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—Worst-Case Execution Time (WCET) is a reliable guarantee for the temporal correctness of hard real-time systems. In this paper, we propose a novel integrated method for WCET analysis where micro-architectural modeling – with emphasis on caches – and systematic path-sensitivity, are synergized. This would give us very high precision for WCET analysis, but at the same time, it is a huge challenge for scalability. Our contribution in this paper is a dynamic programming algorithm with a powerful concept of reuse. Reuse in turn depends on the core concepts of interpolation and dominance. While interpolation-based methods have been used in program verification for the purpose of pruning the search space of symbolic execution, our setting is novel not just because we are performing analysis instead of verification, but because our interpolation with dominance covers reuse under an environment where the timing of program paths are dynamic. In the end, because we are systematically path-sensitive, our algorithm is the most accurate. The important point, however, is that it also can scale to a reasonable level. Our realistic benchmarks will show both aspects: that systematic path-sensitivity in fact brings significant accuracy gains, and also that the algorithm 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. However, performance enhancing features of a processor (e.g. caches) make WCET analysis a difficult problem.

Traditionally, WCET analysis is proceeded in three phases. The first phase, referred to as low-level analysis, often involves micro-architectural modeling to accurately determine the maximum execution time of the basic blocks. The second phase concerns a program level path analysis to determine the infeasible paths in the program’s control flow graph (CFG). The third phase, or the aggregation phase, combines the results of micro-architectural modeling and path analysis. While in some recent approaches, the second and third phases are often fused into one, called generally as high-level analysis; low-level analysis and high-level analysis are still separately performed for the reason of scalability.

In low-level analysis, we need to consider timing effects of performance enhancing processor features such as pipeline and caches. This paper focuses on caches, since caches impact on the real-time behavior of programs much more than other features [1]. Cache analysis – to be scalable – is usually accomplished by abstract interpretation (AI) [2]. In other words, 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 this:

• The cache states are joined at the control flow merge points. This often results in an over-estimation of potential cache misses.

• As proposed by [2], loops are virtually unrolled once before fixed point computation in order to accommodate the persistent behavior of instruction cache when the loops are relatively small compared to the cache size. Beyond this one time virtual unrolling, AI framework is unable to give different timings for a basic block, executed in different iterations of a loop.

A direct improvement attempt would first be to curtail the above-mentioned merge points. That is, when traversing the CFG from a particular source node to a particular sink node, do not visit any intermediate node which is unreachable from the source node, and do not perform merging at the other intermediate nodes. Only perform merging once traversals are finished, 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. That is, they demonstrated that the accuracy of WCET estimates can be improved by integrating some forms of infeasible path discovery into micro-architectural modeling. We note, however, such addition of path-sensitivity is quite limited. In [4], at any program point, the approach tracks a partial path with each micro-architectural state \( \mu \). This partial path captures a subset of all the control flow edges along which the micro-architectural state \( \mu \) has been propagated. The partial path is defined as a propositional logic formula \( \phi \) over the propositions associated with each control flow edge. If the partial path was infeasible, its associated micro-architectural state can be ignored for further consideration. To be tractable, the approach merges different micro-architectural states at appropriate sink nodes\(^1\). By merging, the growth in the number of micro-architectural states can be bounded efficiently. However, a direct implication is that the approach is only effective for detecting infeasible paths whose conflicting branch conditions appeared relatively close to each other in the CFG.

More importantly, in the literature in general, the algorithms still employ a fixed point computation in order to aggregate the analysis across loop iterations. Thus, they still inherit the imprecision from an AI framework, identified as point (2) in the above. More specifically, 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.

In this paper, we propose a symbolic execution framework where micro-architectural modeling and systematic path-sensitivity are synergized. In our algorithm, loops are unrolled fully\(^2\) and summarized. The essence of our proposed method

\(^1\)The partial paths are merged to true.

\(^2\)We should note that the loop unrolling in our algorithm is done virtually and not physically, and is different from the loop unrolling in compilers.
We assume a direct-mapped cache, where block \( a \) and block \( b \) access the memory block \( m_1 \) and \( m_2 \) respectively, and \( m_1 \) and \( m_2 \) map to the same cache set. In other words, the memory blocks \( m_1 \) and \( m_2 \) conflict in the cache. Note that the execution of block \( a \) and block \( b \) are feasible, though in different iterations. A pure \( \Delta t \) approach such as [2] will have to conservatively declare that the fixed point must cache at the looping point contains neither \( m_1 \) nor \( m_2 \).

