improvement in single core w.r.t. time using *statemate*

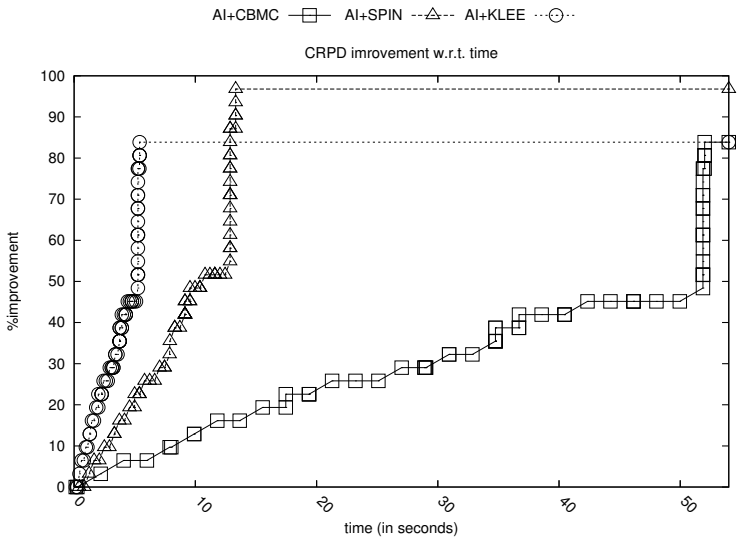


Fig. 14 CRPD improvement w.r.t. time using *statemate* as the high priority task

CBMC and KLEE), whereas the performance of SPIN model checker lies somewhere between the performance of CBMC model checker and KLEE symbolic execution engine. It is, however, worthwhile to note that SPIN model checker is able to generate the best WCET improvement compared to CBMC and KLEE. As shown in Figure 13, the use of SPIN model checker may lead to 75% WCET improvement. On

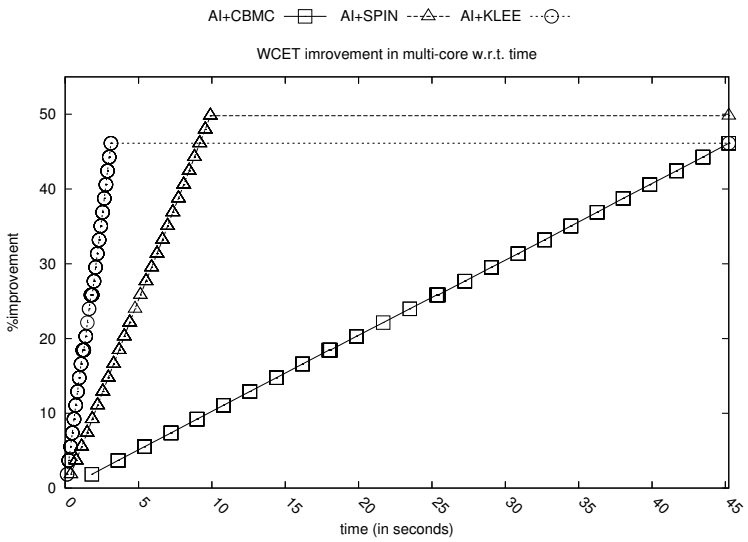


Fig. 15 WCET improvement w.r.t. time using *statemate* as the conflicting task

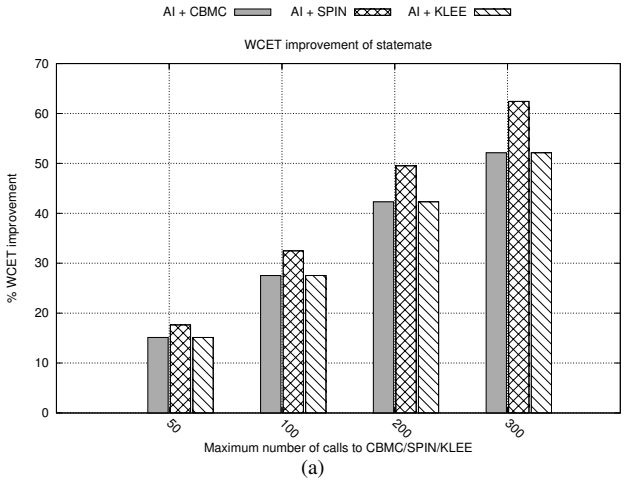
the other hand, the maximum WCET improvement obtained by CBMC and KLEE is below 70%.

As our result is always *safe*, a provably correct WCET/CRPD value can be obtained from any vertical cut along the time axis of Figures 13-15. As illustrated in Figure 13, consider the vertical cut at 100th second. It clearly shows that if we end the refinement process after 100 seconds, we can obtain 17%, 43% and 43% improvement using CBMC, SPIN and KLEE, respectively. Nevertheless, if the refinement process is allowed more time to run, we can obtain better precision in our obtained result (28%, 53% and 57% respectively using CBMC, SPIN and KLEE after 200 seconds, as shown in Figure 13). A similar observation can be made during the inter-task and inter-core cache conflict refinement (as evidenced by Figure 14 and Figure 15 respectively).

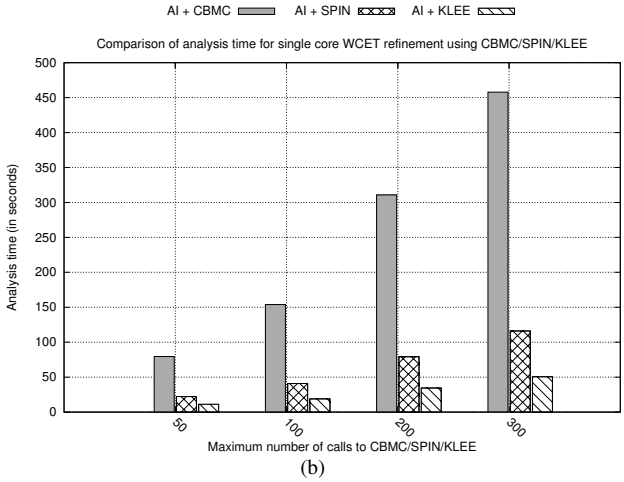
7.5 Reducing intra-task cache conflicts

Our refinement depends both on the choice of conflicting task set and the cache size. We choose an 4-way associative, 32 KB L1 cache with 64 bytes of block size. Applying intra-task cache analysis on `compress` does not leave any NC categorized memory blocks inside loop. Therefore, our refinement pass did not have any additional effect in improving the WCET for `compress`. On the other hand, `statemate` and `nsichneu` contain very large loops (in terms of code size) with multiple paths inside a loop body. Consequently, AI-based cache analysis generates a large number of NC categorized memory blocks.

