Precise Micro-architectural Modeling for WCET Analysis via AI+SAT

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

***Abstract*—Hard real-time systems are required to meet critical deadlines. Worst case execution time (WCET) is therefore an important metric for the system level schedulability analysis of hard real-time systems. However, performance enhancing features of a processor (*e.g.* pipeline, caches) makes WCET analysis a very difficult problem. In this paper, we propose a novel approach to combine abstract interpretation (AI) and satisfiability (SAT) checking (hence the name AI+SAT) for different varieties of micro-architectural modeling. Our work in this paper is inspired by the research advances in program flow analysis(*e.g.* infeasible path analysis). We show that the accuracy of WCET estimates can be improved in a scalable fashion by using SAT checkers to integrate infeasible path analysis results into micro-architectural modeling. Our modeling is implemented on top of the Chronos WCET analysis tool and we improve the accuracy of WCET estimates for instruction cache, data cache, branch predictors and shared caches.**

I. Introduction

Real-time embedded software are required to satisfy some extra-functional properties, such as timing. For hard real-time systems, such timing guarantees must be known in advance. An erroneous timing guarantee for hard real-time systems may lead to serious consequences. As a result, worst-case execution time (WCET) analysis has emerged to be one of the critical problems to address for hard real-time systems. Since the execution time is highly sensitive to the underlying hardware platform, WCET analysis typically involves three different phases: micro-architectural modeling (which analyzes the timing effects of underlying micro-architecture), program flow analysis (which computes the infeasible paths and loop bounds of a program) and WCET calculation (which combines the results of micro-architectural modeling and program flow analysis to derive the whole program WCET).

Micro-architectural modeling systematically considers the timing effects of underlying hardware platform (*e.g.* pipeline, caches, branch predictors) and it produces the WCET of each basic block in the program control flow graph (CFG). On the other hand, program flow analysis usually involves finding infeasible program paths in the CFG. Such infeasible program paths are ignored during WCET calculation phase to produce a tighter WCET estimate. This WCET analysis process is shown in Figure 1. A crucial observation from Figure 1 is that the micro-architectural modeling and program flow analysis are performed *independently*. As a result, the information computed by program flow analysis is typically *not used* by the micro-architectural modeling. In the absence of program flow information, micro-architectural modeling involves considering many *infeasible* micro-architectural states, which may lead to the imprecision in WCET estimate for each basic block. A typical example of such infeasible micro-architectural state would be the set of memory blocks inserted into the cache along some infeasible program path. With the help of some program flow information (which is already computed during the program flow analysis), such infeasible micro-architectural states can be ignored, which in turn will lead to a tighter WCET of each basic block after the micro-architectural modeling. As a result, the integration of program flow information into micro-architectural modeling may lead to a tighter WCET estimate of the overall program. The main novelty of our work lies in the consideration of infeasible program paths (computed by program flow analysis) into micro-architectural modeling. Therefore, our work in this paper establishes the missing link (in Figure 1) between program flow analysis and micro-architectural modeling.

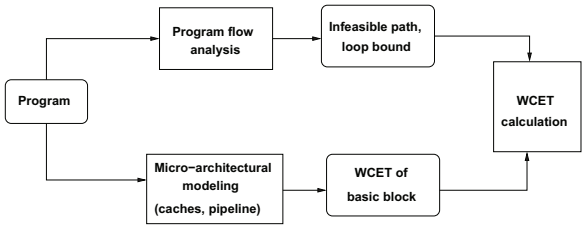


Fig. 1: A typical WCET analysis framework

However, considering program flow information into micro-architectural modeling leads to several technical challenges. A naive strategy is to employ *fully path sensitive* micro-architectural modeling. Such an approach will be infeasible in practice due to the classic state-space explosion problem [24]. Therefore, micro-architectural modeling for real-time systems is usually accomplished by abstract interpretation. Abstract interpretation is usually efficient but often imprecise. This happens due to the "join" of several micro-architectural states at control flow merge points, which may eventually lead to several infeasible micro-architectural states (*e.g.* infeasible cache contents for cache analysis). However, due to this "join" operation, abstract interpretation is *path insensitive*, which in turn leads to its scalability.

In this paper, we propose a generic extension to abstract interpretation (AI) based analysis framework using satisfiability (SAT) checking. Our baseline analysis is abstract interpreta-

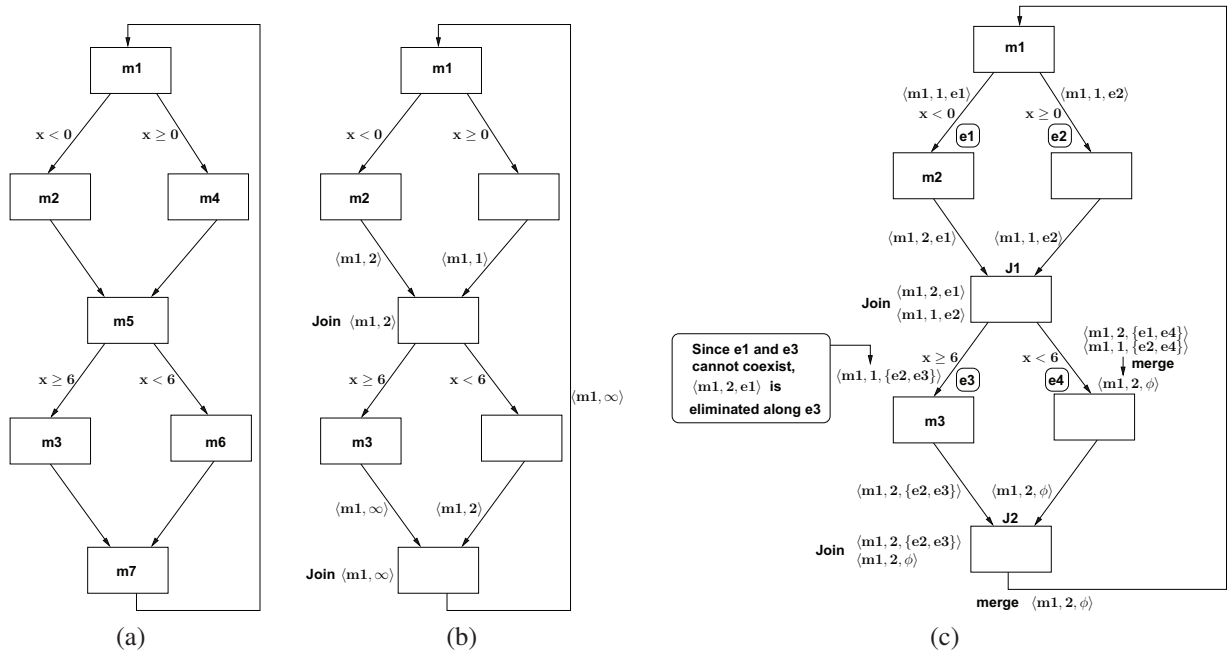


Fig. 2: Illustrative example (a) program control flow graph with accessed memory blocks shown inside each basic block. The branch conditions are shown beside the respective control flow edges, (b) original must cache analysis, (c) must cache analysis instantiated by our proposed framework

tion. At any program point, our proposed framework tracks a *partial path* with each micro-architectural state μ. This partial path captures a subset of all the control flow edges along which the micro-architectural state μ has been propagated. We define the partial path as a propositional logic formula φ over the propositions associated with each control flow edge. At each control flow branch (*i.e.* conditional statements), this partial path formula φ is sent to an *oracle*. The *oracle*, in turn, is generated after program flow analysis and checks the *infeasibility* of the partial path (defined by φ). Such a checking can be accomplished by making an *on-the-fly* call to a satisfiability solver (*e.g.* using Minisat [2]). If the partial path was infeasible, its associated micro-architectural state can be ignored for further consideration. Due to a significant progress in SAT solver technologies for the past few decades, such calls to SAT solvers can be processed very efficiently.