On the other hand, [4] might improve the analysis by ignoring some “simple” infeasible states from being considered. For example, if the conditional expression was \( (i < 50) \), [4] could discover that an execution of \( b \) cannot be followed by an execution of \( a \) in the next iteration. In other words, it could discover that the access \( m_2 \) is indeed persistent [5], [6]. Now consider the case that the conditional expression is a bit more complicated, for example, \( i \% 5 > 1 \). Without knowing the precise value of \( i \), after executing block \( a \) (or block \( b \)) in the current iteration, it is possible to either execute block \( a \) or block \( b \) in the next iteration. In such a case, a fixed point method, equipped with some form of path-sensitivity, will not be of much help to improve the analysis precision.

In our analysis, we precisely capture the value of \( i \) throughout the analysis process. Consequently, we are able to disregard all infeasible states from consideration, thus achieving more accurate analysis result, comparing to the state-of-the-art.

The precision of our framework comes at the cost of scalability. Clearly, any framework which attempts the full exploration of the symbolic execution tree will not scale. The first attempt to make such approaches scalable was [7], where the concept of summarization with interpolation was used for analysis on the specific problem of resource-constrained shortest path (RCSP), which is simpler (though NP-hard) than WCET. The problem was formulated such that the scalability could be achieved by reusing the summarizations of previously computed sub-problems. After analyzing the paths in a subtree, an interpolant, preserving the discovered infeasible paths in the analyzed subtree, and a representative (witness) formula preserving the (sub)-results of the subtree was computed and stored. In case the node was encountered in another path, such that its context entailed the previously computed interpolant and representative formula, the paths emerging from that node were not explored. This step was called reuse and the subsumed node would share the analysis results of the subsuming node. Otherwise, the symbolic execution would naturally explore all the paths emerging from that node. In other words, this was a generalized form of dynamic programming.

**Example 1.** Let us consider the academic example in Fig. 1. We assume a direct-mapped cache, where block \( a \) and block \( b \) access the memory block \( m_1 \) and \( m_2 \) respectively, and \( m_1 \) and \( m_2 \) map to the same cache set. In other words, the memory blocks \( m_1 \) and \( m_2 \) conflict in the cache. Note that the execution of block \( a \) and block \( b \) are feasible, though in different iterations. A pure \( \Delta t \) approach such as [2] will have to conservatively declare that the fixed point must cache at the looping point contains neither \( m_1 \) nor \( m_2 \).

On the other hand, [4] might improve the analysis by ignoring some “simple” infeasible states from being considered. For example, if the conditional expression was \( (i < 50) \), [4] could discover that an execution of \( b \) cannot be followed by an execution of \( a \) in the next iteration. In other words, it could discover that the access \( m_2 \) is indeed persistent [5], [6]. Now consider the case that the conditional expression is a bit more complicated, for example, \( i \% 5 > 1 \). Without knowing the precise value of \( i \), after executing block \( a \) (or block \( b \)) in the current iteration, it is possible to either execute block \( a \) or block \( b \) in the next iteration. In such a case, a fixed point method, equipped with some form of path-sensitivity, will not be of much help to improve the analysis precision.

In our analysis, we precisely capture the value of \( i \) throughout the analysis process. Consequently, we are able to disregard all infeasible states from consideration, thus achieving more accurate analysis result, comparing to the state-of-the-art.

The next step was taken in [8], [9], enhancing reuse of summarizations for WCET analysis of programs with loops. In this modular approach the timings of the basic blocks could be generated from the state-of-the-art low-level analysis. In the analysis, loops were fully unrolled and the analysis was made scalable by introducing compounded summarizations, where one or more loop iterations were summarized and reused for subsuming other loop iterations.