The results obtained for `statemate` and `nsichneu` are shown in Figure 16(a) and Figure 17(a), respectively. The overall analysis time is reported in Figure 16(b)



(a)

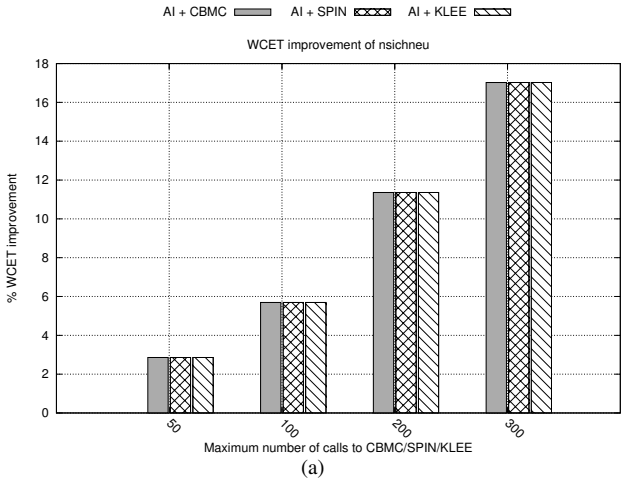


(b)

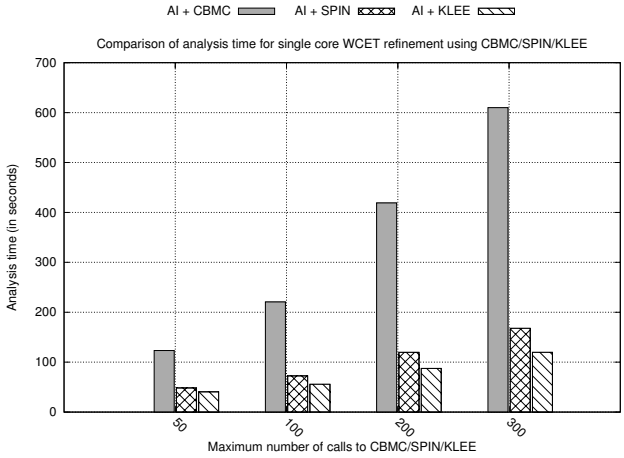
Fig. 16 (a) WCET improvement of `statemate`, (b) analysis time

and Figure 17(b), for `statemate` and `nsichneu`, respectively. Both the figures 16(a) and 17(a) show almost linear improvement in WCET estimation with respect to the number of refinement steps. For these experiments, we restrict the number of refinements to 300. Nevertheless, if time budget permits, the refinement process can be run longer and thereby provides more opportunities to improve the WCET. For `nsichneu`, the contribution of cache misses to the overall execution time is less compared to `statemate`. Consequently, the improvement in WCET is also much smaller compared to the same for `statemate`.

Figures 16(a)-17(a) clearly show that we can obtain the same improvement in WCET estimation using CBMC and KLEE. On the other hand, refinement using



(a)

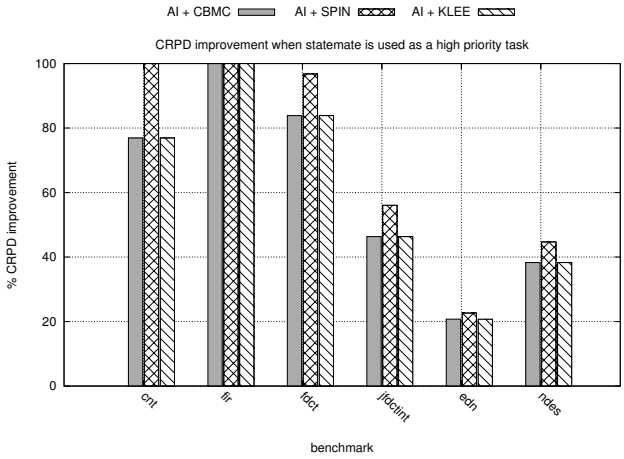


(b)

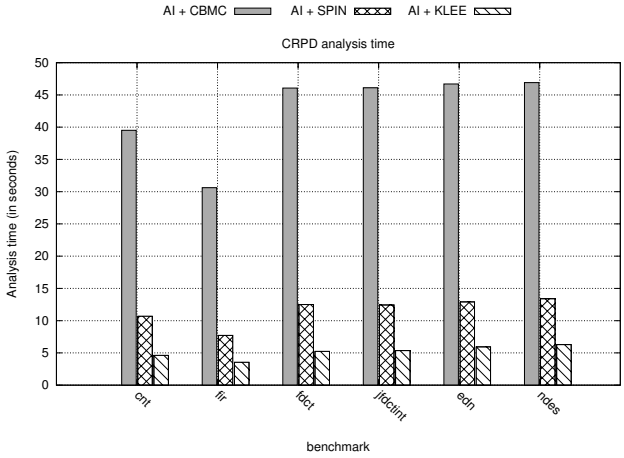
Fig. 17 (a) WCET improvement of `nsichneu`, (b) analysis time

SPIN model checker generates a better improvement in WCET estimation compared to CBMC or KLEE for `statemate` (as evidenced by Figure 16(a)).

Figures 16(b)-17(b) compare the analysis time overhead using CBMC, SPIN and KLEE. As evidenced by Figures 16(b)-17(b), KLEE outperforms the rest. KLEE significantly improves the analysis time compared to CBMC, with the maximum reduction from 500 seconds to 50 seconds. The analysis overhead of SPIN model checker is slightly worse than KLEE, but it can still outperforms CBMC on our proposed approach by a significant magnitude. Moreover, as mentioned in Figure 16(a), refinement using SPIN model checker leads to the best WCET improvement in `statemate`.