The set of micro-architectural states generated by our framework is always *tractable*. To control the number of micro-architectural states, we employ strategies to merge different micro-architectural states at appropriate program points. The growth in the number of micro-architectural states (compared to the original abstract interpretation based framework) is always bounded by a *magnitude* equal to the incoming degree of a control flow node, typically a small number. Therefore, we provide a comprehensive and tractable strategy to integrate program flow analysis into micro-architectural modeling.

Contributions: In summary, this paper presents a generic micro-architectural modeling framework using abstract interpretation and satisfiability (SAT) solvers. Such a framework leverages the progress in program flow analysis research for precise micro-architectural modeling. We show that our framework can be instantiated for the modeling of several micro-architectural entities – i) instruction caches, ii) data caches, iii) branch target buffers, and iv) shared instruction caches. We have implemented our framework using Chronos [14], an open source, freely available WCET analysis tool and Minisat [2], an open source, freely available satisfiability (SAT) solver. Our experiments with several open source subject programs suggest substantial improvements in the accuracy of WCET analysis. This makes the idea of integrating program flow analysis into micro-architectural modeling quite appealing for research in future.

II. Overview

In this section, we shall illustrate the central idea behind our approach through a simple example. Through the example, we shall show how the precision of cache analysis can be improved using our proposed framework.

Figure 2(a) shows the control flow graph (CFG) of a program. The label inside each basic block captures the memory blocks accessed by the same basic block. The branch condition is shown beside each conditional branches. For the sake of illustration, let us assume that variable x (used in the conditional branches) is not modified anywhere in the CFG. Additionally, we assume a 2-way set associative cache, where the memory blocks in the CFG are mapped to different cache sets as follows: $m1 \mapsto \mathcal{S}_1$, $m2 \mapsto \mathcal{S}_1$, $m3 \mapsto \mathcal{S}_1$, $m4 \mapsto \mathcal{S}_2$, $m5 \mapsto \mathcal{S}_3$, $m6 \mapsto \mathcal{S}_4$ and $m7 \mapsto \mathcal{S}_7$. $\mathcal{S}_i$ captures the different cache sets. Therefore, in our example, only the memory blocks $m1, m2$ and $m3$ conflict in the cache.

Figure 2(b) shows the state-of-the-art must cache analysis [23] for LRU replacement policy. Each element in the cache

state is represented as $\langle m, a\rangle$, where m is the memory block with LRU age a. Let us see the propagation of abstract cache states associated with memory block $m1$. Since $m2$ conflicts with $m1$, the *join* operation at the first control flow merge point computes $\langle m1, 2\rangle$ (by taking the must join of $\langle m1, 2\rangle$ and $\langle m1, 1\rangle$). Since $m3$ also conflicts with $m1$, the control flow after accessing $m3$ will evict $m1$ from the abstract cache state (captured by the element $\langle m1, \infty\rangle$ in Figure 2(b)). As a result, the *join* operation at the second control flow merge computes $\langle m1, \infty\rangle$. Therefore, must analysis cannot conclude any subsequent accesses to $m1$ as *cache hits*.

However, careful examination reveals that $x \leq 0$ and $x \geq 6$ cannot be satisfied for any execution (recall that we assume x is not modified anywhere in the CFG). The must cache analysis was unaware of this infeasible execution. As a result, the traditional must cache analysis assumes two cache conflicts to $m1$ (from $m2$ and $m3$), whereas at most one cache conflict is possible for *any feasible execution*. To summarize, in the absence of any program flow information, abstract interpretation based cache analysis cannot determine that accesses to memory block $m1$ are cache hits (excluding the cold cache miss).

To resolve the gap between program flow analysis and micro-architectural modeling, we propose to extend the abstract domain of the micro-architectural state with partial path information. The instantiation of our proposed framework for cache analysis is shown in Figure 2(c). We first label the control flow edges (as shown by $e1, e2, e3$ and $e4$ in Figure 2(c)) and define a predicate associated with each such labeling. Let us assume $pred_e$ captures the predicate associated with label e. $pred_e$ is *true* if and only if control flow edge e is executed.

Program flow analysis can produce infeasible branch pairs in a program (*e.g.* using [12], [22]). For our example, the infeasible branch pairs (*i.e.* $x < 0$ and $x \geq 6$) are captured by the following formula:

$$\Psi_{flow} \equiv \neg pred_{e1} \vee \neg pred_{e3} \tag{1}$$

With the augmented abstract domain, our analysis now labels the cache states with control flow information. Such a labeling enables us to distinguish the control flow along which a single cache state is propagated. As an example, in the beginning, $\langle m1, 1, e1\rangle$ and $\langle m1, 1, e2\rangle$ are propagated along control flow edges $e1$ and $e2$, respectively.

The *join* operation at first control flow merge point keeps both the elements ($\langle m1, 2, e1\rangle$ and $\langle m1, 1, e2\rangle$) in the abstract cache state since they come along different control flows $e1$ and $e2$. The crucial difference is, however, made at the branch point $x \geq 6$. Let us assume that we want to propagate the cache state produced after the first join operation along the control flow labelled $e3$. While propagating a cache state along a branch edge, our framework checks the feasibility of the cache state along the same branch. Therefore, when we try to propagate $\langle m1, 2, e1\rangle$ along $e3$, we check the feasibility of the state along $e3$ by consulting the information generated by program flow analysis (*i.e.* Ψ_{flow}). We make a call to the SAT solver to check the feasibility of the following formula:

$$\begin{aligned} \varphi &\equiv \Psi_{flow} \wedge pred_{e1} \wedge pred_{e3} \\ &\equiv (\neg pred_{e1} \vee \neg pred_{e3}) \wedge pred_{e1} \wedge pred_{e3} \end{aligned} \tag{2}$$

This is due to the fact that the propagation of $\langle m1, 2, e1\rangle$ along $e3$ must execute both the control flow edges $e1$ and $e3$, which in turn means that $pred_{e1} \wedge pred_{e3}$ must be *true*. Since the formula in Equation 2 is *unsatisfiable*, we do not propagate the cache state $\langle m1, 2, e1\rangle$ along $e3$. On the other hand, since $\Psi_{flow} \wedge pred_{e2} \wedge pred_{e3}$ is satisfiable, $\langle m1, 1, e2\rangle$ is propagated along $e3$. Additionally, such a propagation of cache state $\langle m1, 1, e2\rangle$ along $e3$ updates the control flow information of the cache state from $e2$ to $\{e2, e3\}$ (as shown by the element $\langle m1, 1, \{e2, e3\}\rangle$ in Figure 2(c)). $\langle m1, 1, \{e2, e3\}\rangle$ now captures that the original cache state $\langle m1, 1\rangle$ has been propagated through control flow edges $\{e2, e3\}$.