**Example 2.** Figure 2 (a), informally depicts the symbolic execution tree for a program. Each triangle presents one subtree in the execution tree. In the beginning of the analysis the context reaching the subtree on the left is respectively \( \Psi_0 \). The analysis from [8], would first analyze the subtree on the left. After finishing the analysis of the left subtree, the WCET of the subtree is generated which is 15. The longest feasible path in the subtree is named as the representative path (marked with blue color). Due to the path sensitivity of the analysis, the infeasible path (marked with red color and a cross) is ignored from consideration. After finishing the analysis of the left subtree, an interpolant \( \Psi \) is generated and stored in the summarization of the root of the left subtree.

Moving to the right subtree, the incoming context \( \Psi_2 \) is checked against the interpolant. Since, the set of the infeasible paths in the right subtree remain the same and the representative path is still feasible in the right subtree, the summarization of the left subtree is reused to generate the WCET of the right subtree, which is 15. Continuing the analysis, at the root, the path on the right is the longest path in the tree and the WCET of the program is 22. Note that the WCET of the program was generated without even exploring the longest feasible path in the tree.

However, the reuse of the summarizations was limited to loop-free programs.

The core concept of reuse in [8] was based on this idea that the contribution of each basic block in the computation of the timing of a path is constant. However, in the presence of micro-architectural features such as caches, the contribution of each basic block is no longer constant. We define this concept as dynamic timing model. As a result the summarizations can not be reused anymore.

In this paper, we extend the concept of reuse in the presence of the dynamic timing model. The main contribution of this paper is then, furnishing the concept of reuse with means to dynamically generate the WCET of a subtree based on the stored summarization and the cache state reaching a node. For this to happen the representative path should dominates all the other paths in the presence of the dynamic timing model. As a consequence, the safety of reuse is guarded not only by the concept of interpolation, but also by the concept of dominance. By that, we mean, given a new symbolic state, it is sound to reuse a summarization if first the interpolant is satisfied which ensures all the discovered infeasible paths
are maintained at the reuse point, and second a dominating condition is satisfied which ensures the cache state reaching the reuse point is similar enough to the cache state of the subsuming node.

**Example 3.** The analysis of the program from the previous example, in the presence of the dynamic timing model, is presented in Figure 2(b). In the beginning of the analysis the context and the cache state reaching the subtree on the left are respectively $\Psi_0$ and $c_0$. The subtree on the left is fully analyzed. After finishing the analysis of the left subtree, the WCET of the subtree is generated based on the incoming cache state ($c_0$) which is similarly 15. In similar fashion, the representative path, which is the longest feasible path in the subtree, is marked with blue color and due to the path sensitivity of the analysis the infeasible path (marked with red color and a cross) is ignored from consideration. After finishing the analysis of the left subtree, we will generate an interpolant $\Psi$ and a dominating condition $\tau$, and we store them in the summarization of the root of the left subtree along side the representative path.

Moving to the right subtree, the incoming context $\Psi_2$ and the cache state $c_1$ are checked against the interpolant and the dominating condition. Since, the set of the infeasible paths in the right subtree remains the same and the incoming cache state ($c_1$) is a superset of $\tau$, we can reuse the summarization of the left subtree to generate the WCET of the right subtree. The WCET of the right subtree is generated from the stored representative path and the new incoming cache state ($c_1$). The WCET of the right subtree is 13. Note that the timing generated from the representative path is different from before. This is the result of generating dynamic timing in reuse.

Continuing the analysis, at the root, the path on the right is the longest path in the tree and the WCET of the program is 20, while the WCET generated before was 22. Note that again, the WCET of the program was generated without exploring the longest feasible path in the tree.

Reuse is the key to scalability, since with reuse our analysis avoids the full exploration of a symbolic execution tree. Reuse in the presence of the dynamic timing model is only possible when path and timing analysis are performed in one integrated phase. An “integrated path and timing analysis” would theoretically give us ideal precision for WCET analysis. The first attempt to perform integrated analysis based on symbolic execution was Lundqvist et al. [10] which did not scale to realistic programs. Our method, to the best of our knowledge, is the first scalable approach to perform integrated path and timing analysis.

Similar to [10] and [4], our work can be extended to consider pipeline too. With a pipeline, the problem of timing anomaly [11] arises. In the literature, there has been rather ad-hoc solutions to this problem. For example, [12] and [13] introduce bounds in order to produce a range that accommodates the complexity raised by having a pipeline. These general approaches are also applicable to us. Therefore, the pipeline issue is largely orthogonal to this paper.