(a)



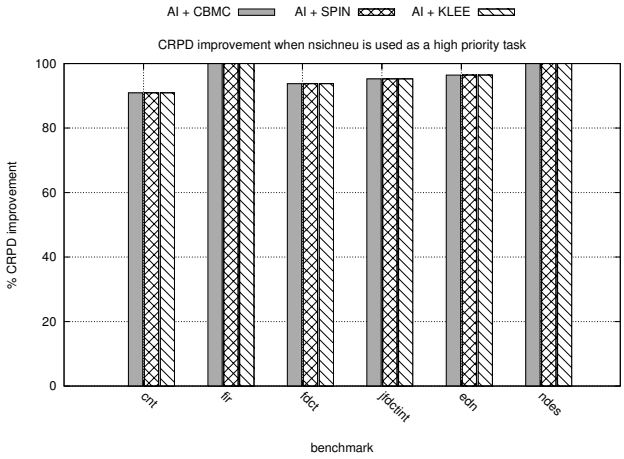
(b)

Fig. 18 (a) CRPD improvement using `statemate`, (b) analysis time

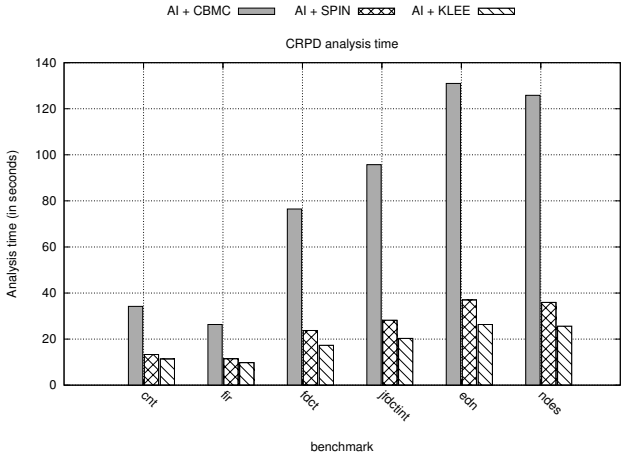
This result demonstrates the potential of our approach even for improving the most basic cache conflict analysis through AI.

7.6 Reducing inter-task cache conflicts

We present the results of inter-task cache conflict refinement through Figures 18(a)-20(a). Figure 18(b)-20(b) report the respective analysis time. The reported CRPD denotes the cache related preemption delay when a low priority task from the standard task set (Table 2) is preempted by a high priority task from the conflicting task set (Table 1). As before, we choose an 4-way associative, 32 KB L1 cache with 64 bytes



(a)

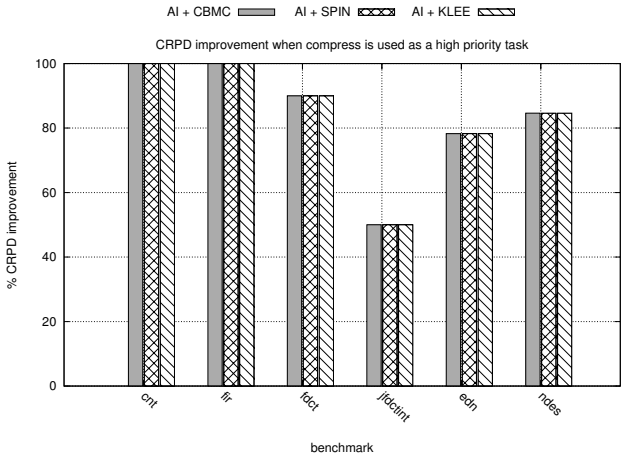


(b)

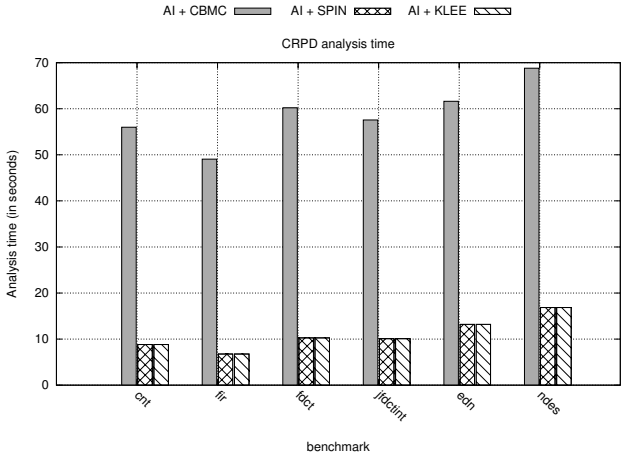
Fig. 19 (a) CRPD improvement using `nsichneu`, (b) analysis time

block size. Unlike the intra-task cache conflict refinement, results reported in Figure 18(a) run the refinement process till the end (*i.e.* until all possible refinements have been checked by CBMC, SPIN or KLEE).

We are able to reduce the number of ECBs as well as the CRPD when `compress`, `statemate` and `nsichneu` are used as high priority tasks. CRPD improvement is significant, with average improvement being more than 80%. Note that we use a set associative cache. Therefore, conflicts generated from high priority tasks may just age the used cache blocks in the low priority tasks (instead of completely evicting the used cache blocks by the low priority task). Consequently, for a few cases, we



(a)

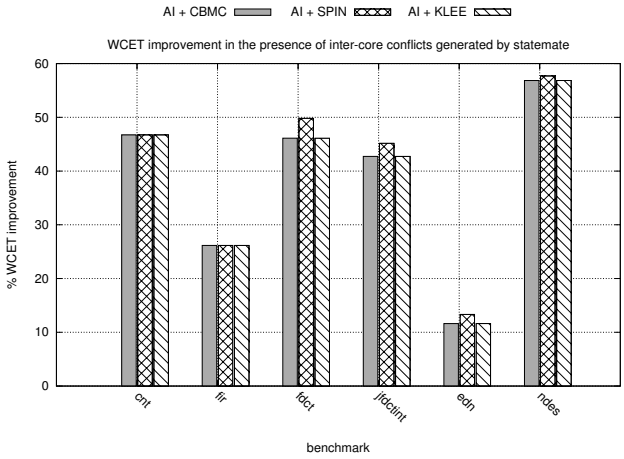


(b)

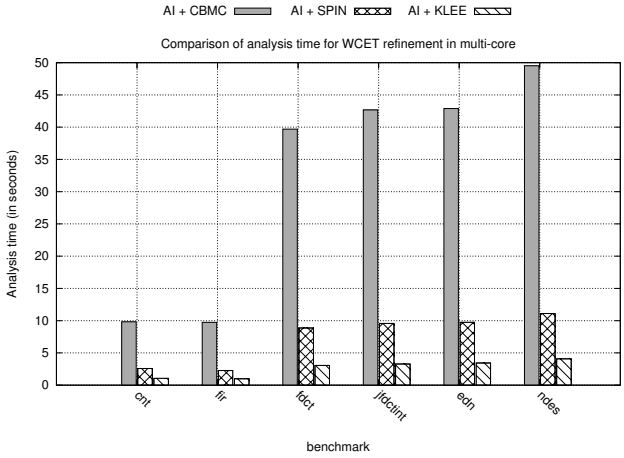
Fig. 20 (a) CRPD improvement using `compress`, (b) analysis time

are able to completely eliminate the CRPD (*e.g.* when `nsichneu` is used as a high priority task to preempt `fir` and `ndes`, Figure 19(a)).

