Precise Cache Timing Analysis via Symbolic Execution

Duc-Hiep Chu
National University of Singapore
Email: hiepcd@comp.nus.edu.sg

Joxan Jaffar
National University of Singapore
Email: joxan@comp.nus.edu.sg

Rasool Maghareh
National University of Singapore
Email: rasool@comp.nus.edu.sg

***Abstract*—We present a framework for WCET analysis of programs with emphasis on cache micro-architecture. Such an analysis is challenging primarily because of the timing model of a *dynamic* nature, that is, the timing of a basic block is heavily dependent on the context in which it is executed. At its core, our algorithm is based on symbolic execution, and an analysis is obtained by locating the "longest" symbolic execution path. Clearly a challenge is the intractable number of paths in the symbolic execution tree. Traditionally this challenge is met by performing some form of abstraction in the path generation process but this leads to a loss of path-sensitivity and thus precision in the analysis. The key feature of our algorithm is the ability for *reuse*. This is critical for maintaining a high-level of path-sensitivity, which in turn produces significantly increased accuracy. In other words, reuse allows scalability in path-sensitive exploration. Finally, we present an experimental evaluation on well known benchmarks in order to show two things: that systematic path-sensitivity in fact brings significant accuracy gains, and that the algorithm still scales well.**

I. Introduction

Hard real-time systems need to meet *hard* deadlines. Static Worst-Case Execution Time (WCET) analysis is therefore very important in the design process of real-time systems.

Traditionally, WCET analysis is proceeded in three phases. The first phase, referred to as low-level analysis, involves micro-architectural modeling to determine the maximum execution time for each basic block. The second phase concerns determining the infeasible paths and loop bounds from the program. The third phase computes the aggregated WCET bound, employing the results of the prior phases. In some recent approaches, the second and third phases are fused into one, called generally as high-level analysis. Importantly, for scalability, in the literature low-level analysis and high-level analysis are often performed *separately*.

The main difficulty of low-level analysis comes from the presence of performance enhancing processor features such as caches and pipeline. This paper focuses on caches, since their impact on the real-time behavior of programs is much more than other features [1]. Cache analysis – to be scalable – is often accomplished using abstract interpretation (AI), e.g., [2]. In particular, we need to analyze the memory accesses of the input program via an iterative fixed-point computation. This process can be efficient, but the results are often *not precise*. There are two main reasons for the imprecision:

(1) The cache states are joined at the control flow merge points. This results in subsequently over-estimating the potential cache misses.
(2) Beyond the one-iteration virtual unrolling of loops [2], AI is unable to give different timings for a basic block executed in different iterations of a loop.

A direct improvement would be to curtail the above-mentioned merge points. That is, when traversing the CFG from a particular source node to a particular sink node: (a) do not visit any intermediate node which is unreachable from the source node; (b) perform merging once traversals are finished, *only* at the sink node. This process should be performed on some, but not necessarily all the possible source/sink node pairs.

Recent works [3], [4] fall into this class. They employ a form of infeasible path discovery, so that unreachable micro-architectural states can be excluded from consideration, i.e., via (a), thus yielding more accurate WCET bounds. We note, however, such addition of infeasible path discovery is quite limited. We will elaborate more in Sections VI and VII.

More importantly, in the literature, most algorithms employ a *fixed-point* computation in order to ensure *sound* analysis across loop iterations. Thus, they inherit the imprecision of AI, identified as reason (2) above. That is, a fixed-point method will compute a worst-case timing for each basic block in all possible contexts, even though the timings of a basic block in different iterations of a loop can diverge significantly.

To overcome the identified shortcomings, we propose a *symbolic execution* algorithm where low-level analysis and high-level analysis are synergized. In our algorithm, loops are unrolled fully[1] and summarized. The only abstraction (or merging) performed is within and once at the end of a loop iteration, not across loop iterations. This leads to a precise inspection of the timing measured from the underlying hardware model, because the (feasibility of) micro-architectural states can be tracked across the iterations. Clearly, our method would produce very accurate WCET bounds since it preserves the programs operational semantics in detail, down to the cache.

While precision is ensured, the scalability of our algorithm becomes questionable. Obviously, a naive attempt to perform exhaustive symbolic execution will not scale. Chu and Jaffar demonstrated that exhaustive symbolic execution can be made scalable, in the presence of (nested) loops, by employing the novel concept of *reuse with interpolation* [5]. Their concept of reuse [5] relies on the fact that the timing of each basic block is determined as *a constant* by a prior low-level analysis.

In the setting of this paper, because of the presence of micro-architectural features such as caches, the timing for each basic block is no longer statically fixed. Instead, the timing depends on the *context* in which the block is executed. We refer to this as *dynamic timing*, and in a dynamic timing model, reuse with interpolation alone is *no longer sound*. In short, one main contribution of this paper is then, furnishing the concept of reuse so that it is still *sound* and *effective* for the setting

[1]We note that our loop unrolling is done virtually and not physically, and is different from loop unrolling done by compilers.

that the timing of a basic block, under different contexts, can be arbitrarily different.

In Section VI, we demonstrate on realistic benchmarks that our algorithm is accurate as well as scalable. Note that our benchmarks include `statemate` and `nsichneu`, which are often used to evaluate the scalability of WCET analyzers. In addition to proving metrics, we will elaborate our improvement in the context of different program characteristics such as loop behavior and the amount of infeasible paths.

II. Overview

Fig. 1(a) informally depicts a symbolic execution tree, where each triangle presents a subtree. The program contexts for the left and right subtrees, i.e., the symbolic states s_0 and s_1 respectively, are of the same program point. If we had applied the algorithm in [5] on the left subtree, we would obtain two things: an *interpolant* Ψ_0, a generalization of s_0, encapsulating any context that would preserve the infeasible paths (indicated with a red cross) of the subtree. We also obtain a "representative" path, called a *witness*, indicated in blue, which gives rise to the WCET (15 in this case) of the subtree.

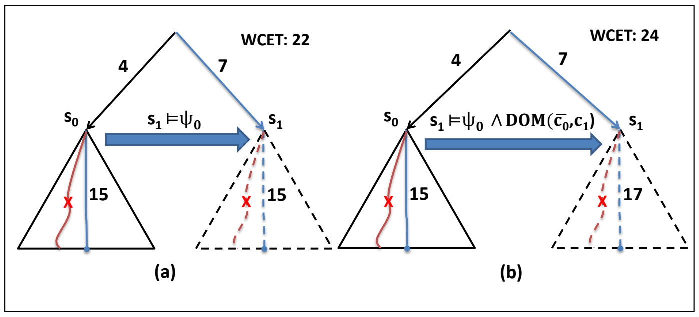


Fig. 1: Reuse of Summarizations: (a) [5] vs. (b) This Paper

The algorithm [5] now considers the right subtree, where two tests are performed. First the context s_1 is checked if it implies the interpolant. If so, every infeasible path in the left subtree remains infeasible in the right subtree. A second test is whether the witness path is still feasible. If both tests are passed, the analysis can be *reused* here, and the WCET of the right subtree can now be computed without traversal.

The final analysis, at the root of the tree, can be computed by collating the analyses of the left and right subtrees, and we can now determine its value of 22 as indicated. Note, importantly, that we never actually traversed the path that gives rise to this result; instead, we *inferred* its value.

So far we have only briefly overviewed the previous work [5]. Next consider Fig. 1(b) where we now focus on dynamic timing, which arises because of cache configurations. A *cache state* c_0 is also part of the context s_0 of the subtree on the left. After analyzing the left subtree we obtain an interpolant Ψ_0 and a witness path (indicated in blue) as before. A most important point of departure here from Fig. 1(a) is that the reuse of this witness path, solely as before, is *unsound* in general. To remedy this, we now compute a *dominating condition* $\overline{c_0}$. Essentially, this is a formula which describes an abstract cache configuration which is sufficient to guarantee the witness path *remains optimal*, i.e., the worst-case path in the subtree, when encountering a new context.

In Fig. 1(b), suppose the dominating condition applies, that is, suppose that the cache context c_1 is covered by $\overline{c_0}$. We indicate this by the predicate $\text{DOM}(\overline{c_0}, c_1)$. Now this allows us to reuse the witness path. We then need to proceed *replaying* the witness path under the new cache configuration c_1. This, importantly, can lead to new *value* of the path (now 17), which is different from the original value (15). Finally, we can conclude the analysis on the whole tree with the value 24.

Now suppose the dominating condition did *not* apply. Then the path indicated by 17 may not be the worst-case path in the right subtree. For example, there could be a path of length 18 somewhere else in the subtree. If we reuse the witness path, we would now report, wrongly, a final value of 24.

III. General Framework

A. Symbolic Execution with Abstract Cache

We model a program by a transition system. A transition system $\mathcal{P}$ is a tuple $\langle \mathcal{L}, \ell_0, \longrightarrow \rangle$ where $\mathcal{L}$ is the set of program points, $\ell_0 \in \mathcal{L}$ is the unique initial program point. Let $\longrightarrow \subseteq \mathcal{L} \times \mathcal{L} \times Ops$, where Ops is the set of operations, be the transition relation that relates a state to its (possible) successors by executing the operations. All basic operations are either assignments or "assume" operations. The set of all program variables is denoted by $Vars$. An assignment $x := e$ corresponds to assign the evaluation of the expression e to the variable x. The expression $assume(cond)$ means: if the conditional expression $cond$ evaluates to true, execution continues; otherwise it halts. We shall use $\ell \xrightarrow{op} \ell'$ to denote a transition relation from $\ell \in \mathcal{L}$ to $\ell' \in \mathcal{L}$ executing the operation $op \in Ops$. Clearly a transition system is derivable from a control flow graph (CFG).

Definition 1 (Symbolic State)**.** *A symbolic state s is a tuple $\langle \ell, c, \sigma, \Pi \rangle$ where $\ell \in \mathcal{L}$ is the current program point, c is the abstract cache state the symbolic store σ is a function from program variables to terms over input symbolic variables, and finally the path condition Π is a first-order formula over the symbolic inputs.* □

The abstract cache is modeled following the standard semantics of abstract cache for *must* analysis, formally defined in [2]. The purpose of Π is to accumulate constraints on input values which enable execution to reach this state.