In Section V, we demonstrate both the accuracy and scalability of our algorithm on realistic benchmarks. In addition to proving metrics, we will also explain our improvement in the context of different characteristics of the examples such as loop behavior and amount of infeasible paths.

II. General Framework

A. Symbolic Execution with Abstract Cache

We build this work on top of the symbolic execution framework as presented in [8]. For this paper, we need to go beyond program path analysis; thus we extend some fundamental concepts in [8] to include an abstract cache. The abstract cache is modeled following the standard semantics of abstract cache for most analysis, formally defined in [2].

We model a program by a transition system. A transition system $P$ is a tuple $\left(\mathcal{L}, \ell_0, \rightarrow\right)$ where $\mathcal{L}$ is the set of program points, $\ell_0 \in \mathcal{L}$ is the unique initial program point. Let $\rightarrow \subseteq \mathcal{L} \times \mathcal{L} \times \mathit{Ops}$, where $\mathit{Ops}$ is the set of operations, be the transition relation that relates a state to its (possible) successors by executing the operations. This transition relation models the operations that are executed when control flows from one program point to another.

We restrict all (basic) operations to be either assignments or assume operations. The set of all program variables is denoted by $\mathit{Vars}$. An assignment $x := e$ corresponds to assign the evaluation of the expression $e$ to the variable $x$. In the assume operator, $\mathit{assume}(\mathit{cond})$, if the conditional expression $\mathit{cond}$ evaluates to true, 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 $op \in \mathit{Ops}$.

A transition system naturally constitutes a directed graph, where each node represents a program point and edges are defined by the relation $\rightarrow$. This graph is similar to (but not exactly the same as) the control flow graph of a program. By representing a program using transition systems, the program can be executed symbolically in a simpler way.

**Definition 1** (Symbolic State). A symbolic state $s$ is a quadruple $\langle \ell, c, \sigma, \Pi \rangle$ where $\ell \in \mathcal{L}$ corresponds to the concrete current program point, $c$ is the abstract cache state, the symbolic store $\sigma$ is a function from program variables to terms over input symbolic variables, and the path condition $\Pi$ is a first-order logic formula over the symbolic inputs which accumulates constraints the inputs must satisfy in order to reach this state (following some program path).

Let $s_0 \equiv \langle \ell_0, c_0, \sigma_0, \Pi_0 \rangle$ 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 $\sigma$ 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 $[\mathit{cond}]_{\sigma}$ can be defined analogously. The set of first-order logic formulas and symbolic states are denoted by $\mathit{FO}$ and $\mathit{SymStates}$, respectively.

Our analysis is performed on LLVM IR, which is expressive enough for cache analysis and maintains the general CFG of the program required for path analysis. Given a program point $\ell$, an operation $op \in \mathit{Ops}$, and a symbolic store $\sigma$, the function $\mathit{acc}(\ell, op, \sigma)$ denotes the sequence of memory block accesses by executing $op$ at the symbolic state $s \equiv \langle \ell, c, \sigma, \cdot \rangle$. While the program point $\ell$ identifies the instruction cache access, the sequence of data accesses are obtained by combining both $op$ and $\sigma$ together. For simplicity of the presentation, we assume that all data accesses can be resolved precisely. In practice, it often is the case, due to the fact that we perform loop unrolling. In the implementation, when a data access cannot be resolved to a specific memory address, we follow the treatment as in [5] for loading memory.
ranges into the cache for persistent analysis\textsuperscript{3}, 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.

**Example 4.** In our symbolic execution framework, both add and load instructions are modeled as assignments. While the former involves no data access, the latter is used to load to the registers. If the respective memory block is not in the cache, then a cache miss happens and if the memory block has been loaded to the cache before the access to the memory block is a cache hit.

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

\[
s' \triangleq \begin{cases} (\ell', c', \sigma, \Pi \land \text{cond}) & \text{if } op \equiv \text{assume}(\text{cond}) \\ (\ell', c', \sigma[x \mapsto \cdot] \Pi) & \text{if } op \equiv x := e \end{cases}
\]

where \( \ell' \) is the new abstract cache derived from \( c \) and the sequence of accesses, \( acc(\ell, op, \sigma) \), which we call \( seq \) for short. \( \ell' \) is updated using the standard update function from the abstract cache semantics for must analysis from [2], where \( c' \equiv U(seq, c) \).