Similar to the intra-task cache conflict refinement, KLEE and CBMC produce the exactly same precision gain in CRPD. For all the subject programs, KLEE and CBMC are able to refine the same number of inter-task cache conflicts, thereby reducing the CRPD by the exactly same amount. This result is evidenced by Figures 18(a)-20(a). On the other hand, when `statemate` is used as a high priority task, refinement using SPIN model checker usually leads to better precision gain in CRPD estimation compared to CBMC or KLEE. Note that such a positive result using SPIN model checker was also observed while refining intra-task cache conflicts in `statemate`.



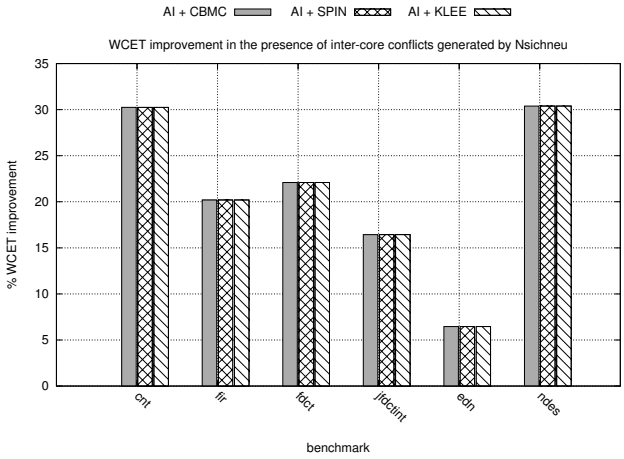
(a)



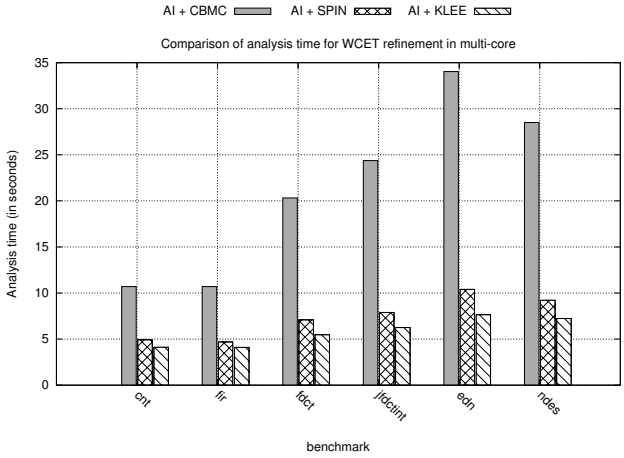
(b)

Fig. 21 (a) Multi-core WCET improvement using `statemate`, (b) analysis time

We compare the time taken by different path sensitive verification tools (*i.e.* CBMC, SPIN and KLEE) for the inter-task cache conflict refinement. The result is shown through Figures 18(b)-20(b). We can observe that the time taken by KLEE is much less compared to CBMC, but comparable to SPIN model checker, for all the subject programs. Although we can gain the same precision using KLEE, the refinement is much faster than CBMC, with as much as 9x times faster when `statemate` is used as a high priority task (Figure 18(b)).



(a)

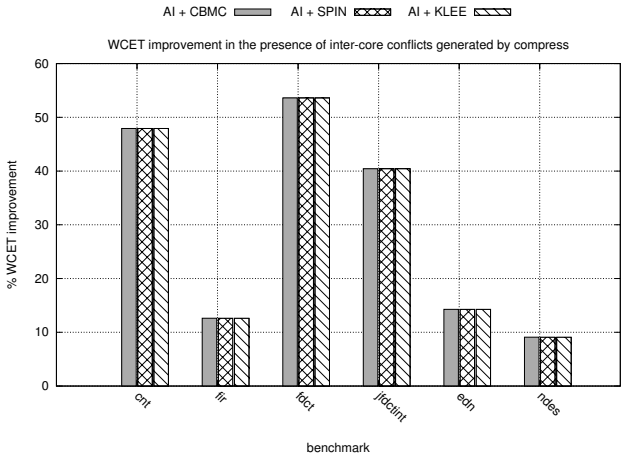


(b)

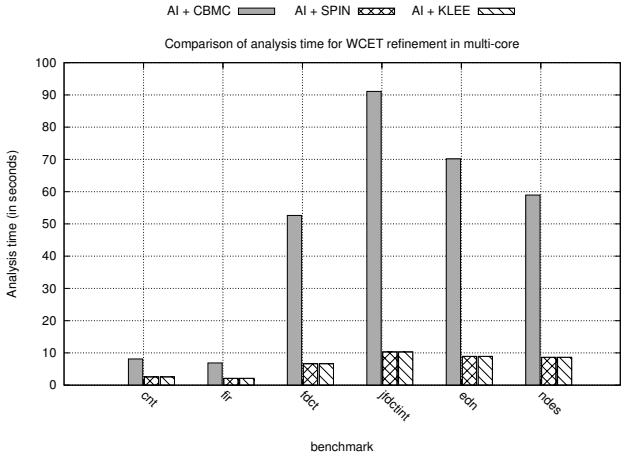
Fig. 22 (a) Multi-core WCET improvement using `nsichneu`, (b) analysis time

7.7 Reducing inter-core cache conflicts

Finally, we present the results of inter-core cache conflict refinement in Figures 21(a)-23(a). The analysis overhead of different experiments are reported in Figures 21(b)-23(b). In one core, we run a task from the standard task set (in Table 2) and in another core, we run a task from the conflicting task set (in Table 1). Reported WCET improvements capture the WCET improvements from the standard task set. For the experiments reported in Figure 21(a), we need the analysis of both L1 and L2 cache. We fixed the L1 cache as a direct-mapped, 256 bytes with a block size of 32 bytes. L1 cache is taken relatively small so that we are able to generate reasonable number



(a)



(b)

Fig. 23 (a) Multi-core WCET improvement using `compress`, (b) analysis time

of conflicts in the shared L2 cache. We take an 4-way associative, 32 KB shared L2 cache having a cache block size of 64 bytes.