Let $s_0 \stackrel{\text{def}}{=} \langle \ell_0, c_0, \sigma_0, \Pi_0 \rangle$ denote the unique initial symbolic state, where c_0 is the initial abstract cache state, usually initialized as an empty cache. At s_0 each program variable is initialized to a fresh input symbolic variable. For every state $s \equiv \langle \ell, c, \sigma, \Pi \rangle$, the evaluation $[\![e]\!]_\sigma$ of an arithmetic expression e in a store σ is defined as usual: $[\![v]\!]_\sigma = \sigma(v)$, $[\![n]\!]_\sigma = n$, $[\![e + e']\!]_\sigma = [\![e]\!]_\sigma + [\![e']\!]_\sigma$, $[\![e - e']\!]_\sigma = [\![e]\!]_\sigma - [\![e']\!]_\sigma$, etc. The evaluation of the conditional expression $[\![cond]\!]_\sigma$ can be defined analogously. The set of first-order logic formulas and symbolic states are denoted by *FO* and *SymStates*, respectively.

Our analysis is performed on LLVM IR, which is expressive enough for cache analysis and where the general CFG of the program can be readily constructed. Given a program point ℓ, an operation $op \in Ops$, and a symbolic store σ, the function $acc(\ell, op, \sigma)$ denotes the sequence of memory block accesses by executing *op* at the symbolic state $\langle \ell, c, \sigma, \cdot \rangle$. While the program point ℓ identifies the instruction cache access, the sequence of data accesses are obtained by considering both *op* and σ together.

Definition 2 (Transition Step)**.** *Given $\langle \mathcal{L}, l_0, \longrightarrow \rangle$, a transition system, and a symbolic state $s \equiv \langle \ell, c, \sigma, \Pi \rangle \in SymStates$, the symbolic execution of transition $tr : \ell \xrightarrow{op} \ell'$ returns another symbolic state s' defined as:*

$$s' \overset{\text{def}}{=} \begin{cases} \langle \ell', c', \sigma, \Pi \wedge cond \rangle & \textit{if } op \equiv assume(cond) \\ \langle \ell', c', \sigma[x \mapsto [\![e]\!]_\sigma], \Pi \rangle & \textit{if } op \equiv x := e \end{cases} \textit{ where}$$

c' is the new abstract cache derived from c and the sequence of accesses. □

Note that c' is computed using the standard *update* function of the abstract cache semantics for *must* analysis from [2]. Thus c' is $\mathcal{U}(acc(\ell, op, \sigma), c)$.

Abusing notation, the execution step from s to s' is denoted as $s \xrightarrow{tr} s'$ where tr is a transition. Given a symbolic state $s \equiv \langle \ell, c, \sigma, \Pi \rangle$ we also define $[\![s]\!] : SymStates \to FO$ as the projection of the formula

$$[\![\Pi]\!]_\sigma \wedge \bigwedge_{v \in Vars} v = [\![v]\!]_\sigma$$

onto the set of program variables *Vars*. The projection is performed by the elimination of existentially quantified variables.

For convenience, when there is no ambiguity, we just refer to the symbolic state s using the abbreviated tuple $\langle \ell, c, [\![s]\!] \rangle$ where ℓ and c are as before, and $[\![s]\!]$ is obtained by projecting s as described above. A path $\pi \equiv s_0 \to s_1 \to \ldots s_m$ is feasible if $s_m \equiv \langle \ell, c_m, [\![s_m]\!] \rangle$ and $[\![s_m]\!]$ is satisfiable. Otherwise, the path is called *infeasible* and s_m is called an infeasible state. Here we query a *theorem prover* for satisfiability checking on the path condition. We assume the theorem prover is sound, but not complete. If $\ell \in \mathcal{L}$ and there is no transition from ℓ to another program point, then ℓ is called the *end point* of the program. Under that circumstance, if s_m is feasible, then s_m is called *terminal* state.

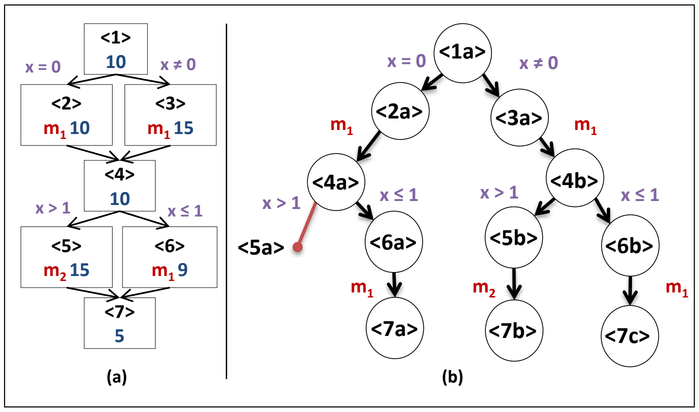


Fig. 2: (a) a CFG and (b) Its Symbolic Execution Tree

Example 1 (Symbolic Execution Tree)**.** *Consider the CFG in Fig. 2(a). Each node abstracts a basic block. Inside the basic blocks, the* program points *($\langle 1 \rangle$, $\langle 2 \rangle$, $\cdots$, $\langle 7 \rangle$) are shown and the integer constants (in blue color) are the* static timings *(timings of the corresponding basic blocks while assuming that all memory accesses are hits). We also show the* memory accesses *(in red color). They are accesses to memory blocks m_1 and m_2. For brevity, we might use interchangeably the identifying program point when referring to a basic block. Two outgoing edges signify a branching structure, while the branch conditions are labeled beside the edges. In this example, we assume a direct-mapped cache, initially empty; and m_1 and m_2 conflict with each other in the cache.*

Next, in Fig. 2(b), we depict the symbolic execution tree *of the program. Each node, shown as a circle, is identified by the corresponding program point, followed by a letter to distinguish the multiple visits to the same program point. Each node is associated with a symbolic state, but for simplicity we do not explicitly show any state content in the figure.*

Now assume that none of the basic blocks modifies the variable x. At node $\langle 5a \rangle$, the projection of the path condition over the program variables $[\![s_{5a}]\!]$ is $x = 0 \ \wedge \ x > 1$, which is equivalent to false*. In other words, the leftmost path in Fig. 2(b) is in fact an* infeasible path*. Moreover, each node will be assigned a cache state. The cache state of a child node is determined by the cache state of the parent node and the memory accesses in the corresponding basic block. For example, the cache at node $\langle 4b \rangle$ contains m_1, while the cache at node $\langle 7b \rangle$ contains m_2, evicting m_1 out of the cache.*

Here we omit the standard definitions for loop, loop head, end point of loop body and same nesting level. (However, they are included in the Appendix, Section IX-A.) We assume that each loop has only one loop head and one unique end point. For each loop, following the back edge from the end point to the loop head, we do not execute any operation. This can be achieved by a preprocessing phase.

B. Constructing Summarizations

In our framework, a "subtree" is a portion of the symbolic execution tree. Given a state s and program point ℓ_2 such that (a) state $s \equiv \langle \ell_1, c, [\![s]\!] \rangle$ appears in the tree, and (b) ℓ_2 post-dominates ℓ_1, then $subtree(s, \ell_2)$ depicts all the paths emanating from s and, if feasible, terminate at ℓ_2. (Note that ℓ_2 may not be the end point of the whole tree.) We call ℓ_1 and ℓ_2 the entry and exit points of the subtree.

A summarization of a subtree, intuitively, is a succinct description of its analysis. This is formalized as a tuple of certain important components of the analysis. These are: the entry and exit program points, an interpolant describing infeasible paths, a witness describing the longest path in the subtree, a domination condition ensuring the witness represents the appropriate worst-case path in the subtree, and finally an abstract transformer relating the input and output program variables and an abstract transformer relating the input and output cache configurations. The abstract transformers are used to generate the outgoing context at ℓ_2.

We start with our notion of interpolant. The idea here is to approximate at the root of a subtree, the weakest precondition in order to maintain the infeasibility of all the nodes inside. (An exact computation is in general intractable.) In the context of program verification, an interpolant captures succinctly a condition which ensures the *safety* of the tree at hand. Adapting this to program analysis is first done in [6] which formalized a generalized form of dynamic programming. Since all infeasible nodes are excluded from calculating the analysis result of a subtree, in order to ensure soundness, at the point of reuse, all such infeasibility must also be maintained.

Next, we discuss the *witness* concept. Intuitively, it is a path that depicts the WCET of a subtree. More specifically, it is depicted by $\Gamma \overset{\text{def}}{=} \langle t, \Upsilon, \pi \rangle$ where t is the (static) execution time of the instructions along the path assuming all the memory accesses are cache hits, Υ is the sequence of all memory accesses along the path, and π is the path constraints along the witness.

In case Υ contains consecutive accesses to the same memory block, all-but-first accesses in that subsequence can be classified as Always Hit and, importantly they will not affect the resulting cache state. As an optimization, we consider them and redundant and remove them from Υ. This helps reducing the size of Υ.

The timing of a witness is obtained dynamically from t and replaying the sequence Υ under an incoming cache state c. The feasibility of a witness w.r.t. to an incoming context is determined by checking if $[\![\pi]\!] \wedge [\![s]\!]$ is satisfiable. In what follows, we abbreviate $[\![\pi]\!] \wedge [\![s]\!]$ by $[\![\Gamma]\!]$.

We say that two nodes in a symbolic execution tree are *similar* if they refer to the same program point. Thus two subtrees are similar if they share the same entry and exit program points.

We next discuss dominating condition, another component of our analysis of a subtree. Intuitively, this is a description of what cache configuration is needed in order that the witness *remains optimal* in a similar subtree. That is, in an analysis of the latter subtree, the witness remains the longest path. More specifically, the constraints in the dominating condition are either of the form $m_i \in \mathit{cache}$, indicating the presence of memory block m_i, or $m_i \notin \mathit{cache}$, indicating the opposite.

We now discuss an abstract transformer Δ_p of a subtree from ℓ_1 to ℓ_2 which is an abstraction of all feasible paths (w.r.t. the incoming symbolic state s) from ℓ_1 to ℓ_2. Its purpose is to capture an input-output relation between the program variables In our implementation, we adopt from [5] which uses the polyhedral domain [7].