Now let us consider the branch edge labelled $e4$. We found that both the formula $\Psi_{flow} \wedge pred_{e1} \wedge pred_{e4}$ and $\Psi_{flow} \wedge pred_{e2} \wedge pred_{e4}$ are *satisfiable*. Therefore, both the cache states $\langle m1, 2, e1\rangle$ and $\langle m1, 1, e2\rangle$ are propagated along $e4$ and the control flow information of $\langle m1, 2, e1\rangle$ and $\langle m1, 1, e2\rangle$ are updated to $\{e1, e4\}$ and $\{e2, e4\}$, respectively (as shown by $\langle m1, 2, \{e1, e4\}\rangle$ and $\langle m1, 1, \{e2, e4\}\rangle$ in Figure 2(c)).

To control the number of cache states containing $m1$, we perform a merge operation at control flow edge $e4$. The merge operation for must cache analysis takes the *maximum* age of a memory block and loses the entire control flow information. As a result, after merging $\langle m1, 2, \{e1, e4\}\rangle$ and $\langle m1, 1, \{e2, e4\}\rangle$ for must cache analysis, we get $\langle m1, 2, \phi\rangle$. Due to this merge operation, we can always control the number of micro-architectural states in our framework.

It is worthwhile to note the difference between merge and join operation in our framework. For the time being assume that we had performed the *merge* operation between $\langle m1, 2, \{e1\}\rangle$ and $\langle m1, 1, \{e2\}\rangle$ at the first control flow join point (*i.e.* at $J1$). As a result, we had obtained a cache state $\langle m1, 2, \phi\rangle$ after the first control flow join. If we try to propagate the cache state $\langle m1, 2, \phi\rangle$ along $e3$, we check the satisfiability of a formula $\Psi_{flow} \wedge pred_{e3}$, which is clearly *satisfiable*. Consequently, we had not eliminated any cache state along $e3$ and all subsequent accesses to $m1$ would not be classified as *cache hits*.

Continuing in a similar sequence of join and merge operation, we obtain the cache state $\langle m1, 2, \phi\rangle$ propagated along the backedge. As a result, all subsequent accesses of $m1$ can be categorized as *cache hits*. Note that the key difference in our framework was made by removing the infeasible cache state $\langle m1, 2\rangle$ (in the traditional must cache analysis framework) along control flow edge $e3$. This infeasible cache state was detected using the infeasible path information computed by program flow analysis (*i.e.* Ψ_{flow}) and augmenting the abstract interpretation framework.

III. General framework

In this section, we shall introduce the general idea behind our analysis framework. We shall show how an abstract

interpretation based analysis framework can be augmented with the help of a satisfiability solver to generate more precise analysis outcome.

A. Program flow analysis

Our proposed framework uses program flow analysis to rule out certain infeasible micro-architectural states. For program flow analysis, we currently look at finding the *infeasible branch pairs*. Assume two branch conditions $x \leq 0$ and $x \geq 2$ in a program. If x is not modified, both $x \leq 0$ and $x \geq 2$ cannot be *true* for any execution. As a result, the control flow edges associated with the *true* evaluations of $x \leq 0$ and $x \geq 2$ constitute an infeasible branch pair. Such infeasible branch pairs can be computed automatically (such as using [12], [22]) or they can be provided manually by the user.

In the past few decades, satisfiability (SAT) solver technology has made significant progress. An interesting feature about infeasible branch pairs is that they can easily be encoded as propositional logic formula. Let us introduce an atomic proposition $pred_e$ associated with each control flow edge e in the program. $pred_e$ captures the execution of control flow edge e. $pred_e$ is *true* if control flow edge e is executed and *false* otherwise. Therefore, an infeasible branch pair $\langle b_i, b_j \rangle$ can be encoded as the following propositional formula:

$$\varphi \equiv \neg pred_{b_i} \vee \neg pred_{b_j} \tag{3}$$

At the end of program flow analysis, therefore, we have a set of clauses (as shown in Equation 3) in *conjunctive normal form* (CNF). Let us assume Ψ_{flow} represents this CNF formula. Therefore, Ψ_{flow} captures certain infeasible path patterns (specifically infeasible branch pairs) in a program.

Given the information computed by program flow analysis (*i.e.* Ψ_{flow}), we define an oracle Θ on any propositional formula η as follows:

$$\Theta(\eta) = \begin{cases} true, & \text{if } \Psi_{flow} \wedge \eta \text{ is satisfiable;} \\ false, & \text{otherwise} \end{cases} \tag{4}$$

Note that any satisfiability solver (such as Minisat [2]) can be used as the oracle Θ. Θ will be used to eliminate infeasible micro-architectural states.

B. Augmenting abstract interpretation

The key idea behind our analysis framework is to inject the notion of path sensitivity inside abstract interpretation. A fully path sensitive approach is definitely not scalable, as it leads to a path explosion. Therefore, we augment the abstract interpretation in a fashion such that path explosion never occur and the state space of the analysis can always be controlled. In the following, we shall briefly describe the key components of the analysis.

Changing the abstract domain: Assume an abstract interpretation based analysis framework with abstract domain $\mathbb{D}$. To handle partial path sensitivity, our proposed analysis framework augments this abstract domain $\mathbb{D}$ as follows:

$$\mathbb{D}' : \mathbb{D} \times \mathcal{P}(\mathbb{E}) \tag{5}$$

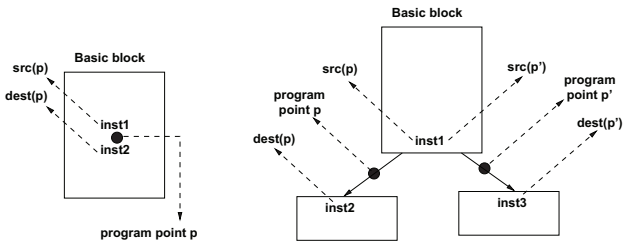


Fig. 3: (a) program point inside a basic block, (b) program point at a branch location

In the above equation, $\mathbb{E}$ captures the set of all control flow edges in the program and $\mathcal{P}(\mathbb{E})$ represents the set of all subsets of $\mathbb{E}$. Therefore, at any specific program point p, an element from the changed abstract domain $\mathbb{D}'$ is of the form $\langle d, \{e_1, e_2, \ldots, e_k\} \rangle$ where $d \in \mathbb{D}$ and $\{e_1, e_2, \ldots, e_k\}$ captures the set of control flow edges along which d has been propagated to p.