We are able to significantly reduce the standard task WCET by refining the inter-core cache conflicts (maximum improvement around 50%). Similar to the inter-task cache conflict refinement, we run the refinement process until we had checked all possible and spurious inter-core cache conflicts. None of our experiments took more than two minutes to complete.

Similar to the refinement of intra-task and inter-task cache conflicts, inter-core cache conflict refinement using KLEE and CBMC also lead to the exactly same precision gain in WCET. Moreover, SPIN model checker gives the best precision gain in

WCET estimation when `statemate` is used to generate the inter-core cache conflicts.

Figures 21(b)-23(b) compare the analysis time overhead using different verification methods. For `nsichneu` and `compress`, KLEE generates the same results at least twice faster than CBMC. On the other hand, for `statemate`, usage of KLEE leads to a significant improvement in analysis time – with as much as 900% for a few subject programs (Figure 21(b)). For all the cases, the time taken by SPIN model checker is significantly lower than CBMC and in most of the cases, the analysis overhead using SPIN model checker is comparable to the same using KLEE.

7.8 Discussion

We have evaluated our proposed framework using two different path sensitive program verification techniques – model checking and symbolic execution. For model checking, we have used CBMC - a SAT-based model checker and SPIN - an explicit-state search based LTL model checker. For symbolic execution, we have used KLEE. In our experiments, CBMC was unable to infer some of the loop bounds in a program automatically. In particular for `statemate`, we provided sufficient loop bounds, so that no unwinding assertion is violated (recall that an unwinding assertion is violated when the user given loop bound may under-approximate the number of times the loop body can be executed). On the other hand, since KLEE performs a symbolic execution of the program, it was able to automatically detect the termination of all the loops and no manual intervention was required during the experiments using KLEE. Our evaluation shows that both KLEE and CBMC improve the analysis precision by exactly same amount. However, the evaluation using SPIN model checker leads to a better precision compared to using KLEE or CBMC in case of `statemate`. A symbolic execution guided refinement process is *faster* than the refinement process based on model checking. This time efficiency of symbolic execution has been made possible by the recent advances in SMT technologies. KLEE uses the fast SMT solver STP for constraint solving, hence improving the refinement process of our framework significantly. Moreover, if any execution of an assertion property leads to a violation, the entire symbolic execution by KLEE can be terminated. This in turn makes the violation check of an assertion much faster than CBMC. Although SPIN model checker takes more time than KLEE to complete, the additional overhead is negligible.

Our experience with the evaluation process shows that both the model checking and symbolic execution lead to exciting results. Whereas symbolic execution outperforms the model checkers in terms of analysis time, model checking may sometime lead to better improvement in WCET estimate (*e.g.* using SPIN model checker for refining the cache conflicts generated by `statemate`). Therefore, use of model checker and symbolic execution has a precision-time tradeoff on our proposed approach. As a result, we believe that both the approaches of path sensitive program verification (*i.e.* model checking and symbolic execution) should be explored in the context of WCET estimation. However, note that we do not make any generality claim about the performance of different program verification techniques in this paper. All the claims shown through our evaluation are only meaningful for our proposed ap-

proach. Therefore, all the program verification techniques and tools used in this paper are of equal interest for a variety of other applications and technologies.

8 Extension

In this section, we shall discuss the modifications required to our proposed framework for analyzing data caches and caches with non-LRU replacement policies. Note that the baseline analysis used by our framework is abstract interpretation (AI). Therefore, in the following discussion, we shall assume the presence of an AI-based framework for data caches and caches with non-LRU replacement policies. We shall show the instantiation of our general code transformation framework for such analyses.

8.1 Caches with non-LRU replacement policies

Although *least-recent-used* (LRU) replacement policy has been argued to be the most suitable for real-time application [Wilhelm et al, 2009], LRU is difficult to implement in hardware. As a result, many commercial embedded processors employ non-LRU replacement policies. *First-in-first-out* (FIFO) is such a non-LRU cache replacement policy and FIFO replacement policy has been widely used in ARM processor families (*e.g.* ARM9 and ARM11).

WCET research community has investigated abstract interpretation (AI) based analysis of FIFO replacement policies [Grund and Reineke, 2010a]. The work in [Grund and Reineke, 2010a] employs *must* and *may* cache analysis to categorize memory blocks as *all-hit* (AH) and *all-miss* (AM). If a memory block cannot be categorized as AH or AM, it is categorized *unclassified* (NC). Therefore, the output generated by the AI-based analysis has the same format as in the case of LRU replacement policy and we can use our framework to refine the set of NC categorized memory blocks. More precisely, the instantiation of our framework for intra-task cache conflict refinement is exactly the same as shown in Figure 4.

Nevertheless, for FIFO replacement policy, the code transformation will be different compared to the LRU replacement policy. We shall illustrate this difference through an example in Figure 24. We use the same example used previously in the paper. However, for the sake of illustration, let us assume a 2-way set associative cache. Moreover, assume the following mapping of memory blocks to cache sets: $m0 \mapsto \mathcal{S}_1$, $m1 \mapsto \mathcal{S}_2$, $m2 \mapsto \mathcal{S}_3$, $m3 \mapsto \mathcal{S}_2$, $m4 \mapsto \mathcal{S}_4$, $m5 \mapsto \mathcal{S}_2$, $m6 \mapsto \mathcal{S}_5$. $\mathcal{S}_i$ denotes the different cache sets. Therefore, in our example, only $m1$, $m3$ and $m5$ conflict in the cache. Due to the two different cache conflicts generated from $m3$ and $m5$ along two different control flow paths, AI-based analysis will categorize $m1$ as unclassified (NC).

To refine the categorization of memory block $m1$ via path sensitive program verification, the code is modified as shown in Figure 24. To prove that $m1$ cannot be evicted from the cache, we need to ensure that the number of unique and conflicting memory blocks entered into the cache cannot be more than one, whenever $m1$ is accessed. This is due to the fact that two new and conflicting memory blocks will