Similarly, we also have an abstract transformer Δ_c for cache. Suppose s is at program point ℓ_1 and a summarization of $\mathit{subtree}(s, \ell_1)$ is reused at another visit to ℓ_1 with the incoming cache state c_1, then the cache state at ℓ_2 can be generated by applying the abstract transformer of cache to c_1. That is, this transformer captures the memory accesses along the feasible paths, which start from s and end at ℓ_2.

We collect together the components discussed above into a summarization.

Definition 3. *A* summarization *of* $\mathit{subtree}(s, \ell_2)$*, where* ℓ_1 *is the program point of* s*, is a tuple*

$$[\ell_1, \ell_2, \Psi, \Gamma, \delta, \Delta_p, \Delta_c]$$

where Ψ *is an interpolant,* Γ *is the witness and* δ *is the dominating condition.* Δ_p *is an abstract transformer relating the input and output variables and finally,* Δ_c *is a an abstract transformer of cache.* □

We now display a key feature of our algorithm: reuse of a summarization. Suppose we have already computed a summarization $[\ell_1, \ell_2, \Psi, \Gamma, \delta, \Delta_p, \Delta_c]$ where the witness is $\Gamma \equiv \langle t, \Upsilon, \pi \rangle$. Suppose we then encounter a symbolic state $s' \equiv \langle \ell_1, c, [\![s']\!] \rangle$. The summarization now can be reused if:

1) $[\![s']\!]$ implies the stored interpolant Ψ i.e., $[\![s']\!] \models \Psi$.
2) The context of s' is consistent with the witness formula, i.e., $[\![\pi]\!] \wedge [\![s']\!]$ is satisfiable.
3) The dominating condition is satisfied by c, i.e., $\textsc{dom}(\delta, c)$ holds.

The WCET of the subtree beneath the state s' is then derived from the witness Γ and the cache state c. The WCET of the subtree is t plus the sum of the access times of all the memory accesses in Υ. Using the context c, we resolve each memory access in Υ to either a cache hit or a cache miss.

We now conclude this subsection by mentioning that we only summarize at select program points. Given entry point ℓ_1, the corresponding exit point ℓ_2 is determined as follows. It is the program point that post-dominates ℓ_1 s.t. ℓ_2 is of the same nesting level as ℓ_1 and either is (1) an end point of the program, or (2) an end point of some loop body. In other words, we only perform "merging" abstraction at loop boundaries. As ℓ_2 can always be deduced from ℓ_1, in a summarization, we omit the component about ℓ_2.

IV. An Example Analysis

Consider the CFG and symbolic execution tree in Fig. 3. Here we assume a direct-mapped cache, initially empty, and a *cache miss penalty* of 10 cycles. Consider accesses to memory blocks m_1, m_2, m_3, and m_4, where only m_1 and m_3 conflict with each other in the cache. Note that in Fig. 3(b), we have not (fully) drawn the subtree below node $\langle 4b \rangle$.

Suppose the subtree $\langle 7a \rangle$ has been analyzed, and its summarization is $[\langle 7 \rangle, \Psi, \Gamma, \delta, \Delta_p, \Delta_c]$. We now explain the components of this summarization. The interpolant Ψ is easily determined as true because all (two) paths of this subtree are feasible. Next, because the incoming cache state which contains only m_1, the timing of the sub-path $\langle 7a \rangle$, $\langle 8a \rangle$, $\langle 10a \rangle$ is $40 = (\mathbf{10} + 5 + \mathbf{10} + 15)$, with both accesses are misses. Similarly, the timing of the other sub-path $\langle 7a \rangle$, $\langle 9a \rangle$, $\langle 10b \rangle$ is $45 = (\mathbf{10} + 5 + \mathbf{10} + \mathbf{10} + 10)$. So, the sub-path $\langle 7a \rangle$, $\langle 9a \rangle$, $\langle 10b \rangle$ is longer than the other and it is chosen as the worst-case path[2] of subtree $\langle 7a \rangle$. Consequently, the witness Γ is computed as $\langle 15, [m_2, m_3, m_4], z \geq 0 \rangle$, where 15 is the static timing of the witness path, $[m_2, m_3, m_4]$ are the memory accesses along the path, and $z \geq 0$ is the (partial) path constraints of the path. Next, we capture a dominating condition δ as $m_4 \notin \mathit{cache}$. This condition is sufficient to ensure that the chosen path dominates (i.e., is longer than) any other path in the subtree.

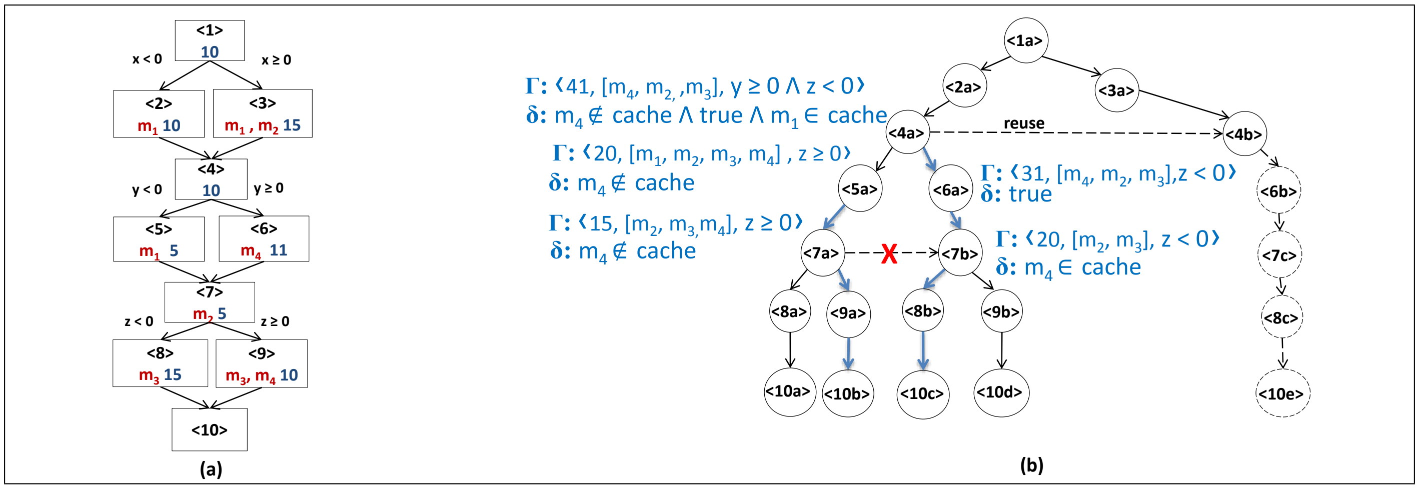
The abstract transformer Δ_p is the trivial one where the output is the same as the input. This is because in this example we abstract away all the instructions executed by the basic blocks. Next, since m_2 and m_3 are common along both paths, the abstract transformer for cache Δ_c is $[m_2, m_3]$. More specifically, any path from program point $\langle 7 \rangle$, if it reaches to program point $\langle 10 \rangle$, then m_2 and m_3 are for sure present in the cache. However, this is not true for m_4. In short, after analyzing $\langle 7a \rangle$, we also have computed a summarization $[7, \mathit{true}, \langle 15, [m_2, m_3, m_4], z \geq 0 \rangle, m_4 \notin \mathit{cache}, \mathit{Id}(\mathit{Vars}), [m_2, m_3]]$.

For brevity, in what follows, we do not detail on abstract transformers Δ_p and Δ_c.

Next we propagate the analysis of $\langle 7a \rangle$ to its parent $\langle 5a \rangle$ whose summarization is now updated so that the witness is stored in the form $\langle 20, [m_1, m_2, m_3, m_4], z \geq 0 \rangle$, where 20 is computed as the sum of: (1) the static timing of block $\langle 5 \rangle$, which is 5; (2) the static timing of the witness for $\langle 7a \rangle$, which is 15. The dominating condition is $m_4 \notin \mathit{cache}$, as before.

We fast forward to node $\langle 7b \rangle$, and consider now if the above analysis of $\langle 7a \rangle$ can be *reused*. That is, even though we have depicted the subtree $\langle 7b \rangle$ in full, could we in fact have simply declared that the witness in the subtree below $\langle 7b \rangle$ would remain the same as the witness in subtree below $\langle 7a \rangle$? (Recall that the witness in the subtree below $\langle 7a \rangle$ spans along the program points $\langle 7 \rangle$, $\langle 9 \rangle$, $\langle 10 \rangle$.) Unfortunately, the answer is negative, and the reason is that the dominating condition, $m_4 \notin \mathit{cache}$, is not met. This non-reuse is depicted by a red cross. We thus have to analyze $\langle 7b \rangle$ fully. We get

[2]When it is clear, we often use "path" to mean "sub-path".



<1> 10
x < 0
x ≥ 0
<2> m1 10
<3> m1, m2 15
<4> 10
y < 0
y ≥ 0
<5> m1 5
<6> m4 11
<7> m2 5
z < 0
z ≥ 0
<8> m3 15
<9> m3, m4 10
<10>
(a)
Γ: ⟨41, [m4, m2, m3], y ≥ 0 ∧ z < 0⟩
δ: m4 ∉ cache ∧ true ∧ m1 ∈ cache
Γ: ⟨20, [m1, m2, m3, m4], z ≥ 0⟩
δ: m4 ∉ cache
Γ: ⟨15, [m2, m3, m4], z ≥ 0⟩
δ: m4 ∉ cache
Γ: ⟨31, [m4, m2, m3], z < 0⟩
δ: true
Γ: ⟨20, [m2, m3], z < 0⟩
δ: m4 ∈ cache
reuse
<1a> <2a> <3a> <4a> <4b> <5a> <6a> <6b> <7a> <7b> <7c> <8a> <9a> <8b> <9b> <8c> <10a> <10b> <10c> <10d> <10e>
(b)

Fig. 3: (a) a CFG (with memory accesses and static instruction timing shown in each block); and (b) Our Analysis Tree

a different longest sub-path this time, $\langle 7b\rangle$, $\langle 8b\rangle$, $\langle 10c\rangle$, with the witness $\langle 20, [m_2, m_3], z < 0\rangle$. The dominating condition is also different: $\delta : m_4 \in \mathit{cache}$.