Transfer and join function: Before going into the details of *transfer* and *join* function, we first briefly describe the notion of *program points* in our analysis framework. We define the program point as the control flow between two instructions. We distinguish the two different types of program points in our framework as shown in Figure 3. Figure 3(a) captures the control flow inside a basic block of the program CFG. On the other hand, Figure 3(b) captures the control flow between two different basic blocks. For a specific program point p, we shall use $src(p)$ to denote the *source* of the control flow captured by p, whereas $dest(p)$ will be used to denote the destination of the control flow captured by p.

Transfer function of our proposed analysis framework is applied at each program point. However, our proposed transfer function has two key components, depending on the location of the program point. More precisely, our proposed transfer function has the following semantics ($\mathbb{P}$ denotes the set of all program points):

$$\tau : \mathbb{D}' \times \mathbb{P} \rightarrow \mathbb{D}'$$

$$\tau(d', p) = \begin{cases} \tau_{br} \bullet \tau_{in}(d', p), & \text{if } src(p) \text{ is at the end} \\ & \text{of a basic block;} \\ \tau_{in}(d', p), & \text{otherwise.} \end{cases} \tag{6}$$

$\bullet$ denotes the function composition. In Equation 6, τ_{br} captures the transfer function used at the control flow across basic blocks (Figure 3(b)) and τ_{in} captures the transfer function used at the control flow inside a basic block (Figure 3(a)).

The computation of τ_{in} largely depends on the type of analysis, as it requires updating the abstract state by considering each instruction. τ_{br}, on the other hand, is the key to our proposed framework and it is used to eliminate the *spurious* (*i.e.* infeasible) abstract states. Assume that e_p captures the control flow edge associated with program point p when $src(p)$ is at the end of a basic block. Further assume $d' \in \mathbb{D}'$ and $d' = \langle d, E \rangle$ (where $E \subseteq \mathbb{E}$ and $\mathbb{E}$ is the set of all control flow edges).

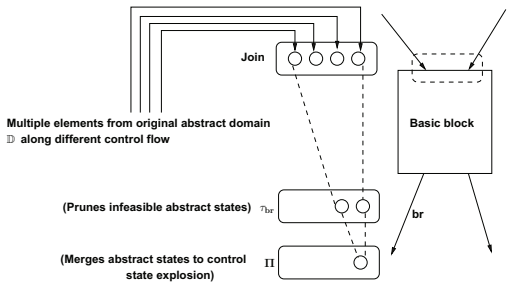


Fig. 4: A schematic representation of the *join*, τ_{br} and merge (Π) used in our proposed analysis framework

We define $\tau_{br}(d', p)$ as follows:

$$\tau_{br}(d', p) = \begin{cases} \phi, & \text{if } \Theta(\bigwedge_{e \in E} pred_e \wedge pred_{e_p}) \text{ is } false \\ \langle d, E \cup \{e_p\} \rangle, & \text{otherwise.} \end{cases} \tag{7}$$

Recall that $pred_e$ denotes the atomic proposition which evaluates to *true* if and only if control flow edge e is executed. Θ is the oracle (as described in Equation 4) used to check the feasibility of control flow $E \cup \{e_p\}$. Equation 7 serves two purposes: first, to eliminate *spurious* abstract states (captured by the first case in Equation 7) and secondly, to associate the notion of path sensitivity with abstract states (captured by the second case in Equation 7).

Join operation with our augmented abstract domain $\mathbb{D}'$ operates in a similar fashion as with the *join* operation with original abstract domain $\mathbb{D}$, with one crucial difference. After the *join* operation is performed with $\mathbb{D}'$, a single element from the original abstract domain $\mathbb{D}$ may have multiple instances in the *joined* abstract state. These multiple instances may appear due to the propagation of a single element in $\mathbb{D}$ along different control flow paths. As a result, a single element from the original abstract domain $\mathbb{D}$ might be associated with different subsets of control flow edges, leading to different elements in the augmented abstract domain $\mathbb{D}'$ in the joined abstract state.

Controlling the number of abstract states: The *join* operation in our proposed framework may enlarge the number of abstract states compared to the original abstract interpretation based framework. As a result, performing the join operation in an uncontrolled fashion may lead to state explosion. Therefore, our proposed framework controls the number of elements in the abstract state at each control flow edges (*i.e.* after performing the τ_{br} operation). This is done by pruning the number of tractable elements with a special operation Π. If there is a pair of elements $d'_1, d'_2 \in \mathbb{D}'$ such that $d'_1 = \langle d, E_1 \rangle$ and $d'_2 = \langle d, E_2 \rangle$, they are merged to a single element $\langle d, \phi \rangle$ using the operation Π. In an abstract state, Π is applied until there is *at most* one element in the abstract state ($\in \mathbb{D}'$) for each element in the original abstract domain $\mathbb{D}$.

Figure 4 shows a schematic view of our overall framework. The *join* operation used by our framework may increase the number of elements in the abstract state, due to the presence of different control flows. τ_{br} operation at a branch location may prune some of the infeasible elements in an abstract state as also shown in Figure 4. Finally, after merge operation (Π), the size of the abstract state is controlled, which in turn lead to a tractable analysis framework.

Note that we apply the merge operation at each control flow edge. Therefore, the expansion in the number of elements after the *join* operation is bounded by a magnitude equal to the incoming degree of any basic block, typically a small number.

C. Instruction cache analysis via AI+SAT

In this section, we shall instantiate our proposed framework for instruction cache analysis. Abstract interpretation based instruction cache analysis was initiated in [23]. Each instruction is categorized as *all-hit* (AH), *all-miss* (AM) or *not-classified* (NC). An instruction is categorized as AH, if it is found in the cache whenever it is accessed, whereas an instruction is categorized AM, if it is never in the cache whenever it is accessed. If an instruction cannot be categorized as either AH or AM, it remains *not-classified* (NC). For such a categorization of different instructions, usually two different analyses are employed: *must* and *may*. *Must* cache analysis is used for statically predicting a *sound* under-estimation of cache content at each program point. As a result, must cache analysis can be used for AH categorization of different instructions. On the other hand, *may* cache analysis is used for statically predicting a *sound* over-estimation of cache content at each program point. Consequently, we can use *may* cache analysis for AM categorization of instructions.

Modifying abstract domain: The abstract domain of the original instruction cache analysis can be defined as a cross product of two different abstract domains as follows:

$$\mathbb{D} : \mathcal{P}(\mathbb{M} \times \mathbb{A}) \tag{8}$$

with

$$\mathbb{A} = \{1, \ldots, K\} \cup \{\infty\} \tag{9}$$

where $\mathbb{M}$ denotes the set of all memory blocks and $\mathbb{A}$ denotes the set of all possible *ages* of a memory block inside a K-way set-associative cache. A tuple of the form $\langle m, \infty \rangle \in \mathbb{D}$ captures that m does not reside in the cache.