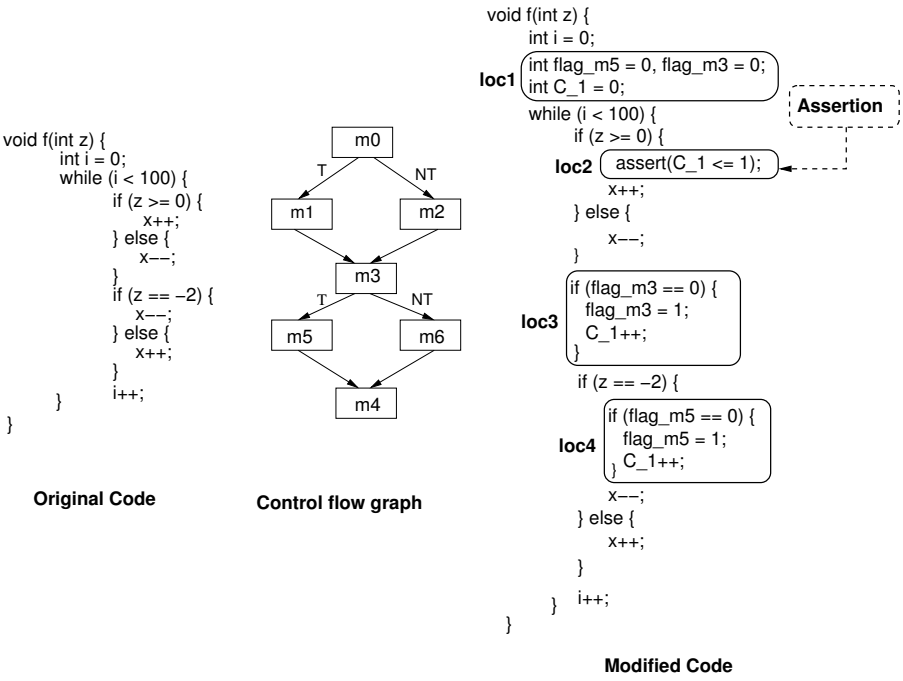


Fig. 24 Intra-task cache conflict refinement for FIFO cache replacement policy

evict the *oldest* memory block $m1$. In Figure 24, variable C_1 serves the purpose of counting the number of cache conflicts to $m1$. Therefore, we check an assertion property $C\_1 \leq 1$ before accessing $m1$. Since $m3$ and $m5$ both conflict with $m1$, cache conflict count (*i.e.* C_1) is incremented before accessing $m3$ and $m5$. $flag\_m3$ and $flag\_m5$ are required to count the unique accesses to memory blocks $m3$ and $m5$, respectively. Note that the crucial difference with LRU replacement policy is made by elimination of the code after accessing $m1$ (see Figure 5). In LRU, each memory block becomes *most recently used* after its access. Therefore, cache conflict count (*i.e.* C_1) was reset after accessing $m1$. However, in FIFO replacement policy, cache state does not change after a cache hit and therefore, the conflict count (*i.e.* C_1) was unchanged after accessing $m1$. Since $m1 - m3 - m5$ is an *infeasible path*, the assertion property at $P1$ can be successfully verified and we can conclude that $m1$ cannot be evicted from the cache. As marked in Figure 5, our general code transformation framework $\langle \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} \rangle$ was configured as follows: $\mathcal{L} = \{m3, m5\}$, $\mathcal{A}$ is the "assertion" clause checking the property $C\_1 \leq 1$, $\mathcal{P}_l = \{loc3, loc4\}$, $\mathcal{P}_c = \{loc2\}$ and $\mathcal{I} = \{loc1\}$.

Caches with non-LRU and non-FIFO replacement policies So far, there have been very few works on cache replacement policies other than LRU. In the preceding, we have shown the instantiation of our framework for FIFO replacement policy. Recent literatures [Grund and Reineke, 2010b] have shown the technical challenges in de-

signing an abstract interpretation based analysis framework for *pseudo LRU* (PLRU) cache replacement policy. In an N-way set associative PLRU cache, $\log_2(N) + 1$ is a tight lower bound on the number of unique memory block accesses to evict a just inserted memory block into the cache [Reineke et al, 2007]. As a result, an N-way set associative PLRU cache can be instantiated as the analysis of a $\log_2(N) + 1$-way set associative LRU cache [Reineke and Grund, 2008]. Due to this correlation between different cache replacement policies, our proposed framework can also be integrated nicely with PLRU cache replacement policy. The only change in our code transformation will be made in designing the assertion property. For an N-way set associative LRU cache, we check an assertion property of the form $C_i \leq N - 1$, as less than N cache conflicts will guarantee non-eviction of a just inserted memory block into LRU cache. For an N-way set associative PLRU cache, such an assertion property can be modified into $C_i \leq \log_2(N) + 1 - 1$ or simply $C_i \leq \log_2(N)$ due to the above mentioned correlation between a PLRU and LRU cache (refer to [Reineke et al, 2007] for a proof of this correlation).

Refinement of inter-task and inter-core cache conflicts So far in this section, we have discussed the refinement of intra-task cache conflicts. It is worthwhile to note that during inter-task cache conflict refinements, we refine the number of memory blocks accessed by the high priority tasks (termed as evicting cache blocks or ECBs), a quantity *independent of cache replacement policy*. Similarly, during inter-core cache conflict refinement, we refine the number of memory blocks accessed by different cores, again a quantity which is independent of any cache replacement policy. Therefore, for a particular cache replacement policy, a CRPD analysis for multi-tasking system or a WCET analysis for multi-core system can always be benefited from our framework by refining the number of inter-task and inter-core cache conflicts, respectively.

8.2 Data caches

Analysis of data caches introduces additional complications, as different instances of the same instruction may access different data memory blocks (*e.g.* array element access inside a loop). Although the basic structure of our refinement process remains exactly the same (*i.e.* Figure 4), the code transformation for the refinement requires minor modifications as explained below.

The code transformation need to account two crucial facts in the presence of data caches. Note that a data access may correspond to multiple memory blocks. Since multiple memory blocks might map to different data cache sets, the code transformation need to record the cache conflicts for all such cache sets. Assume that we want to refine the categorization of a data reference R which may access the cache sets $\{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k\}$. Therefore, we introduce k variables $C_1, C_2, \ldots, C_k$ to count the cache conflicts in cache sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$; respectively. C_i is incremented before a data reference if some memory block accessed by the data reference may access cache set i. Finally, to refine the categorization of R in an N-way set associative LRU data cache, we check an assertion property $C_1 \leq N-1 \wedge C_2 \leq N-1 \wedge \ldots \wedge C_k \leq N-1$. A successful verification of the assertion property leads to the fact that none of the