Finally, this analysis of $\langle 7b\rangle$ is propagated for its parent $\langle 6a\rangle$. The dominating condition is $m_4 \in \mathit{cache}$ which always holds due to the access of m_4 at $\langle 6\rangle$. Thus the dominating condition for $\langle 6a\rangle$ is simply true.

Having now analyzed both $\langle 5a\rangle$ and $\langle 6a\rangle$, we can now compute an analysis for their common parent $\langle 4a\rangle$. Here the observed longest sub-path is $\langle 4a\rangle$, $\langle 6a\rangle$, $\langle 7b\rangle$, $\langle 8b\rangle$, $\langle 10c\rangle$, and the witness is stored as $\langle 41, [m_4, m_2, m_3], y \geq 0 \wedge z < 0\rangle$. The dominating condition is conjoined from: (a) the dominating condition of its left child $\langle 5a\rangle$; (b) the dominating condition of its right child $\langle 6a\rangle$; and (c) the reason for the dominance of the above observed longest path over the other path. In particular, δ is $m_4 \notin \mathit{cache} \wedge \mathit{true} \wedge m_1 \in \mathit{cache}$.

Now we can exemplify reuse on the subtree $\langle 4b\rangle$. We first check if the context of $\langle 4b\rangle$ implies the interpolant computed for $\langle 4a\rangle$. Because all paths from $\langle 4a\rangle$ are feasible, the interpolant is true, thus, it trivially holds. We then check if the dominating condition holds. Examining the cache context of $\langle 4b\rangle$, indeed m_1 is in the cache and m_4 is not in the cache. Furthermore, the witness is still feasible w.r.t. the incoming context ($x \geq 0$). So we can reuse the *witness* of $\langle 4a\rangle$, yielding the timing of 61. We remark here that the timing of the sub-path $\langle 4b\rangle$, $\langle 6b\rangle$, $\langle 7c\rangle$, $\langle 8c\rangle$, $\langle 10e\rangle$ is less than the timing of $\langle 4a\rangle$,$\langle 6a\rangle$, $\langle 7b\rangle$, $\langle 8b\rangle$, $\langle 10c\rangle$ because now m_2 is present in the cache at $\langle 4b\rangle$.

Finally, we easily arrive at the WCET of the entire tree, thus, the entire example program, to be 106 cycles (= 10 + **10** + **10** + 15 + 61, since the accesses to m_1 and m_2 at $\langle 3a\rangle$ are cache miss).

Let us reconsider the same example using a *pure* abstract interpretation (AI) framework such as [2]. A pure AI method would typically perform merging at the three join points: $\langle 4\rangle$, $\langle 7\rangle$, $\langle 10\rangle$. Importantly, it discovers that at $\langle 4\rangle$, m_1 *must* be in the cache. Thus, the access to m_1 at $\langle 5\rangle$ is hit. However, at $\langle 7\rangle$, AI has to conservatively declare that m_4 is not in the cache. As a result the access to m_4 at $\langle 9\rangle$ will be cache miss. Consequently, the final worst case timings for the basic blocks that have some memory accesses are: ($\langle 2\rangle$,**20**), ($\langle 3\rangle$,**35**), ($\langle 5\rangle$,**5**), ($\langle 6\rangle$,**21**), ($\langle 7\rangle$,**15**), ($\langle 8\rangle$,**25**), ($\langle 9\rangle$,**30**).

If we aggregate using a path-insensitive high-level analysis, the WCET estimate is 121 (= 10 + max(20, 35) + 10 + max(5, 21) + 15 + max(25,30)). If we aggregate using a path-sensitive high-level analysis [5], we cannot improve the estimate for this example, because the program contains no infeasible paths.

V. Symbolic Execution for Dynamic Timing

Algorithm 1 consists of two important functions. The function Analyze takes as input the initial symbolic state s_0 and the transition system $\mathcal{P}$ of an input program. It then invokes Summarize to generate a summarization for the whole program (line 1). We then compute the WCET by replaying the witness path, starting from the initial cache state c_0. This is considered as a standard task (line 2).

Before elaborating on the Summarize function, we first explain how summarizations are compounded through two helper functions, Compose and Join, presented in Fig. 4.

Compounding Vertically Two Summarizations: Considering $\mathit{subtree}(s_2, \ell_3)$ suffixing $\mathit{subtree}(s_1, \ell_2)$, where $s_2 \equiv \langle \ell_2, c_2, [\![s_2]\!]\rangle$ and $s_1 \equiv \langle \ell_1, c_1, [\![s_1]\!]\rangle$. In other words, a path π_1 from ℓ_1 to ℓ_2 followed by a path π_2 from ℓ_2 to ℓ_3 corresponds a path π in $\mathit{subtree}(s_1, \ell_3)$. The Compose function returns a summarization for $\mathit{subtree}(s_1, \ell_2)$ by compounding the two existing summarizations, respectively for $\mathit{subtree}(s_1, \ell_2)$ and $\mathit{subtree}(s_2, \ell_3)$.

The abstract transformer Δ_p is computed as the conjunction of the input abstract transformers (line 23), with proper variable renaming. We use Combine-Witnesses to compound the witnesses of the two input summarizations and Combine-Caches to construct the overall cache input-output relation for $\mathit{subtree}(s_1, \ell_1)$. For interested readers, Combine-Witnesses and Combine-Caches are elaborated more in the Appendix.

Note that in our implementation, abstract transformers are computed using polyhedral domain. We employ Δ_p to generate *one* continuation context, before proceeding the analysis with subsequently program fragments. Finally, the desired interpolant must capture the infeasiblity of S_1, as well as the infeasibility of S_2 given that we treat $\mathit{subtree}(s_1, \ell_2)$) as an abstract transition, of which the operation is Δ_p. We rely on the function Pre-Cond, which in line 24 under-approximates the weakest-precondition of the post-condition Ψ_2 w.r.t. to the transition relation Δ_p.

Compounding Horizontally Two Summarizations: Given two summarizations of rooted at two nodes which are siblings,

Algorithm 1 Integrated WCET Analysis Algorithm

```
function ANALYZE($s_0$, $\mathcal{P}$)
        Let $s_0$ be $\langle \ell_0, c_0, [\![s_0]\!] \rangle$
⟨1⟩  $[\ell_0, \cdot, \Gamma, \cdot, \cdot, \cdot]$ := SUMMARIZE($s_0, \mathcal{P}$)
⟨2⟩  return COMPUTE-TIMING($\Gamma, c_0$)
end function

function SUMMARIZE($s$, $\mathcal{P}$)
        Let $s$ be $\langle \ell, c, [\![s]\!] \rangle$
⟨3⟩  if ($[\![s]\!] \equiv false$) return $[\ell, false, \langle -\infty, [\,], false \rangle, [\,], false, [\,]]$
⟨4⟩  if (OUTGOING($\ell, \mathcal{P}$) $= \emptyset$)
⟨5⟩      return $[\ell, true, \langle 0, [\,], true \rangle, [\,], Id(Vars), [\,]]$
⟨6⟩  if (LOOP-END($\ell, \mathcal{P}$))
⟨7⟩      return $[\ell, true, \langle 0, [\,], true \rangle, [\,], Id(Vars), [\,]]$
⟨8⟩  $S := [\ell, \Psi, \Gamma, \delta, \Delta_p, \Delta_c]$ := MEMOED($\ell$)
⟨9⟩  if ($[\![s]\!] \models \Psi \ \wedge\ [\![\Gamma]\!] \not\equiv false \ \wedge$ DOM($\delta, c$)) return $S$
⟨10⟩ if (LOOP-HEAD($\ell, \mathcal{P}$))
⟨11⟩     $S_1 := [\cdot, \cdot, \Gamma_1, \cdot, \Delta_{p1}, \Delta_{c1}]$
             := TRANSSTEP($s, \mathcal{P}$, ENTRY($\ell,\mathcal{P}$))
⟨12⟩     if ($[\![\Gamma_1]\!] \equiv false$)
⟨13⟩         $\overline{S}$ := JOIN($c, S_1$, TRANSSTEP($s, \mathcal{P}$, EXIT($\ell, \mathcal{P}$)))
         else
⟨14⟩         Let $tr$ be $\ell \xrightarrow{\Delta_{p1},\Delta_{c1}} \ell'$
⟨15⟩         $s \xrightarrow{tr} s'$
⟨16⟩         $S'$ := SUMMARIZE($s', \mathcal{P}$)
⟨17⟩         $S$ := COMPOSE($S_1, S'$)
⟨18⟩         $\overline{S}$ := JOIN($c, S$, TRANSSTEP($s, \mathcal{P}$, EXIT($\ell, \mathcal{P}$)))
⟨19⟩ else $\overline{S}$ := TRANSSTEP($s, \mathcal{P}$, OUTGOING($\ell, \mathcal{P}$))
⟨20⟩ memo and return $\overline{S}$
end function
```

we want to propagate the information back and compute the summarization for the parent node. While propagation can be achieved by COMPOSE, we need JOIN to "merge" the contributions of the two children to the parent node. Note that unlike COMPOSE, we need to select the longer path between the two witnesses of the input summarizations. Such selection depends on the current cache context. That is why the cache context c is passed as an input to JOIN, which subsequently pass it on to MERGE-WITNESSES.

As before, we use MERGE-WITNESSES and MERGE-CACHES and delegate the details to the Appendix. The abstract transformer Δ_p, however, is computed straightforwardly as the disjunction of the input abstract transformers. All the infeasible paths in both sub-structures must be maintained, thus the desired interpolant is the conjunction of the two input interpolants.

In depth-first traversal of the symbolic execution tree, at a node either (1) a summarization is reused, thus we do not need to expand the node; or (2) after expanding it, we compute its summarization based on the summarizations of its child nodes. This summarization in turn can be reused later. In such case, we avoid the cost a traversing larger portion of the tree.

We now discuss function SUMMARIZE in details.