In our proposed framework we augment the original abstract domain $\mathbb{D}$ to $\mathbb{D}'$ as follows:

$$\mathbb{D}' : \mathcal{P}(\mathbb{M} \times \mathbb{A} \times \mathcal{P}(\mathbb{E})) \tag{10}$$

where $\mathbb{E}$ is the set of all control flow edges. Therefore, an entity in the abstract domain $\mathbb{D}'$ is a triplet $\langle m, a, E \rangle$.

Modifying transfer and join function: The modified transfer function can now be described as follows:

$$\tau : \mathbb{D}' \times \mathbb{P} \rightarrow \mathbb{D}'$$

$$\tau(\langle m, a, E \rangle, p) = \begin{cases} \tau_{br} \bullet \tau_{in}(\langle m, a, E \rangle, p), & \text{if } src(p) \text{ is at} \\ & \text{the end of a basic block;} \\ \tau_{in}(\langle m, a, E \rangle, p), & \text{otherwise} \end{cases} \tag{11}$$

$\bullet$ denotes the function composition and the transfer function τ now has two different controls depending on the location of program point p. If we assume that m_p denotes the memory block accessed at $src(p)$, τ_{in} can be defined as follows:

$$\tau_{in}(\langle m,a,E\rangle,p) = \begin{cases} \langle \mathcal{U}_{repl}(\langle m,a\rangle,p),\phi\rangle, & \text{if } m_p = m \\ \langle \mathcal{U}_{repl}(\langle m,a\rangle,p),E\rangle, & \text{otherwise} \end{cases} \quad (12)$$

In Equation 12, $\mathcal{U}_{repl}$ captures the instruction cache update operation (in the original abstract domain $\mathbb{D}$) for a particular cache replacement policy $repl$. Note that our framework is not restricted to a particular replacement policy employed by the instruction cache.

τ_{br} is applied at a branch point. Some of the triplets present in the input abstract cache state may not be feasible along some branches. Therefore, while performing the transfer operation along a control flow edge, only the feasible triplets are transferred to the output abstract cache state. Assume that e_p captures the control flow edge associated with program point p. We define τ_{br} for instruction cache analysis as follows:

$$\tau_{br}(\langle m,a,E\rangle,p) = \begin{cases} \emptyset, \text{if } \Theta(\bigwedge_{e\in E} pred_e \wedge pred_{e_p}) \text{ is } false \\ \langle m,a,E\cup\{e_p\}\rangle, \text{otherwise} \end{cases} \quad (13)$$

Recall that Θ is the oracle (as described in Equation 4) used to check the feasibility of control flow $E \cup \{e_p\}$.

While performing a Join operation, a memory block m, may appear along different control flows. Moreover, m might have different *ages* in the cache along different control flows. In our proposed framework, we do not lose the information about the different ages of the same memory block along different control flows. Therefore, in our proposed framework, we define the *must* and *may* join operations as follows:

$$\bigsqcup_{Must}, \bigsqcup_{May} : \mathbb{D}' \times \mathbb{D}' \rightarrow \mathbb{D}' \quad (14)$$

$$\bigsqcup_{Must}(D_1,D_2) = \{\langle m,a_1,E_1\rangle \in D_1 \mid \exists a_2,E_2 : \langle m,a_2,E_2\rangle \in D_2\} \cup \{\langle m,a_2,E_2\rangle \in D_2 \mid \exists a_1,E_1 : \langle m,a_1,E_1\rangle \in D_1\} \quad (15)$$

$$\bigsqcup_{May}(D_1,D_2) = D_1 \cup D_2 \quad (16)$$

May join operation simply takes the union of two abstract cache states, on the other hand, must join operation takes also the union operation, but restricted to the memory blocks which are present in both the abstract cache states (D_1 and D_2). It is important to note that the different *ages* ($\in \mathbb{A}$) of a memory block (along different control flows) will be retained after Equations 15-16. Changes in the *age* of a memory block in a cache set are handled in merge operation (discussed next).

Merging abstract states: We shall now show how we control the number of abstract cache states at each control flow edges. Note that the join operation in our proposed framework leads to more elements in the abstract cache state as compared to the original analysis proposed in [23]. However, since we prune the number of abstract cache states at each control flow edges, the expansion in the number of abstract cache states is still bounded.

Assume that we obtain an abstract cache state $D \in \mathbb{D}'$ after performing τ_{br} at a branch location. The output of merge operation depends on the type of cache analysis (*i.e. must* or *may* cache analysis). Assume that Π_{must} (Π_{may}) denotes the merge operation applied for *must* (*may*) cache analysis. The main purpose of the merge operation is to control the number of cache states and therefore, after each merge operation, we ensure that the resulting abstract cache state contain *at most one* element for each memory block. Consider two elements $\langle m,a_1,E_1\rangle, \langle m,a_2,E_2\rangle \in D$. The output of Π_{must} and Π_{may} will be as follows:

$$\Pi_{must}(\langle m,a_1,E_1\rangle,\langle m,a_2,E_2\rangle) = \langle m,\max(a_1,a_2),\phi\rangle \quad (17)$$

$$\Pi_{may}(\langle m,a_1,E_1\rangle,\langle m,a_2,E_2\rangle) = \langle m,\min(a_1,a_2),\phi\rangle \quad (18)$$

Therefore, after performing Π_{must}, we retain the *maximum age* of each memory block m. On the other hand, after performing Π_{may}, we retain the *minimum age* of each memory block m. Since the merge operation is performed at each control flow edge, we can now state the following property for our proposed framework:

Property 3.1: Let us assume $\mathbb{B}$ denotes the set of all basic blocks and $deg_{in}(B)$ denotes the incoming degree of a basic block B in the CFG. For a K-way set-associative cache, there could be at most $|K| \times \max_{B\in\mathbb{B}} deg_{in}(B)$ number of elements for any memory block in any abstract cache state.

Since both $deg_{in}(B)$ and K are typically small, we can easily control the number of abstract cache states.

It is worthwhile to mention that the necessary pruning of abstract cache states has already been performed during the computation of τ_{br} (Equation 13). Therefore, by merging the cache states using Π_{must} and Π_{may} we are not entirely losing the partial path sensitivity used by our framework. On the other hand, merging of abstract cache states leads our computation to be tractable in practice.

Since our proposed framework is built up on the basic abstract interpretation (AI) approach, it is guaranteed to give at least as precise cache analysis as the basic AI approach [23]. The main purpose of our proposed framework is to integrate the program flow information into cache analysis. With little or absence of any program flow information, our framework will give exactly same result as in [23].

IV. Extension

A. Data Cache Analysis

In this section, we describe the extension of our framework for data cache analysis. We use the *scope-aware persistent* (SCP) analysis [18] as the baseline analysis for data caches. SCP analysis was selected because the WCET estimates generated by this method are *safe* and more accurate than other existing methods. In SCP analysis, each memory block m is associated with a temporal scope. For a particular memory block m, its temporal scope denotes the set of loop iterations where m might be accessed. If two different memory blocks map to the same cache set but they have disjoint temporal scopes, they cannot conflict in the cache. SCP analysis categorizes data blocks as persistent or non-persistent, on the basis of their temporal scopes. Although the SCP analysis is more

accurate than other abstract interpretation based methods, the WCET estimated by the method can still be over-estimated. This might happen due to the presence of infeasible paths in the program CFG. By extending the SCP analysis using our framework, we can remove such over-estimation and thereby produce a more accurate WCET.