Our framework models the instruction and the data cache separately. So a cache state \( c \) contains two separate abstract caches \( \langle c_i, c_d \rangle \), where \( c_i \) is the abstract instruction cache and \( c_d \) is the abstract data cache. The update function from the abstract cache semantics for must analysis from [2] is used to capture the updates on both the data and the instruction cache. Although, the must analysis is originally only used to model the abstract instruction cache, since our framework performs loop unrolling, the must analysis is still enough expressive to model the abstract data cache.

Abusing notation, the execution step from \( s \) to \( s' \) is denote as \( s \xrightarrow{\ell} s' \). Given a symbolic state \( s \equiv (\ell, c, \sigma, \Pi) \) we also define \([s]\) : \( \text{SymStates} \to \text{FO} \) as the projection of the formula \( \bigwedge_{e \in V \sigma} v = \cdot\sigma \land \Pi \sigma \) onto the set of program variables \( V \sigma \). 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 tuple \( \langle \ell, c, \sigma, \Pi \rangle \) where \( c \) is the cache state reaching the symbolic state \( s \) and \([s]\) is the constraint component of the symbolic state \( s \), obtained by projecting the symbolic store onto the set of program variables. A path \( \pi \equiv s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_m \) is feasible if \( s_m \equiv (\ell, c_m, \sigma_m) \) 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 \( \ell \) to another program point, then \( \ell \) is called the ending point of the program. Under that circumstance, if \( s_m \) is feasible, then \( s_m \) is called terminal state.

The set of program variables \( V \sigma \), also include the timing variable \( t \). Our analysis computes a sound and accurate bound for \( t \) in the end, across all feasible paths of the program.

\textsuperscript{3}Huynh et al. in [6] have fixed a safety issue with the treatment of loading memory ranges into the cache from [5]. 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.

Note that \( t \) is always initialized to 0 and the only operations allowed upon it are concrete increments. Given a symbolic state \( s \equiv (\ell, c, [s]) \) and a transition \( tr : \ell \xrightarrow{op} \ell' \), the amount of increment at \( s \) by executing \( tr \) will be evaluated by the execution time of the LLVM instructions and the access time for \( seq \). The timing variable \( t \) is not used in any other way.

Recall that our transition system is a directed graph. We now introduce concepts which are required in our loop unrolling framework.

**Definition 3 (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 4 (Loop Head).** Given a directed graph \( G = (V, E) \) and a loop \( L = (V_L, E_L) \) of \( G \), we call \( E \in V_L \) a loop head of \( L \), also denoted by \( E(L) \), if no node in \( V_L \), other than \( E \) has a direct successor outside \( L \).

**Definition 5 (Ending Point of Loop Body).** Given a directed graph \( G = (V, E) \), a loop \( L = (V_L, E_L) \) of \( G \) and its loop head \( E \). We say that a node \( u \in V_L \) is an ending point of \( L \) body if there exists an edge \( (u, E) \in E_L \).

**B. Constructing Summarizations**

Given the definitions in Section II-A, a symbolic execution tree can be obtained in an obvious manner. However, it is not possible to naively construct the full symbolic execution tree for any non-trivial program. While symbolic execution allows us to exclude infeasible states from the timing calculation, thus achieve precision, the scalability remains as the key challenge.

The main contribution of this work is then an adaptation of a dynamic programming algorithm, which employs the concept of summarization and reuse. As briefly mentioned in Section I, for each finite subtree, already analyzed, the safety of reuse is checked by the core concepts of interpolation and dominance. If the state satisfies the reuse conditions, the interpolant and the dominating condition, the representative path will be maintained as the dominating one at the reuse point. We now elaborate on these technical concepts. The construction of correct summarizations requires the concept of [Craig] interpolant [14].

**Definition 7.** (Interpolant) Given two first-order logic formulas \( F \) and \( G \) such that \( F \models G \), then there exists an interpolant \( H \) denoted as \( \text{Int}(F, G) \), which is a first-order logic formula such that \( F \models H \) and \( H \models G \), and each variable of \( H \) is a variable of both \( F \) and \( G \).