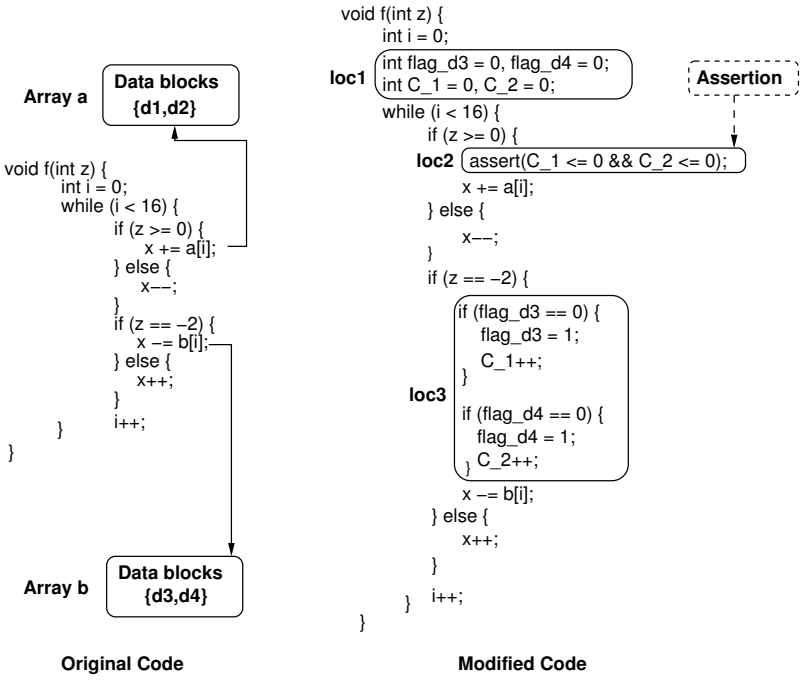


Fig. 25 Intra-task cache conflict refinement for data caches

memory blocks accessed by data reference R could be evicted from the cache. As a result, we can refine the categorization of data reference R.

Figure 25 illustrates an example with a direct mapped data cache. Assume that the array reference a accesses a set of data memory blocks $\{d1, d2\}$ and the array reference b accesses a set of data memory blocks $\{d3, d4\}$. The memory block mapping to the data cache is as follows: $d1 \mapsto \mathcal{S}_1$, $d2 \mapsto \mathcal{S}_2$, $d3 \mapsto \mathcal{S}_1$, $d4 \mapsto \mathcal{S}_2$. $\mathcal{S}_i$ captures the data cache sets. Therefore, in our example, $d1$ and $d3$ conflict in the data cache. Similarly, $d2$ and $d4$ also conflict in the data cache. Additionally, assume that no other data accesses conflict in the data cache with array references a and b. A traditional abstract interpretation based analysis will classify both the array references (*i.e.* a and b) as unclassified. Let us assume that we want to refine the categorization of array reference a. The transformed code is also shown in Figure 25. Since array reference a may access the cache sets $\mathcal{S}_1$ and $\mathcal{S}_2$, we count the cache conflicts both in cache set 1 and 2 (through variables C_1 and C_2, respectively). Such cache conflicts are incremented at array reference b, since the array reference b may access both the cache sets $\mathcal{S}_1$ and $\mathcal{S}_2$. Finally, since array reference a may access both the cache sets $\mathcal{S}_1$ and $\mathcal{S}_2$, we check the assertion property $C\_1 \leq 0 \wedge C\_2 \leq 0$. Due to the infeasible path condition $z \geq 0 \wedge z = -2$, the assertion property will be verified successfully and we can conclude that the memory blocks of array a cannot be evicted from the data cache. A similar successful refinement can also be carried out on array reference b. Note that we can no longer reset the conflict counts (*i.e.* variables C_1 and

C_2) after accessing the array a, as the array reference may correspond to multiple cache sets and we do not know statically which cache set is accessed. As illustrated in Figure 25, our general code transformation framework $\langle \mathcal{L}, \mathcal{A}, \mathcal{P}_l, \mathcal{P}_c, \mathcal{I} \rangle$ can be configured as follows: $\mathcal{L} = \{d3, d4\}$, $\mathcal{A}$ is the "assertion" clause checking the property $C\_1 \leq 0 \wedge C\_2 \leq 0$, $\mathcal{P}_l = \{loc3\}$, $\mathcal{P}_c = \{loc2\}$ and $\mathcal{I} = \{loc1\}$.

Our inter-task and inter-core cache conflict refinement framework remains mostly unchanged. However, to count the data memory blocks by a conflicting task (a high priority task or a task running on a remote core), the code transformation must account the multiple memory blocks accessed by a data reference, as shown by our previous example in Figure 25.

It is worthwhile to note that the accuracy of our refinement phase largely depends on the accuracy of address analysis (*e.g.* [Balakrishnan and Reps, 2004]). Address analysis is used to statically estimate an over-approximation of the set of addresses accessed by a data reference. In the example shown in Figure 25, address analysis computes the set of possibly accessed memory blocks by array references a and b (*i.e.* $\{d1, d2\}$ and $\{d3, d4\}$, respectively). If the set of memory blocks estimated by address analysis is over-approximated, our framework may increase the cache conflict count unnecessarily for the set of over-approximated memory blocks. As a result, an imprecise address analysis may lead to many assertion violations in the transformed code and our framework may not be able to eliminate many *spurious* data cache conflicts. However, we believe that designing precise address analysis is beyond the technical scope of this paper and any precision gain in the underlying address analysis will directly improve the data cache analysis precision using our framework.

9 Conclusion and future work

In this paper, we have proposed two compositional WCET analysis frameworks, one of which combines abstract interpretation with model checking and the second one combines abstract interpretation with constraint solving, both for cache analysis. Our framework does not affect the flexibility of abstract interpretation based cache analysis and it can be composed with the analysis of different other micro-architectural features (*e.g.* pipeline). Moreover, our model checker or symbolic execution guided refinement process is always *safe*. Therefore, the refinement process can be terminated at any point if the time budget is violated. Experimental results show that we can obtain significant improvement for various types of cache analysis in single and multi-cores using both of our compositional analysis frameworks.

Our current work can be extended to improve the state-of-the-art data cache analysis as follows: recall that address analysis computes an over-approximation of accessed memory blocks for a particular data reference. As a result, the precision of data cache analysis may be significantly affected by the overestimation of address analysis. In our proposed approach, such an address analysis may also affect the refinement process by generating many false assertion violations (as mentioned in Section 8.2). A recent work [Huynh et al, 2011] has proposed a new approach to overcome the problem caused by address analysis. Instead of computing the set of memory blocks accessed by each data reference, the work proposed in [Huynh et al, 2011] computes

the set of loop iterations in which a particular memory block is accessed. Such a computation strategy is quite useful for data accesses, as the data memory blocks mapping to the same cache set can never conflict with each other if they are accessed in disjoint loop iteration space. Our code transformation technique can be extended for the type of address analysis proposed in [Huynh et al, 2011]. Such an extension will be helpful for improving the precision of data cache analysis by removing *spurious data cache conflicts*.