Since the SCP analysis is based on temporal scopes, the transfer functions (*i.e.* data cache update operations) are defined with respect to each program loop. For a given loop level L, the transfer function of our proposed framework is similar to Equation 11 and it can be described as follows:

$$\tau(\langle m,a,E\rangle,p,L) = \begin{cases} \tau_{br} \bullet \tau_{in}(\langle m,a,E\rangle,p,L), \\ \quad src(p) \text{ is at the end of a basic block;} \\ \\ \tau_{in}(\langle m,a,E\rangle,p,L), \text{otherwise} \end{cases} \tag{19}$$

Note that $\bullet$ denotes function composition. τ_{br} for the extended SCP analysis operates in a similar fashion as in the instruction cache analysis (see Equation 13). However, τ_{in} for the SCP analysis is slightly different from the τ_{in} described in Equation 12. This is due to the fact that SCP analysis is scope-sensitive (unlike the instruction cache analysis). Let us assume $\mathbb{M}_p$ denotes the set of memory blocks accessed at a program point p. For a given loop level L, τ_{in} can be defined as follows: $\tau_{in}(\langle m,a,E\rangle,p,L) =$

$$\begin{cases} \langle m,a,E\rangle, \text{ if } \forall m_i \in \mathbb{M}_p.\ (\psi(m_i) \cap \psi(m) = \phi \vee \\ \qquad\qquad \pi(m_i) \neq \pi(m)) \\ \langle \mathcal{UD}(\langle m,a\rangle,p,L),E\rangle, \text{ if } \exists m_i \in \mathbb{M}_p.\ (\psi(m_i) \cap \psi(m) \neq \phi \\ \qquad\qquad \wedge \pi(m_i) = \pi(m)) \wedge m \notin \mathbb{M}_p \\ \langle \mathcal{UD}(\langle m,a\rangle,p,L),\phi\rangle, \quad \text{otherwise} \end{cases} \tag{20}$$

$\psi(m)$ denotes the set of iterations in loop L where m might be accessed (*i.e.* temporal scope of m) and $\pi(m)$ captures the cache set in which memory block m is mapped. The first case (in Equation 20) captures the scenario when none of the memory blocks in $\mathbb{M}_p$ conflict with m in the data cache (either due to disjoint temporal scopes or due to the mapping in disjoint cache sets). If some memory block in $\mathbb{M}_p$ conflicts with m in the data cache, the scope-aware data cache update operation $\mathcal{UD}$ (used in [18]) is used to update the data cache state. Since data cache replacement policy may only change $\mathcal{UD}$ in Equation 20, our proposed framework remains unchanged for different data cache replacement policies.

The SCP join function as defined in [18], can be modified in the same fashion as we do for the join operation in the instruction cache analysis (see Equations 15-16).

B. Branch Target Buffer Analysis

Branch Target Buffer (BTB) is used to predict the target address of a branch instruction, before the target address is actually computed. Since the computation of a target address has some associated penalty, correct prediction of a target address from BTB may greatly improve the program execution time. As a result, an improved BTB analysis may lead to a more accurate WCET prediction. Previous study (*e.g.* [8]) has shown the importance of BTB analysis for static WCET prediction. The work of [8] uses an abstract interpretation based framework for statically analyzing the BTB content and categorizing the branch instructions on the basis of BTB states. As with any other abstract interpretation based approach, the analysis proposed in [8] is also path in-sensitive and it suffers due to the estimation of infeasible BTB states.

We extend the framework for BTB analysis as proposed in [8] with our approach. The new domain of the BTB analysis can be formulated as follows:

$$\mathbb{D}' : \mathcal{P}(\mathbb{BR} \times \mathbb{A} \times \mathcal{P}(\mathbb{E})) \tag{21}$$

where $\mathbb{BR}$ is the set of all branch instructions. All other parameters in the equation have the same interpretation as in equation (10). The join and the transfer functions can be modified in exactly the same fashion as they were modified for the instruction cache analysis (see Equations 11-16).

C. Shared instruction cache analysis

Finally we show how our framework can be extended for analyzing multi-core systems with shared instruction caches. We use our previous work [15] as a baseline for the augmented abstract interpretation framework. Let us assume a multi-core system where each core has a private L1 cache and all the cores share an L2 cache. The work in [15] first analyses both the L1 and L2 cache using [23], [17] ignoring the inter-core cache conflicts and finally, it recategorizes the memory blocks in L2 cache by taking into account the shared cache conflicts.

Assume a program P_1 running on Core 1 and assume that P_1 accesses a memory block m_{p1}. Further assume m_{p1} is categorized as AH in the shared L2 cache without considering inter-core cache conflicts. After inter-core cache conflict analysis, the AH categorization of memory block m_{p1} is changed to NC if and only if the following condition holds: $k - age_{singlecore}(m_{p1}) \leq X$, where k is the associativity of the shared L2 cache, $age_{singlecore}(m_{p1})$ captures the *age* of memory block m_{p1} in the shared L2 cache set before inter-core conflict analysis and X is the amount of shared cache conflicts generated by cores other than Core 1.

If we use augmented abstract interpretation instead of basic abstract interpretation, we can get a precise cache hit-miss categorization for L1 cache. A precise cache hit-miss categorization of memory blocks in L1 cache leads to a lesser number of memory blocks accessing the shared L2 cache. As a result, the amount of shared cache conflicts (*i.e.* X) may reduce. Moreover, since our proposed framework may generate a better must cache analysis (see Section III-C), it may also reduce the quantity $age_{singlecore}(m_{p1})$. Both of these developments in turn reduce the number of memory blocks of P_1 which need to be re-categorized to NC in the shared L2 cache. As a result, we may increase the accuracy of WCET estimates even in the presence of shared caches.

V. Evaluation

Experimental Set-up: The experiments were performed using the timing analyser Chronos [14]. It uses a 5-stage

pipeline with in-order execution, when generating the WCET estimates for our experiments. Chronos uses the abstract interpretation based framework proposed in [23], to analyze instruction caches. This serves as the baseline for our instruction cache analysis. The baseline for our data cache analysis framework was published in [18]. The BTB analysis is implemented into Chronos using the abstract interpretation based framework proposed in [8]. The framework proposed in [15] serves as a baseline analysis for the multi-core, shared instruction cache analysis. To check the satisfiability of a given partial path φ, we use the open-source SAT solver Minisat [2].

Subject programs	Description	Code size	
		Bytes[1]	LOC
nsichneu	Simulates an extended Petri Net. Auto-generated code with many if-statements [4]	63720	4253
Papabench	Auto-navigation utility from an Unmanned Aerial Vehicles (UAV) controller [20]	16920	1097
Jetbench	Single-thread *getThermo* utility from a real-time jet engine performance calculator [21]	6984	315
Communication manager	Auto-generated code from the Rhapsody [3] model of CTAS weather manager [5]	4248	273

TABLE I: Program Set I

Table I shows the subject programs for our experiments. Note that the prime motivation behind our work is to reduce the over-approximation in WCET estimation, due to the presence of infeasible paths. Infeasible paths can often be found in the programs, auto-generated from high level modeling languages. Although, it is not uncommon to have a few infeasible paths in manually written programs as well. Therefore, we choose a combination of auto-generated and manually written programs for our experiments.