The concept of interpolant enables us to approximate efficiently, at the root of a subtree, the weakest precondition in order to maintain the infeasibility of all the nodes inside. In the context of program verification, for each particular subtree, it helps capture succinctly the condition which ensures the safety of the subtree, since safety proof is achieved by showing that the error nodes are unreachable or infeasible.
The originating work applying interpolation for reuse is [7]. In the context of WCET analysis, the pioneer work applying interpolation for reuse are [8], [9]. In the context of program analysis, the problem is formulated so that the scalability can be achieved by reusing previously computed sub-problem. In other words, this is 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. Our framework adopts the efficient algorithm for computing the interpolant from [15].

Definition 8. [Representative Path]. The representative path \((\Psi)\) is the sequence of the program points demonstrating the longest path in an explored subtree. A representative path is of the form \([t_s, \Psi, s_t]\), where \(t_s\) is the execution time of the instructions in the representative path. \(\Psi\) is the memory access sequence in the representative path and \(s_t\) contains the set of the constraints along the representative path.

The memory access sequence \(\Psi\) is used to calculate the access time of the memory accesses in a representative path w.r.t. a cache state. As a result, the timing of a representative path is obtained dynamically and can alter based on the incoming cache state. Note that we do not need to store the complete memory access sequence of a representative path. Some of the memory accesses can be resolved to Always Hit/Always Miss. These accesses can be resolved regardless of the input cache state\(^4\) and their timing are added to the static time stored in \(t_s\). \(s_t\) is used to check that the satisfiability of a representative path w.r.t a state \(s \equiv (\ell, c, [s])\). If \([s_t \land s]\) is satisfiable, we say that the satisfiability of the representative path is satisfiable in the new context, otherwise it is unsatisfiable. Abusing notation, we denote the satisfiability test \([s_t \land s]\) in a short form as \([\Gamma]\).

Definition 9. [Dominating Condition]. The dominating condition (\(\delta\)) is a set of constraints reasoning over the existence and non-existence of memory blocks in a cache state, which while satisfied, guarantees that its respective representative path dominates all other paths inside a subtree.

The conditions in the dominating condition are either of the form of existence constraints (\(m_i \in \text{cache}\)), which assure the presence of a memory block \(m_i\) in a cache state, or non-existence constraints (\(m_i \notin \text{cache}\)) which do the opposite. When existence constraints are checked against a cache state, if satisfied, the access for \(m_i\) is for sure a cache hit and a cache miss otherwise. The domination of a representative path over the other paths in a subtree, is guaranteed by adding non-existence constraints over the memory blocks accessed only in the representative path and adding existence constraints over the memory blocks accessed only in the other paths.

The representative path and the dominating condition are generated recursively by the help of combine-path and merge-path steps, such that the representative path and dominating condition at a node is generated based on the representative path and dominating condition of its children nodes without affecting the soundness of the analysis. More details on generating the representative path and the dominating condition are presented in the next Section II-D.

Definition 10. [Summarization of a Subtree]. Given two program points \(\ell_1\) and \(\ell_2\) such that \(\ell_2\) post-dominates \(\ell_1\) in the transition system and assume we analyze all the paths from entry point \(\ell_1\) to exit point \(\ell_2\) w.r.t. an incoming symbolic state \(s\). The summarization of this subtree is defined as the tuple \([\ell_1, \ell_2, \Gamma, \Delta, \Psi, \delta]\), where \(\Gamma\) is the representative path and \(\delta\) is the dominating condition. \(\Delta\) is an abstract transformer capturing the abstract relation between variables and the cache state at \(\ell_1\) and \(\ell_2\). Finally, \(\Psi\) is an interpolant.

By definition, the abstract transformer \(\Delta\) in a summarization of a subtree from \(\ell_1\) to \(\ell_2\) will be the abstraction of all feasible paths (w.r.t. the incoming symbolic state \(s\)) from \(\ell_1\) to \(\ell_2\).