Base Cases: SUMMARIZE handles 4 base cases. First, when the symbolic state s is infeasible (line 3). Note that here path-sensitivity plays a role because provably infeasible paths will be excluded from contributing to the analysis result. Thus the returned witness is $\langle -\infty, [\,], false \rangle$. Second, s is a terminal state (line 5). Here Id refers to the identity function, which keep the program variables unchanged. The end point of a loop is treated similarly in the third base case (line 7). The last base case, lines 8-9, is the case that a summarization can be reused. We have discussed this step in Section III-B.

```
function COMPOSE($S_1, S_2$)
        Let $S_1$ be $[\ell_1, \Psi_1, \Gamma_1, \delta_1, \Delta_{p1}, \Delta_{c1}]$
        Let $S_2$ be $[\ell_2, \Psi_2, \Gamma_2, \delta_2, \Delta_{p2}, \Delta_{c2}]$
⟨21⟩ $\{\Gamma, \delta\}$ := COMBINE-WITNESSES($\Gamma_1, \Gamma_2, \delta_1, \delta_2$)
⟨22⟩ $\Delta_c$ := COMBINE-CACHES($\Delta_{c1}, \Delta_{c2}$)
⟨23⟩ $\Delta_p := \Delta_{p1} \ \wedge\ \Delta_{p2}$
⟨24⟩ $\Psi := \Psi_1 \ \wedge$ PRE-COND($\Delta_{p1}, \Psi_2$)
⟨25⟩ return $[\ell_1, \Psi, \Gamma, \delta, \Delta_p, \Delta_c]$
end function

function JOIN($c, S_1, S_2$)
        Let $S_1$ be $[\ell, \Psi_1, \Gamma_1, \delta_1, \Delta_{p1}, \Delta_{c1}]$
        Let $S_2$ be $[\ell, \Psi_2, \Gamma_2, \delta_2, \Delta_{p2}, \Delta_{c2}]$
⟨26⟩ $\{\Gamma, \delta\}$ = MERGE-WITNESSES($c, \Gamma_1, \Gamma_2, \delta_1, \delta_2$)
⟨27⟩ $\Delta_c$ := MERGE-CACHES($\Delta_{c1}, \Delta_{c2}$)
⟨28⟩ $\Delta_p := \Delta_{p1} \ \vee\ \Delta_{p2}$
⟨29⟩ $\Psi := \Psi_1 \ \wedge\ \Psi_2$
⟨30⟩ return $[\ell, \Psi, \Gamma, \delta, \Delta_p, \Delta_c]$
end function

function TRANSSTEP($s, \mathcal{P}, TransSet$)
        Let $s$ be $\langle \ell, \cdot, \cdot, \cdot \rangle$
⟨31⟩ $\overline{S} := [\ell, false, \langle 0, [\,], true \rangle, [\,], Id(Vars), [\,]]$
⟨32⟩ foreach ($tr \in TransSet$) do
⟨33⟩     $s \xrightarrow{tr} s'$
⟨34⟩     $S'$ := SUMMARIZE($s', \mathcal{P}$)
⟨35⟩     $S$ := COMPOSE(SUMMARIZE-A-TRANS($s, tr$), $S'$)
⟨36⟩     $\overline{S}$ := JOIN($c, \overline{S}, S$)
     endfor
⟨37⟩ return $\overline{S}$
end function

function SUMMARIZE-A-TRANS($s, tr$)
        Let $s$ be $\langle \ell, c, \sigma, \Pi \rangle$ and Let $tr$ be $\ell \xrightarrow{op} \ell'$
⟨38⟩ $t$ := EXECUTION-TIME($op$); $\Upsilon := acc(\ell, op, \sigma)$
⟨39⟩ Iterate through $\Upsilon$ and remove repeating accesses
⟨40⟩ $i := 0; \mathcal{M} := \emptyset$
⟨41⟩ foreach $m \in$ REVERSE($acc(\ell, op, \sigma)$) do
⟨42⟩     Add $\langle m, i \rangle$ into $\mathcal{M}$; $i := i + 1$
     endfor
⟨43⟩ $\Delta_c := \langle \mathcal{M}, i \rangle$
⟨44⟩ return $[\ell, true, \langle t, \Upsilon, [\![op]\!]_\sigma \rangle, [\,], op_\Delta, \Delta_c]$
end function
```

Fig. 4: Helper Functions

Expanding to the next programming point: Line 19 depicts the case when transitions can be taken from the current program point ℓ, and ℓ is not a loop head. We call TRANSSTEP to move recursively to next program points. TRANSSTEP considers all transitions emanating from ℓ, denoted as OUTGOING($\ell, \mathcal{P}$), then calls SUMMARIZE recursively and compounds the returned summarizations into a summarization of ℓ.

In more detail, for each tr in $TransSet$, TRANSSTEP extends the current state with the transition. We then call SUMMARIZE with the resulting child state (line 34). The algorithm aggregates each returned summarization into a single summarization, namely $\overline{S}$. This is achieved by first calling COMPOSE (line 35), then calling JOIN (line 36). Note here that we construct a summarization from a single transition before calling COMPOSE.

SUMMARIZE-A-TRANS computes a summarization for a single transition tr at state s. This can be seen as a basic step in our algorithm. Because no infeasible path has been discovered, the interpolant Ψ is just $true$. There is a single path, thus the dominating condition is $true$. We denote this as [ ], meaning that the cache is unconstrained. The cache abstract transformer is computed from the sequence of all memory accesses, namely $acc(\ell, op, \sigma)$. We delegate the discussion of it to the Appendix, i.e., Section IX-C. The abstract transformer Δ_p (for program variables) is the operation op itself, but translated to the language of input-output relation. As an example, x := x + 1 is translated to $x_{out} = x_{in} + 1$. We use op_Δ to denote such translated op.

We now elaborate on the computation of the witness. First, the static timing t is initialized as the static execution time of the operator op, assuming all memory accesses are cache $hits$. Secondly, Υ is initialized to $acc(\ell, op, \sigma)$. For consecutive accesses to a same memory block, only the first access is kept, the rest are removed from Υ. This can be achieved by iterating through Υ once. (Those removed accesses are classified as Always Hit.) The path constraints for the witness is computed by projecting op onto the set of program variables w.r.t. the symbolic store σ, denoted as $[\![op]\!]_\sigma$.

Handling Loops: Lines 11-18 handle the case when the current program point ℓ is a loop head. Let ENTRY$(\ell, \mathcal{P})$ denote the set of transitions going into the body of the loop, and EXIT$(\ell, \mathcal{P})$ denote the set of transitions exiting the loop.

Upon encountering a loop, our algorithm attempts to unroll it once by calling the function TRANSSTEP to explore the entry transitions (line 11). If the returned witness formula is *false*, meaning that it is infeasible to execute another iteration, we thus proceed with the exit branches. The returned summarization is merged (using JOIN) with the summarization of the previous unrolling attempt (line 13). Otherwise, we first use the returned abstract transformer to produce a new continuation context, (line 14 and 15), then we continue the analysis from the next loop iteration onwards (line 16). The returned information is then compounded with the summarization of the first iteration (line 17). Note that, importantly, compounded summarizations of the inner loop(s) can be reused in later iterations of the outer loop.

Finally, we conclude this section with a formal statement about the soundness of our framework.

Theorem 1 (Soundness)**.** *Our algorithm always produces* safe *WCET estimates.*

Proof Outline: Our algorithm performs a depth-first traversal of the symbolic execution tree. In all steps except when reuse happens, what we perform only widen the execution contexts, not narrowing them. Because of such steps, we might over-approximate the real WCET; but this is safe.

Assume that we reuse a summarization $[\ell, \Psi, \Gamma, \delta, \Delta_p, \Delta_c]$ of a subtree T at some symbolic state $s \equiv \langle \ell, c, [\![s]\!] \rangle$. Also assume that the reuse is *unsafe*. Note that when reuse happens, we employ the abstract transformers to generate a continuation context and continue the analysis from there. This step is also a widening step, thus it is safe. As a result, there must be a feasible path in the *avoided* subtree emanating from s, of which the timing is more than the timing of the witness Γ. Let us call this path Γ'.

Because the first condition for reuse implies that all infeasible paths of T stay infeasible under the new context s, Γ' must be feasible in T as well. Obviously, in order for Γ to be reported as the witness, in T, the timing of Γ' must be not more than the timing of Γ.

The third condition for reuse ensures that the dominating condition is satisfied. This implies that the cache configuration at s maintains the optimality of Γ. In particular, if the timing of Γ (in T) is not less than the timing of some other feasible path (in T), it is still the case under the new context s. Consequently, under the new context s, the timing of Γ' can not be more than the timing of Γ. Contradiction. $\square$

We remark here that we do not make use of the second condition for reuse in the proof of soundness. In fact, that condition has to do with the *precision* of reuse, rather than its soundness. An important implication – which has been shown in [6] – is that our algorithm produces "exact" analysis for loop-free programs.

VI. EXPERIMENTAL EVALUATION

The data and instruction cache settings in our experiments is borrowed from [8] for ARM9 target processor. Our instruction and data caches are separate. A cache state c contains two separate abstract caches $\langle c_i, c_d \rangle$, where c_i is a 4KB abstract instruction cache and c_d is a 4KB abstract data cache. The cache configurations are write-through, with no-write-allocate, 4-way set associative L1 cache with LRU replacement policy. The cache miss and cache hit latencies are respectively 10 and 0 cycles.

Because we perform loop unrolling, it is sufficient to employ a must analysis for precisely tracking the data cache, as opposed to a persistent analysis. We follow the treatment as in [9] for loading memory ranges into the cache for persistent analysis[3] when a data access cannot be resolved to a single memory address, meaning that the blocks in the memory address range are not loaded into the cache, but the blocks already in the cache are relocated as if all the blocks in the memory address range were loaded into the cache.

A. Results

We used an Intel Core i5 @ 3.2Ghz processor having 4Gb RAM for our experiments and built our system upon CLP($\mathcal{R}$) [11] and Z3 as the constraint solver, thus providing an accurate test for feasibility. The analysis was performed on LLVM IR which, while being expressive enough, a program's transition system can be easily constructed. The LLVM instructions are simulated for a RISC architecture. We use Clang 3.2 [12] to generate the IR.

Table I presents our results on three kinds of algorithms:

• **`AI+SAT⊕ILP`** implements the algorithm in [4]. It comprises micro-architectural modeling combined with an ILP formulation for WCET aggregation. This algorithm represents the state-of-the-art method.