All of our experiments were performed on a machine having an Intel Core-i5 processor with 4 GB RAM and running Ubuntu 9.04 OS. For all our experimental results, we measure the WCET improvement as $\frac{WCET_{base}-WCET_{augmented}}{WCET_{base}} \times 100\%$, where $WCET_{augmented}$ captures the WCET obtained using our approach and $WCET_{base}$ captures the WCET obtained using the baseline approach.

Instruction Cache Result: Figure 5a shows the results of Instruction cache analysis using the augmented abstract interpretation (AI+SAT) approach. We perform the analysis for all programs in Table I, for a block size of 64 Bytes, on a 4-way set-associative L1 cache. For a fair comparison, we chose the cache size approximately equal to the program size (closest power of two), for each subject program. We assume that there is no L2 cache for this experiment and the latency for memory access is 36 cycles. We also compare our approach with the work in [6] (say AI+CBMC). The framework proposed in [6] first generates the cache hit-miss categorization of a program using basic abstract interpretation. It then uses CBMC [7] to refine the set of NC categorized memory blocks. As a result of this refinement, the WCET estimate might improve.

We compare the improvements in WCET estimation achieved by the AI+SAT approach, with that of AI+CBMC approach. We observe that the estimates generated by the AI+SAT approach are more accurate than that of the AI+CBMC approach, when both the analysis are run for a comparable amount of time. We observed a maximum improvement in WCET estimates of up to 25% for this set of experiments. This improvement can be attributed to the fact that all the programs in Table I have multiple program paths inside loops. Some of these program paths are infeasible and hence cause some over-estimation in the base analysis. By applying the AI+SAT approach, we were able to remove some of the over-estimation caused due to such infeasible paths.

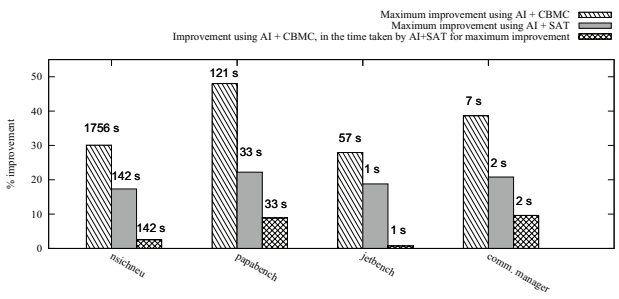
However, it is worthwhile to mention that AI+CBMC is a verification based method. As a result, it might be able to produce better estimates, if run for sufficiently long. This introduces a trade-off between WCET accuracy and analysis time for using AI+SAT over AI+CBMC.

Data Cache Result: Figure 5c shows the results of data cache analysis using the AI+SAT approach. Note that the baseline analysis for data cache is scope-aware persistence analysis [18]. We assume that L1 cache hit latency is 1 cycle whereas the memory access latency is 36 cycles. Although all the programs in Table I have very little data accesses on their infeasible paths, we still get a noticeable improvement for most of the programs via AI+SAT approach. The improvements shown in Figure 5c are the maximum improvement for programs in Table I, for any given cache size. The maximum time taken for all the experiments under this section was under 2 minutes. As Figure 5c shows, all the subject programs other than `nsichneu` have noticeable improvement in their WCET estimates. This is because `nsichneu` has a very small data set and it has very little data accesses across its infeasible paths. However, this should not be considered as a limitation of our approach, as the AI+SAT based framework will give reasonable improvement for programs with many conflicting data accesses along infeasible paths.

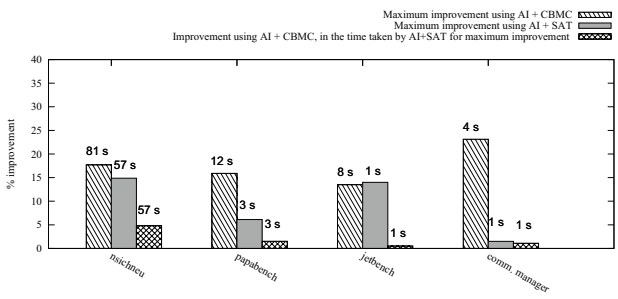
Another factor which might be affecting the efficacy of our method can be the underlying address analysis. Address analysis usually generates an over-approximation of the memory ranges which can be accessed for a given data access. Due to this over-approximation, additional memory blocks might lose their control flow information over the merge operations. This leads to an imprecision in the abstract cache states and overall WCET. Therefore, using a better address analysis will directly improve the WCET accuracy via our approach.

Branch Target Buffer Result: Figure 5d shows the results of branch target buffer (BTB) analysis, using the AI+SAT approach. All the experiments for this set of analysis took less than a minute to complete. We used a 2-way set associative BTB with 256 entries. Also we took the branch misprediction penalty as 15 cycles. The maximum improvement in WCET estimation was observed for `Papabench` (approximately 14%). We did not observe any considerable improvement for `Jetbench`. This can be due to fact that `Jetbench` has very less branch instructions along the infeasible program paths. For the other two subject programs we observed a moderate improvement in the WCET estimation.

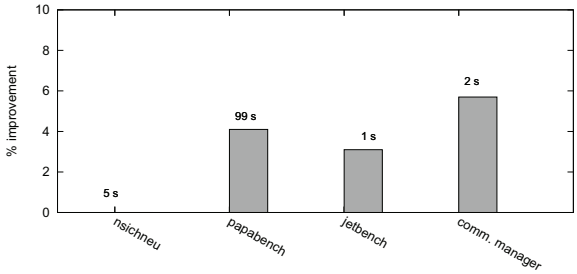
[1] code size in bytes = ending instruction address - starting instruction address



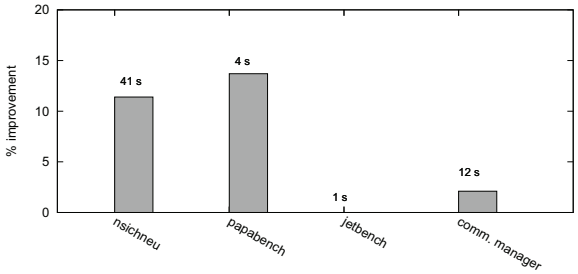
(a) Instruction cache analysis



(b) Shared instruction cache analysis



(c) Data cache analysis



(d) Branch target buffer analysis

Fig. 5: Improvement in the WCET accuracy via AI+SAT approach, analysis time (in seconds) is shown above each bar

Shared cache result: Finally, we present the results for the shared instruction cache analysis. For this set of experiments, we assume a multi-core system with two cores. Moreover, we assume that each core has a private L1 instruction cache and the L2 instruction cache is shared by both the cores. We ran a program from Table I on $Core_1$ and a program from Table II (Table II programs are taken from [4]) on $Core_2$.