In future, we plan to test our framework with real-world applications apart from the programs in [Gustafsson et al, 2010] and we also plan to explore the possibilities of applying such a compositional analysis framework for micro-architectural modeling other than caches in a scalable fashion.

Acknowledgements This work was partially supported by A*STAR Public Sector Funding Project Number 1121202007 - "Scalable Timing Analysis Methods for Embedded Software".

References

aiT (2000) aiT WCET analyzer. URL `http://www.absint.com/ait`

Altmeyer S, Burguière C (2009) A new notion of useful cache block to improve the bounds of cache-related preemption delay. In: ECRTS, pp 109–118

Altmeyer S, Maiza C, Reineke J (2010) Resilience analysis: tightening the CRPD bound for set-associative caches. In: LCTES, pp 153–162

Balakrishnan G, Reps TW (2004) Analyzing memory accesses in x86 executables. In: CC, pp 5–23

Balakrishnan G, Reps TW, Kidd N, Lal A, Lim J, Melski D, Gruian R, Yong SH, Chen CH, Teitelbaum T (2005) Model checking x86 executables with codesurfer/x86 and wpds++. In: CAV, pp 158–163

Cadar C, Dunbar D, Engler DR (2008) KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp 209–224

Chattopadhyay S, Roychoudhury A (2011) Scalable and precise refinement of cache timing analysis via model checking. In: RTSS, pp 193–203

Chattopadhyay S, Roychoudhury A, Mitra T (2010) Modeling shared cache and bus in multi-cores for timing analysis. In: SCOPES, pp 6–15

Chattopadhyay S, Chong LK, Roychoudhury A, Kelter T, Marwedel P, Falk H (2012) A unified wcet analysis framework for multi-core platforms. In: RTAS, pp 99–108

Clarke E, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: TACAS, pp 168–176

Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst 8(2):244–263

Clarke EM, Biere A, Raimi R, Zhu Y (2001) Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1):7–34

Dalsgaard AE, Olesen MC, Toft M, Hansen RR, Larsen KG (2010) METAMOC: Modular execution time analysis using model checking. In: WCET, pp 113–123

Godefroid P, Klarlund N, Sen K (2005) DART: directed automated random testing. In: PLDI, pp 213–223

Grund D, Reineke J (2010a) Precise and efficient FIFO-replacement analysis based on static phase detection. In: ECRTS, pp 155–164

Grund D, Reineke J (2010b) Toward precise PLRU cache analysis. In: WCET, pp 23–35

Gustafsson J, Betts A, Ermedahl A, Lisper B (2010) The Mälardalen WCET benchmarks – past, present and future. In: WCET, pp 137–147, URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`

Hardy D, Piquet T, Puaut I (2009) Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In: RTSS, pp 68–77

Huber B, Schoeberl M (2009) Comparison of implicit path enumeration and model checking based wcet analysis. In: WCET

Huynh BK, Ju L, Roychoudhury A (2011) Scope-aware data cache analysis for WCET estimation. In: RTAS, pp 203–212

Ju L, Huynh BK, Roychoudhury A, Chakraborty S (2008) Performance debugging of esterel specifications. In: CODES+ISSS, pp 173–178

Kelter T, Falk H, Marwedel P, Chattopadhyay S, Roychoudhury A (2011) Bus-aware multicore wcet analysis through tdma offset bounds. In: ECRTS, pp 3–12

King JC (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394

KLEE (2008) The KLEE Symbolic Virtual Machine. URL `http://klee.llvm.org`

Lee CG, Hahn J, Seo YM, Min SL, Ha R, Hong S, Park CY, Lee M, Kim CS (1998) Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. IEEE Trans Comput 47(6):700–713

Li X, Roychoudhury A, Mitra T (2004) Modeling out-of-order processors for software timing analysis. In: RTSS, pp 92–103

Li X, Liang Y, Mitra T, Roychoudhury A (2007) Chronos: A timing analyzer for embedded software. Science of Computer Programming pp 56–67, `http://www.comp.nus.edu.sg/~rpembed/chronos`

Li Y, Suhendra V, Liang Y, Mitra T, Roychoudhury A (2009) Timing analysis of concurrent programs running on shared cache multi-cores. In: RTSS, pp 57–67

Li YTS, Malik S, Wolfe A (1999) Performance estimation of embedded software with instruction cache modeling. ACM Trans Design Autom Electr Syst 4(3):257–279

LLVM (2003) The LLVM compiler infrastructure. URL `http://llvm.org`

Lv M, Yi W, Guan N, Yu G (2010) Combining abstract interpretation with model checking for timing analysis of multicore software. In: RTSS, pp 339–349

Metzner A (2004) Why model checking can improve WCET analysis. In: CAV, pp 334–347

Negi HS, Mitra T, Roychoudhury A (2003) Accurate estimation of cache-related preemption delay. In: CODES+ISSS, pp 201–206

Pellizzoni R, Schranzhofer A, Chen JJ, Caccamo M, Thiele L (2010) Worst case delay analysis for memory interference in multicore systems. In: DATE, pp 741–746

Reineke J, Grund D (2008) Relative competitive analysis of cache replacement policies. In: LCTES, pp 51–60

Reineke J, Grund D, Berg C, Wilhelm R (2007) Timing predictability of cache replacement policies. Real-Time Systems 37(2):99–122

SPIN (1991) SPIN Model Checker. URL http://spinroot.com/spin/whatispin.html

STP (2007) The STP Constraint Solver. URL http://sites.google.com/site/stpfastprover

Suhendra V, Mitra T, Roychoudhury A, Chen T (2006) Efficient detection and exploitation of infeasible paths for software timing analysis. In: DAC, pp 358–363

Tan Y, Mooney V (2007) Timing analysis for preemptive multitasking real-time systems with caches. ACM Trans Embed Comput Syst 6(1)

Theiling H, Ferdinand C, Wilhelm R (2000) Fast and precise WCET prediction by separated cache and path analyses. Real-Time Systems 18(2/3):157–179

Wilhelm R (2004) Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In: VMCAI, pp 309–322

Wilhelm R, Grund D, Reineke J, Schlickling M, Pister M, Ferdinand C (2009) Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans on CAD of Integrated Circuits and Systems 28(7):966–978

Yan J, Zhang W (2008) WCET analysis for multi-core processors with shared L2 instruction caches. In: RTAS, pp 80–89