• **`AI+SAT⊕Unroll_s`** implements a *hypothetical* algorithm. This analysis is constructed to benefit from combining the low-level analysis in [4] and the high-level analysis in [5]. This combined algorithm generates *static timing* for each basic block before aggregating results via a path analysis phase. More specifically, this algorithm improves on the previous because of loop unrolling and increased infeasible path detection.

[3]Huynh et. al. in [10] have fixed a safety issue with the treatment of loading memory ranges into the cache from [9]. However, this safety issue occurs in the semantics of abstract cache for persistent analysis and does not affect the semantics of abstract cache for must analysis, which is used by our method.

TABLE I: Comparing our Algorithm (Unroll_d) to the State-of-the-art

Benchmark	LLVM	AI+SAT $\oplus$ILP		AI+SAT $\oplus$Unroll_s		Unroll_d						Unroll_d vs	
						w. reuse			w.o. reuse			AI+SAT $\oplus$ILP	AI+SAT$\oplus$ Unroll_s
	LOC	T(s)	WCET	T(s)	WCET	T(s)	State	WCET	T(s)	State	WCET		
tcas	736	0.84	1427	9.07	1212	21.36	2389	1112	-	∞	-	22.07%	8.25%
nsichneu	12879	161.58	85845	504.88	66808	709.03	3776	48388	-	∞	-	43.63%	27.57%
statemate	3345	13.89	12382	248.41	9101	358.94	4152	7644	-	∞	-	38.27%	16.01%
ndes	1755	11.45	304369	37.95	174266	38.92	1065	148368	-	∞	-	51.25%	14.86%
fly-by-wire	2459	1.32	12171	10.97	9761	11.16	279	8751	-	∞	-	28.10%	10.35%
adpcm	2876	4.82	39088	106.53	33676	118.92	1617	31574	-	∞	-	19.22%	6.24%
compress	1334	9.18	478191	179.43	31665	204.82	1622	28670	911.38	10984	28180	94.00%	9.46%
edn	1226	1.47	437158	534.28	437158	676.11	2369	321028	-	∞	-	26.56%	26.56%
cnt	269	0.17	21935	0.29	21935	0.44	230	19355	1.56	1426	19355	11.76%	11.76%
matmult	286	1.75	874348	5.38	874348	6.5	906	621458	-	∞	-	28.92%	28.92%
jfdctint	693	0.08	20332	1.02	20332	1.43	254	17572	0.9	328	17572	13.57%	13.57%
fdct	831	0.08	17442	0.05	17442	0.13	58	14572	0.04	70	14572	16.45%	16.45%

• **Unroll_d** is the algorithm presented in this paper. This further improves on the already quite accurate hypothetical algorithm above because we now accomodate *dynamic timing*. Note, however, as explained in the earlier sections, that this entails more cost. Our results below show that this cost is bearable.

We have divided our benchmark programs, which are quite standard in evaluating WCET analysis algorithms, into three groups, separated by horizontal double lines. The columns **T(s)** and **State** denote the running time and number states (in symbolic execution) respectively. The symbol ∞ denotes out-of-memory. The WCET improvement is computed as $\frac{B-U}{B} \times 100\%$, where U is the WCET obtained using our analysis algorithm, and B is the WCET obtained using the baseline approach. In order to highlight the importance of reuse, we tabulate separate results for the cases where it is employed or not. The last two columns, separated by a vertical double line, summarize the improvement of Unroll_d over the other two analyses.

Benchmarks with lots of Infeasible Paths: The first group contains statemate and nsichneu from Mälardalen benchmarks [13] and tcas, a real life implementation of a safety critical embedded system. tcas is a loop-free program with *many* infeasible paths, which is used to illustrate the performance of our method in analyzing loop-free programs. On the other hand, nsichneu and statemate are programs which contain loops of big-sized bodies, also with many infeasible paths. These benchmarks are often used to evaluate the scalability of WCET analysis algorithms [14].

Standard Timing Analysis Benchmarks with Infeasible Paths: This group contains standard programs from [13], and fly-by-wire from [15].

Benchmarks with Simple Loops: This group contains a set of academic programs from [13]. Though the loops in these programs are simple for high-level analysis, they contain memory accesses that a fixed-point computation might resolve to a range of memory addresses, leading to imprecise low-level WCET analysis.

B. Discussion on Precision

The generated WCET by Unroll_d for the first group of benchmarks, compared to AI+SAT$\oplus$ILP, on average is improved by 34%; compared to AI+SAT$\oplus$Unroll_s, the number is 17%. Focussing on nsichneu and statemate, it can be seen that part of the improvement of Unroll_d over AI+SAT$\oplus$ILP comes from the detection of infeasible paths (i.e., the common improvement between Unroll_d and AI+SAT$\oplus$Unroll_s over AI+SAT$\oplus$ILP). The improvement of Unroll_d over AI+SAT$\oplus$Unroll_s, on the other hand, is due to infeasible path detection directly reflected in the tracking of micro-architectural states. This avoids lossy merging of cache states at the join points in the CFG.

For a loop-free program like tcas, the improvement of Unroll_d over the other two analyses is clearly not advantaged by tighter loop bounds in unrolling, nor disadvantaged by fixpoint computation in AI+SAT. Next, consider the fact that the (high-level) infeasible paths detected by Unroll_d and AI+SAT$\oplus$Unroll_s are the same. Even so, Unroll_d is more accurate by 8%. Once again, this improvement comes from our integration of low-level analysis with high-level analysis, making infeasible path detection reflected in the precise tracking of micro-architectural states.

For benchmarks in the second group, Unroll_d produces significantly more accurate WCET than AI+SAT$\oplus$ILP, on average 48%, peaking at 94%. In compress and ndes, many infeasible paths have to do with loops, and being able to detect them improves the WCET estimates dramatically. AI+SAT$\oplus$Unroll_s performs relatively well on this group of benchmarks. However, for ndes and fly-by-wire, the accuracy improvement of Unroll_d over AI+SAT$\oplus$Unroll_s is still noticeable. Further investigation reveals that these two benchmarks contain memory accesses which are resolved to address ranges in the AI+SAT component – ultimately is still a fixed-point computation – leading to imprecise analysis results from the combined algorithm.

The effect of such memory accesses on analysis precision can be seen more clearly by examining the third benchmark group. Unroll_d is still better than the other two algorithms by 18% on average. These benchmarks do not contain many infeasible paths nor complicated loops and that is the reason why AI+SAT$\oplus$Unroll_s does not produce better estimates than AI+SAT$\oplus$ILP. However, these benchmarks contain memory accesses which are resolved to address ranges in a fixed-point computation, leading to the imprecision of AI+SAT. In contrast, Unroll_d performs loop unrolling, thus it can precisely resolve the addresses of the accesses, leading to superior precision.

In summary, in terms of precision, Unroll_d outperforms the other two algorithms in all benchmarks. The WCET estimations from Unroll_d have improved 32% on average compared to AI+SAT$\oplus$ILP and 15% on average compared to AI+SAT$\oplus$Unroll_s. These improvements clearly uphold our proposal that performing WCET analysis in one integrated phase in the presence of *dynamic timing* will enhance the precision over modular approaches. However, the scalability of our method is not yet discussed.

C. Discussion on Scalability

As expected, reuse is important for scalability. For most of the benchmarks (8 out of 12) the analysis cannot finish without reuse. Between the benchmarks in the first group which contain many infeasible paths (`tcas`, `nsichneu` and `statemate`), none of the benchmarks can be analyzed without reuse. The two largest benchmarks, `nsichneu` and `statemate`, are used as an indicator of the scalability of the WCET tools. The WCET analysis for `nsichneu` and `statemate`, uses at most 53% and 40% of the 4GB available. It is worth noting that, for `nsichneu`, the overhead of the analysis time and memory usage compared to AI+SAT$\oplus$`Unroll_s` is 31% and 40%, respectively, while the precision is improved by 27%.

In conclusion, our analysis framework relies a lot on reuse for scalability. From these experiments we can infer that only small size programs where the number of paths is limited can be analyzed without reuse.

VII. Related Work

WCET analysis has been the subject of much research, and substantial progress has been made in the area (see [14], [16] for surveys of WCET). As discussed before, WCET analysis is often conducted by separating low-level analysis and high-level analysis into different phases.

High-level analysis: Among the works on high-level analysis, our most important related work is [5]. The origin of this approach dates back to [6], which introduced the concept of summarization with interpolation, to harness better "reuse" in the setting of dynamic programming and address the scalability issue of the resource-constrained shortest path (RCSP) problem. RCSP, though NP-hard, is still simpler than WCET analysis. In [6], reuse was limited to loop-free programs.

Chu and Jaffar [5] have advanced [6] by introducing *compounded* summarizations, so that reuse can be effective in the presence of loops and nested loops. Specifically, [5] has demonstrated that exhaustive symbolic execution for WCET analysis can be made scalable. Given the effect of caches on the basic block timings, making the timings dynamic, [5] is no longer applicable. One key contribution of this paper is that, by capturing the *dominating condition*, we enable reuse, now under the existence of caches.

Recently, there are CEGAR-like methods, which start by generating a rough WCET estimate and then gradually refine it. "WCET squeezing" [17] is built on top of the Implicit Path Enumeration Technique (IPET) [18]. A solution to the given integer linear programming (ILP) formula corresponds to number of program traces, of which the feasibility will be checked (one-by-one) via SMT solving. If such a trace is infeasible, additional ILP constraints are added to exclude it from further consideration. Subsequently, [19] proposes hierarchical segment abstraction, thus allows the computation of WCET by solving a number of independent ILP problems, instead of one large *global* ILP problem. Since the abstract segment trees can store more *expressive* constraints than ILP, better refinement procedure can be implemented.

We also mention the recent work [20], which also employs the concept of interpolation, but under the SMT framework, to avoid state explosion in WCET analysis. Like [6], this approach is formulated for loop-free programs, and not yet suitable for analyzing programs with loops.

In summary, we can see a trend of research where recent advances in software verification are employed for WCET high-level analysis. However, it is unclear if these approaches will remain scalable when extended towards low-level analysis, under the presence of loops and/or many infeasible paths.