Subject programs	Description	Code size	
		Bytes[1]	LOC
jfdcint	Discrete cosine transform on 8x8 block	5512	375
edn	Signal processing application	3160	285
ndes	Complex embedded code	3816	231
adpcm	ADPCM coder	12568	879

TABLE II: Program Set II

We used a direct mapped L1 cache with a cache size of 256 Bytes and a block size of 32 bytes. We choose a small L1 cache to generate sufficient number of conflicts in the L2 cache. The size of the shared L2 cache is chosen depending upon the code size of the program running on $Core_1$. L2 cache hit latency is taken as 6 cycles and memory access latency is taken as 30 cycles. We perform the shared cache analysis using the AI+SAT approach and measure the improvement in WCET estimation for the programs running on $Core_2$. We then compare the improvements achieved by our approach, with the improvements achieved by using the AI+CBMC [6] approach (using the same cache configurations). AI+CBMC was run for the same amount of time as AI+SAT.

Figure 5b shows the *geometric mean* of improvements in WCET estimation, for all the programs running on $Core_2$. We observed a noticeable improvement in the WCET estimates for all the programs running on $Core_2$, produced by the AI+SAT approach. We also observed that the improvements achieved by the AI+SAT were significantly better than the improvement achieved by the AI+CBMC, in the same amount of time. By applying AI+SAT to the programs running on $Core_1$, we were able to reduce a reasonable number of conflicts in the shared cache. This leads to more accurate WCET estimates, for programs running on $Core_2$.

Discussion: In this section, we have described some of the experiments we performed to evaluate the efficacy of our framework. We have performed experiments to analyze instruction caches, data caches, branch target buffers and shared instruction caches via AI+SAT. We observed a reasonable improvement in WCET estimation for most of the subject programs, in all of the above mentioned analysis (key results of the experiments are presented in Figure 5). This observation supports the fact that AI+SAT approach can indeed be used as a general framework to perform various parts of micro-architectural modeling. We also observe that better improvements in WCET estimation were achieved for programs which had more infeasible paths. Since this property is often observed in auto-generated software, our analysis can be useful for micro-architectural modeling of such software.

VI. Related work

Since the initiation of research in WCET analysis, micro-architectural modeling has been an active research area. Initial work on micro-architectural modeling has used integer linear programming (ILP) [19]. However, the work of [19] faces scalability issues due to a huge number of generated ILP constraints. Therefore, abstract interpretation (AI)

based micro-architectural modeling have been proposed subsequently. Among others, the work proposed in [23] deserves mention. The work in [23] proposes AI-based cache analysis for WCET prediction. The solution has been proved scalable and it has also been applied in industry strength tool chain (*e.g.* aiT [1]). The basic approach proposed in [23] has also later been extended to analyze multi-level caches [17], data caches [18], branch target buffers (BTB) [8] and shared caches [15]. As a result, for most of the WCET analyzers, AI has emerged to be the basic approach used for micro-architectural modeling.

Program flow analysis has also been parallely investigated by the WCET research community. Program flow analysis has mainly been concentrated towards finding the loop bound and infeasible path patterns in a program [9], [11], [22]. As certain infeasible path patterns can easily be specified by the programmer, researchers have proposed a *flow-fact* language [10] to capture the common forms of infeasible path patterns. Subsequent research [12] has developed techniques to compute the flow facts of a program automatically. However, all these works aim to improve the WCET at the program path level and they ignore the effect of flow analysis results on micro-architectural modeling. Our work leverages the research done in program flow analysis for precise micro-architectural modeling, an issue mostly ignored by the research community. Nevertheless, the precision gain obtained by our framework will lead to a better WCET of the overall program.

A recent approach [13] has looked at the combination of abstract interpretation (AI) and model checking for WCET analysis. However, the work proposed in [13] uses model checking for WCET calculation only; cache analysis is accomplished by conventional AI-based methods. As a result, the work proposed in [13] can overestimate the WCET due to the presence of infeasible cache states. On the other hand, our primary goal is to generate an infeasible path-aware micro-architectural modeling framework (including cache analysis) to improve the accuracy of WCET estimation. Other works such as [16] have proposed techniques to make AI-based analysis, path-sensitive, but none of these works discusses techniques for path-sensitive, micro-architectural analysis.

Our previous work [6] has investigated precise cache modeling using abstract interpretation (AI) and model checking. The primary goal of [6] was to efficiently eliminate certain infeasible cache states. This was accomplished by using the path-sensitive search process employed in a model checker. However, the work in [6] requires code instrumentation to refine the number of cache conflicts. Such a code instrumentation requires modification for different cache replacement policies and for micro-architectural modeling other than caches. On the other hand, our work in this paper proposes to extend any AI-based framework using satisfiability (SAT) checking and it does not require any code instrumentation. As a result, we can show the applicability of our current approach for a variety of micro-architectural modeling including (shared) instruction caches, branch target buffers and data caches. Moreover, the gain in WCET accuracy using our approach is much better compared to [6], in a comparable amount of time.

VII. Conclusion

In this paper, we have designed and implemented a general micro-architectural modeling framework using abstract interpretation and satisfiability checking. The key novelty in our work is to eliminate the infeasible micro-architectural states. Such an elimination has been achieved by using the program flow analysis results and augmenting the classical abstract interpretation framework. Our experiments clearly suggest the importance of our proposal, as we are able to substantially improve the accuracy of WCET analysis in the presence of many infeasible program paths.

VIII. Acknowledgement

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

References

[1] aiT AbsInt. http://www.absint.com/ait.
[2] MINISAT satisfiability solver. http://minisat.se/.
[3] Rhapsody. http://www-01.ibm.com/software/awdtools/rhapsody/.
[4] WCET benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
[5] CTAS case study overview, requirements. In *SCSEM Case Study*, 2003.
[6] S. Chattopadhyay and A. Roychoudhury. Scalable and precise refinement of cache timing analysis via model checking. In *RTSS*, 2011.
[7] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *TACAS*, 2004.
[8] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3), 2000.
[9] D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO*, 2009.
[10] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *RTSS*, 2000.
[11] C. A. Healy et al. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3), 2000.
[12] J. Gustafsson et al. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS*, 2006.
[13] M. Lv et al. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*, 2010.
[14] X. Li et al. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007. http://www.comp.nus.edu.sg/~rpembed/chronos.
[15] Y. Li et al. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.
[16] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS*, 1998.
[17] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.
[18] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *RTAS*, 2011.
[19] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Des. Autom. Electron. Syst.*, 4(3), 1999.
[20] Fadia Nemer, Hugues Cassé, and Pascal Sainrat. Papabench: a free real-time benchmark. *WCET Workshop*, 2006.
[21] M Qadri, D. Matichard, and K. M. Maier. JetBench: an open source real-time multiprocessor benchmark. *ARCS*, 2010.
[22] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *DAC*, 2008.
[23] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3), 2000.
[24] Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *VMCAI*, 2004.