Low-level analysis: Low-level analysis, with emphasis on caches, has always been an active research topic in WCET analysis. Initial work on instruction cache modeling uses integer linear programming (ILP) [21]. However, the work does not scale due to a huge number of generated ILP constraints. Subsequently, the abstract interpretation framework (AI) [22] for low-level analysis, proposed in [2], has made an important step towards scalability. The solution has also been applied in commercial WCET tools (e.g., [23]). For most existing WCET analyzers, AI framework has emerged to be the basic approach used for low-level analysis. Additionally, static timing analysis with data cache has been investigated in [9], [10], [24].

Recent approaches [3], [4] by the same research group – combining AI with verification technology – have shown some promising results. In the more recent work [4], a partial path is tracked together with each micro-architectural state μ. This partial path captures a subset of the control flow edges along which the micro-architectural state μ has been propagated. If a partial path was infeasible, its associated micro-architectural state can be excluded from consideration. To be tractable, micro-architectural states are merged at appropriate sink nodes. (In fact, the partial path constraints are merged to $true$.) As a result, the approach is only effective for detecting infeasible paths whose conflicting branch conditions appeared relatively close to each other in the CFG.

In a similar spirit as [17] and [3], Nagar and Srikant [25] propose the concept of *cache miss paths*. The method employs IPET formulation, using the information from the worst-case solution of the ILP problem (which corresponds to a number of program paths) to improve the precision of AI-based cache analysis. However, it is reported that for benchmarks `statemate` and `nsichneu` – which contain a large number of program paths – little improvement is obtained.

It is important to note that, in general, the above-mentioned approaches still employ a fixed-point computation in order to ensure sound analysis across loop iterations. Thus, they inherit the imprecision of AI, because the timings of a basic block in different iterations of a loop often can diverge significantly.

Other Related Work: We mention some orthogonal works that represent recent and interesting advances in WCET research. [26] performs loop unrolling, passing flow information from source level through the process of compiler optimization in order to help tighten the WCET estimates. At the current stage, the approach seems to be limited to single-path programs. Zolda and Kirner [27] propose to incorporate the information from collected program execution traces into IPET framework to enhance the precision of calculated WCET bounds. The effectiveness of this approach seems dependent on the quality of the *collected* traces as well as the amount of infeasible paths in the given input program.

We remark that the idea of coupling low-level analysis with high-level analysis (with loop unrolling) dates back to [28]. However, to counter state explosion, the only solution of [28] is to perform merging frequently. In the end, the approach forfeits its intended precision, while at the same time, does not scale realistic benchmarks.

Finally, we remark on the issue of *timing anomaly* [29]. In general, timing anomaly can make abstraction (and therefore AI) *unsound*. It is extremely hard to systematically address this issue. More often, custom solutions are employed. For example, [30] can compute a constant bound to be added to the local worst-case path to safely handle timing anomalies, provided they are not of "domino-effect" type. This approach is also applicable to us. Extension towards integrating such method (or the alike) is left as future work.

VIII. Conclusion

We have presented a framework for WCET analysis of programs with consideration of a cache micro-architecture. At its core is a symbolic execution algorithm. Its key feature is the ability for *reuse*. This is critical for maintaining a high-level of path-sensitivity, which in turn produces significantly increased accuracy. In other words, reuse allows scalability in path-sensitive exploration. Finally, we demonstrated using realistic benchmarks.

References

[1] F. Mehnert, M. Hohmuth, and H. Hartig, "Cost and benefit of separate address spaces in real-time operating systems," in *RTSS 2002*.

[2] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by seperate cache and path analyses," *RTS 18(2/3) 2000*.

[3] S. Chattopadhyay and A. Roychoudhury, "Scalable and precise refinement of cache timing analysis via model checking," in *RTSS 2011*.

[4] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, "Precise micro-architectural modeling for wcet analysis via ai+sat," in *RTAS 2013*.

[5] D. H. Chu and J. Jaffar, "Symbolic simulation on complicated loops for wcet path analysis," in *EMSOFT 2011*.

[6] J. Jaffar, A. E. Santosa, and R. Voicu, "Efficient memoization for dynamic programming with ad-hoc constraints," in *AAAI 2008*.

[7] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *POPL, Pages 84–96, 1978*.

[8] "Wcet tool competition 2014," URL www.mrtc.mdh.se/projects/WTC/.

[9] C. Ferdinand and R. Wilhelm, "On predicting data cache behavior for real-time systems," in *LCTES 1998*.

[10] B. K. Huynh, L. Ju, and A. Roychoudhury, "Scope-aware data cache analysis for WCET estimation," in *RTAS 2011*.

[11] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap, "The CLP($\mathcal{R}$) language and system," *ACM TOPLAS 14(3) 1992*.

[12] "clang: a c language family front-end for llvm," http://www.clang.llvm.org, 2014, accessed: 2015-02-01.

[13] "The malardalen wcet benchmarks," http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2014.

[14] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *TECS 7(3) 2008*.

[15] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel, "Papabench: a free real-time benchmark," in *WCET 2006*.

[16] P. Puschner and A. Burns, "A review of worst-case execution-time analysis," *RTS 18(2/3) 2000*.

[17] J. Knoop, L. Kovács, and J. Zwirchmayr, "Wcet squeezing: on-demand feasibility refinement for proven precise wcet-bounds," in *RTNS 2013*.

[18] Y. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *DAC 1995*.

[19] P. Černý *et al.*, "Segment abstraction for worst-case execution time analysis," in *ESOP 2015*.

[20] J. Henry, M. Asavoae, D. Monniaux, and C. Maïza, "How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics," in *LCTES 2014*.

[21] Y.-T. S. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," *TODAES 4(3)1999*.

[22] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis," in *POPL 1977*.

[23] "aiT Worst-Case Execution Time Analyzers," URL `http://www.absint.com/ait/index.htm`.

[24] R. T. White, F. Mueller, C. A. Healy, and o. Whalley, "Timing analysis for data caches and set-associative caches," in *RTAS 1997*.

[25] K. Nagar and Y. Srikant, "Path sensitive cache analysis using cache miss paths," in *VMCAI 2015*.

[26] H. Li *et al.*, "Tracing flow information for tighter wcet estimation: Application to vectorization," in *RTCSA 2015*.

[27] M. Zolda and R. Kirner, "Calculating wcet estimates from timed traces," *RTS*, pp. 1–50, 2015.

[28] T. Lundqvist and P. Stenström, "An integrated path and timing analysis method based on cycle-level symbolic execution," *RTS 17(23) 1999*.

[29] J. Reineke *et al.*, "A definition and classification of timing anomalies." *WCET 2006*.

[30] J. Reineke and R. Sen, "Sound and efficient wcet analysis in the presence of timing anomalies," in *WCET 2009*.

IX. Appendix

A. Standard Definitions Related to Loops

(Note that our transition system is a directed graph.)

Definition 4 (Loop). *Given a directed graph $G = (V, E)$ (our transition system), we call a strongly connected component $S = (V_S, E_S)$ in G with $|E_S| > 0$, a loop of G.*

Definition 5 (Loop Head). *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$ of G, we call $\mathcal{E} \in V_L$ a loop head of L, also denoted by $E(L)$, if no node in V_L, other than $\mathcal{E}$ has a direct successor outside L.*

Definition 6 (End Point of Loop Body). *Given a directed graph $G = (V, E)$, a loop $L = (V_L, E_L)$ of G and its loop head $\mathcal{E}$. We say that a node $u \in V_L$ is an end point of a loop body if there exists an edge $(u, \mathcal{E}) \in E_L$.*

Definition 7 (Same Nesting Level). *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$, we say two nodes u and v are in the same nesting level if for each loop $L = (V_L, E_L)$ of G, $u \in V_L \iff v \in V_L$.*

B. Generating Witness and Dominating Condition

In this section, we present Combine-Witnesses and Merge-Witnesses functions.

In Fig. 5, Combine-Witnesses produces a witness and a dominating condition, by compounding the witnesses and dominating conditions of two subtrees, where one suffixes the other. This can be understood as a *sequential* composition.

```
function COMBINE-WITNESSES($\Gamma_1, \Gamma_2, \delta_1, \delta_2$)
      Let $\Gamma_1$ be $\langle t_1, \Upsilon_1, \pi_1\rangle$ and Let $\Gamma_2$ be $\langle t_2, \Upsilon_2, \pi_2\rangle$
⟨45⟩ $t = t_1 + t_2$
⟨46⟩ if (LAST($\Upsilon_1$) $\equiv$ FIRST($\Upsilon_2$)) then $\Upsilon_2$ := REMOVE-FIRST($\Upsilon_2$)
⟨47⟩ $\Upsilon = \Upsilon_1 \cdot \Upsilon_2$
⟨48⟩ $\pi := \pi_1 \wedge \pi_2$
⟨49⟩ $\delta_2'$ := PRE-CACHE($\Upsilon_1, \delta_2$); $\delta = \delta_1 \wedge \delta_2'$
⟨50⟩ return $\{\langle t, \Upsilon, \pi\rangle, \delta\}$
end function
```

Fig. 5: Combining Witnesses

The static timing of the witness t is initialized as the sum of t_1 and t_2 (line 45). Let m be the last access in Υ_1. If m is also the first access in Υ_2, it will be removed from Υ_2 (line 46). The combined Υ is then the concatenation of Υ_1 and Υ_2 (line 47). Next, the witness path constraint π is computed as the conjunction of π_1 and π_2 (line 48).

The combined dominating condition δ is computed as the conjunction of δ_1 and a condition, say δ_2', as in line 49. Intuitively, δ_2' describes an abstract cache state c, such that if from c we perform all the accesses in Υ_1, we will produce a cache state c' which satisfies δ_2. The computation of δ_2' is a precondition computation, but in the nature of caches. We abstract this computation with the function Pre-Cache. We omit the details here, but remark that in our implementation we need a more elaborate representation of cache constraints in order to facilitate such computation. In particular, $m \in cache$ is represented as $age(m) < A$ while $m \notin cache$ is represented as $age(m) \geq A$ (assuming that age starts from 0).

In Fig. 6, Merge-Witnesses produces a witness and a dominating condition, by compounding the witnesses and dominating conditions of two sibling subtrees. We need to choose one witness from the two input witnesses. The combined dominating condition must ensure the dominance of each witness (in its respective subtree) and the dominance of the chosen witness over the other.

The dominating condition δ is initialized as the conjunction of the two dominating conditions (line 51). We next compare the timing of the two witnesses; and we select the one with higher timing as the combined witness. After line 53, the chosen witness and its corresponding dominating condition are captured in Γ_1 and δ_1.

Next, we test if δ is sufficient to ensure that Γ_1 dominates Γ_2. Given a condition δ, a witness *dominates* another witness if its minimum timing is more than the maximum timing of the other. The minimum timing is calculated by: (1) first determine some accesses in the Υ component are necessary misses as the consequence of the condition δ; (2) classifying the remaining accesses in Υ as cache hits. Whereas the maximum timing is calculated in the opposite manner: (1') first determine some accesses in the Υ component are necessary hits as the consequence of the condition δ; (2') classifying the remaining accesses in Υ as cache misses. This *dominance test* is shown in line 54.

If Γ_1 dominates Γ_2, then Γ_1 is returned as the dominating witness with δ as the dominating condition. If not, we need to further constrain the dominating condition δ.

First, for each access m_i in Υ_1, if m_i has not been constrained in δ, $m_i \notin cache$ is added to δ (lines 58). This cache constraint might increase the the minimum timing of Γ_1 and lead to passing the dominance test. If the dominance test indeed succeeds, Γ_1 and δ are returned.

If we have not succeeded yet, we can do similarly for each m_j in Υ_2. Note the difference that now we add the cache constraint of the form $m_j \in cache$, with the hope to reduce the maximum timing of Γ_2 enough that the dominance test can be passed (line 64).

At the end of the first **for** loop, Min(Υ_1, δ) would be larger than (or equal to) the original timing of Γ_1 (w.r.t. cache context c) while at the end of the second **for** loop, Max(Υ_2, δ) would be less than (or equal to) the original timing of Γ_2 (w.r.t. cache context c). In other words, eventually, we will end up with a condition δ so that Γ_1 dominates Γ_2.

```
function MERGE-WITNESSES($c, \Gamma_1, \Gamma_2, \delta_1, \delta_2$)
      Let $\Gamma_1$ be $\langle t_1, \Upsilon_1, \pi_1\rangle$ and Let $\Gamma_2$ be $\langle t_2, \Upsilon_2, \pi_2\rangle$
⟨51⟩ $\delta := \delta_1 \wedge \delta_2$
⟨52⟩ if (COMPUTE-TIMING($\Gamma_1, c$) < COMPUTE-TIMING($\Gamma_2, c$))
⟨53⟩    SWAP($\Gamma_1, \Gamma_2$), SWAP($\delta_1, \delta_2$)
⟨54⟩ if ($t_1$ + MIN-TIME($\Upsilon_1, \delta$) $\geq t_2$ + MAX-TIME($\Upsilon_2, \delta$))
⟨55⟩    return $\{\Gamma_1, \delta\}$
⟨56⟩ foreach $m_i \in \Upsilon_1$ do
⟨57⟩    if (NO-CONS($m_i, \delta$))
⟨58⟩       $\delta := \delta \wedge \{m_i \notin cache\}$
⟨59⟩       if ($t_1$ + MIN-TIME($\Upsilon_1, \delta$) $\geq t_2$ + MAX-TIME($\Upsilon_2, \delta$))
⟨60⟩          return $\{\Gamma_1, \delta\}$
      endfor
⟨61⟩ foreach $m_j \in \Upsilon_2$ do
⟨62⟩    if (NO-CONS($m_j, \delta$))
⟨63⟩       $\delta := \delta \wedge \{m_j \in cache\}$
⟨64⟩       if ($t_1$ + MIN-TIME($\Upsilon_1, \delta$) $\geq t_2$ + MAX-TIME($\Upsilon_2, \delta$))
⟨65⟩          return $\{\Gamma_1, \delta\}$
      endfor
end function
```

Fig. 6: Merging Witnesses

```
function COMBINE-CACHES($\Delta_{s1}$,$\Delta_{s2}$)
      Let $\Delta_{s1}$ be $\langle \mathcal{M}_1, n_1 \rangle$ and Let $\Delta_{s2}$ be $\langle \mathcal{M}_2, n_2, \rangle$
⟨66⟩ $\mathcal{M} := \mathcal{M}_2; n := 0$
⟨67⟩ foreach $\langle m, k \rangle \in \mathcal{M}_1$ do
⟨68⟩     foreach $\langle m', i \rangle \in \mathcal{M}_2$ do
⟨69⟩         if $m \not\equiv m'$ then
⟨70⟩             Increase the age of $m'$ in $\mathcal{M}$ by 1
             else
⟨71⟩             Move $\langle m, k \rangle$ to the beginning of $\mathcal{M}$
⟨72⟩             break
     endfor
⟨73⟩     if $\langle m, k \rangle \notin \mathcal{M}$ then
⟨74⟩         Add $\langle m, 0 \rangle$ to the beginning of $\mathcal{M}$
⟨75⟩         $n := n + 1$
 endfor
⟨76⟩ foreach $\langle m, i \rangle \in \mathcal{M}$ do
⟨77⟩     if $i \geq A - 1$ then
⟨78⟩         Remove $\langle m, i \rangle$ from $\mathcal{M}$; $n := A$
 endfor
⟨79⟩ return $\langle \mathcal{M}, n \rangle$
end function

function MERGE-CACHES($\Delta_{s1}$,$\Delta_{s2}$)
      Let $\Delta_{s1}$ be $\langle \mathcal{M}_1, n_1 \rangle$ and Let $\Delta_{s2}$ be $\langle \mathcal{M}_2, n_2 \rangle$
⟨80⟩ $\mathcal{M} := \emptyset$
⟨81⟩ foreach $\langle m, i \rangle \in \mathcal{M}_1 \wedge \langle m, j \rangle \in \mathcal{M}_2$ do
⟨82⟩     $\mathcal{M} := \mathcal{M} + \langle m, \text{MAX}(i, j) \rangle$
 endfor
⟨83⟩ return $\langle \mathcal{M}, \text{MAX}(n_1, n_2) \rangle$
end function
```

Fig. 7: Combining and Merging Two Set Summaries

C. Generating Abstract Transformer for Cache

Let us first review on abstract set-associative must-cache. An abstract set-associative cache c is consisted of N cache sets, where $N = C/(BS * A)$, C is the cache capacity, BS is block size and A is the associativity. We denote a cache-set with cs where $c \equiv [cs_1, ..., cs_N]$. Each cache set is considered as a set of cache lines $cs = [l_1, ..., l_A]$. We use $cs(l_i) = m$ to indicate the presence of a memory block m in a cache-set, where i describes the *relative age* of the memory block according to the LRU replacement strategy and not the physical position in the cache hardware.

The cache abstract transformer Δ_c is partitioned to N independent abstract transformers of respective cache-sets, i.e., $\Delta_c \equiv [\Delta_{s_0}, ...\Delta_{s_{N-1}}]$. Applying a cache abstract transformer on a cache state, each abstract transformer of a cache-set is applied to the corresponding cache-set.

Each abstract transformer of a particular cache-set is depicted by $\langle \mathcal{M}, n \rangle$, where $\mathcal{M}$ is a sequence of pairs $\langle m, i \rangle$. Each pair indicates a memory block m and its age i indicating where m will be loaded to the cache. Moreover, n depicts the number of cache lines that the memory blocks in $\mathcal{M}$ are loaded to. It is the maximum i in the sequence $\mathcal{M}$ plus 1. Thus, n is always less than A.

The size of cache abstract transformer is *linear* w.r.t. the cache capacity. This is because in computing the abstract transformer, we only store the memory blocks with a age less than the *associativity*. The rest of the memory blocks would naturally be pushed out of the cache and we do not need to maintain them in the abstract transformer.

We present COMBINE-CACHES and MERGE-CACHES functions in Fig. 7. We only focus on demonstrating how to compound two abstract transformers for a particular cache-set. The generalization to deal with a whole cache is straightforward.

Consider COMBINE-CACHES. $\mathcal{M}$ is first initialized to $\mathcal{M}_2$. For each m in $\mathcal{M}_1$, for each memory accesses m' in $\mathcal{M}_2$, if they are not the same, the age of m' is increased by 1 (lines 69-70). This process terminates if one of the memory accesses is the same as m. The memory access is moved to the beginning of $\mathcal{M}$ with age 0 (line 71). Next, if m is not in $\mathcal{M}$, it is added to the beginning of $\mathcal{M}$ with age 0 (lines 73-75).

Finally, for all pair $\langle m, i \rangle$ in $\mathcal{M}$, where i denotes the age of m, if $i \geq A$, then $\langle m, i \rangle$ is removed from $\mathcal{M}$ (lines 76-78).

MERGE-CACHES preserves the memory blocks common on both abstract transformers with their maximum age. For each memory block m that is in both $\mathcal{M}_1$ and $\mathcal{M}_2$, it is added to $\mathcal{M}$ with the maximum age from $\mathcal{M}_1$ and $\mathcal{M}_2$. We compute n as the maximum of n_1 and n_2.

Example 2. *Consider a sequence of accesses $\langle m_1, m_2, m_3, m_2 \rangle$ and a fully associative cache of size 4 which initially contains m_0:*

0	*1*	*2*	*3*
m_0			

Applying the sequence $\langle m_1, m_2, m_3, m_2 \rangle$ to the given cache state, we achieve:

0	*1*	*2*	*3*
m_2	m_3	m_1	m_0

The abstract transformer we compute for the previous sequence of accesses would be of the form $\langle [\langle m_2, 0 \rangle, \langle m_3, 1 \rangle, \langle m_1, 2 \rangle], 3 \rangle$. The application of this transformer to the initial cache state is as follows:

First, all items in the initial cache are aged by 3 – the number of cache lines that memory blocks will be loaded to. This gives us:

0	*1*	*2*	*3*
			m_0

Next, for each pair among $\langle m_2, 0 \rangle$, $\langle m_3, 1 \rangle$, $\langle m_1, 2 \rangle$, the respective memory blocks (m_2, m_3 and m_1) are loaded into the cache at their ages (0, 1 and 2), respectively. The generated cache state after applying the abstract transformer on the initial cache is the same as if we had loaded the memory blocks one by one:

0	*1*	*2*	*3*
m_2	m_3	m_1	m_0