The abstract transformer consists of program abstract transformer \((\Delta_p)\) and cache summary \((\Delta_c)\). The program abstract transformer, adopted from [8], captures the input-output relation between the program variables using the polyhedral domain. Note here that, in general, the program abstract transformer is not a functional relation. On the other hand, the cache summary is generated by merging and combining the memory access sequences such that the generated cache summary would summarize the effect of the memory access sequences. The cache summary is needed when a summarization is reused. For example, if a summarization between the program points \(\ell_1\) and \(\ell_2\) is used in \(\ell_1\), by applying the cache summary to \(c_1\) (the cache state at \(\ell_1\)), we can generate \(c_2\) (the cache state at \(\ell_2\)). A combine-summary and merge-summary step is defined to generate the cache summary in a recursive manner, such that the cache summary of a node is generated based on the cache summary of its children nodes without affecting the soundness of the analysis. More details on the how to generate and apply cache summaries is presented in Section II-C.

Summarization of a subtree is important for summarizing loop iterations. However, we need to define the summarization of a program point too.

Definition 11. [Summarization of a Program Point]. A summarization of a program point \(\ell\) is the summarization of all paths from \(\ell\) to \(\ell'\) (w.r.t. the same context), where \(\ell'\) is the nearest program point that post-dominates \(\ell\) s.t. \(\ell'\) is of the same nesting level as \(\ell\) and either is (1) an ending point of the program, or (2) an ending point of some loop body.

As \(\ell'\) can always be deduced from \(\ell\), in the summarization of program \(\ell\), we usually omit the component about \(\ell'\).

Moving back to the symbolic execution framework, consider a new node \(j\) with the symbolic state \(s_j \equiv (\ell, c_j, [s_j])\) and an analyzed node \(i\) with the stored summarization \([\ell, \Gamma, \Delta, \Psi, \delta]\), associating to the same program point \(\ell\). Node \(j\) will not be further expanded while the following conditions are satisfied:

1. Its constraint component \([s_j]\) is less general than the previously stored interpolant \(\Psi\) i.e. \([s_j] \nsubseteq [\Psi]\).
2. The representative path is still feasible in the incoming program context i.e. \([\Gamma]\) s.t. SAT.
3. The dominating condition is still satisfied w.r.t. to the incoming cache context. This is denoted by check(\(\delta, c_j\)).

The timing for node \(j\) will be generated from the representative path at node \((\Gamma)\) and the cache state at node \((c_j)\).

C. Generating Cache Summaries

The cache summary captures the relation between the abstract cache components. The cache summary is needed when a summarization is reused. The cache summary is the

\(^4\)Similar to the idea of the Always Hit/Always Miss/Persistent categories in the AI framework presented in [2]
The cache summary is generated to summarize a set-associative abstract cache for must analysis with LRU replacement policy. However, the concept of cache summary can be accordingly updated based on the other cache policies too. We consider an abstract set-associative cache $c$ as a set of $N$ cache sets, where $N = C/(BS \times A)$, where is the cache capacity, $BS$ is block size and $A$ is cache associativity. $BS$ is block size and $A$ is cache associativity. We denote a cache-set with $cs$ where $c \in [cs_1, ..., cs_N]$. 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.

Similarly, the cache summary ($\Delta_s$) consists of $N$ set summaries for each respective cache-set. We denote a set summary with $\Delta_s$ where $\Delta_s = [\Delta_s_1, ..., \Delta_s_N]$. Each set summary is of the form $[n, M]$, where $n$ is a static number which denotes the number of the cache lines that the summary will load memory block to. $n$ is always less than or equal to $A$. $M$ is a set of $[m, i]$ tuples which indicate that the memory block $m$ should be loaded to the cache with relative age $i$. When a cache summary is applied to an abstract cache state, for each set the respective set summary is applied to the respective set.

**Example 5.** For example, consider the sequence of memory blocks $m_1, m_2, m_3, m_2$. The cache summary for this sequence of memory blocks would be $\langle (\langle m_2, 0 \rangle, \langle m_2, 1 \rangle, \langle m_1, 2 \rangle) \rangle$. Now, consider a fully associative cache of size 4 which initially contains $m_1$. If we apply the cache summary to this cache state, the generated cache state would be as follow which is similar to the cache state if we had loaded the memory blocks one by one.

<table>
<thead>
<tr>
<th>$\emptyset$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_2$</td>
<td>$m_3$</td>
<td>$m_1$</td>
<td>$m_2$</td>
<td></td>
</tr>
</tbody>
</table>

The size of the cache summary is of the order of $O(C)$, where $C$ is the cache capacity, since in the process of generating the cache summary only the memory blocks which their relative age is less than or equal to the cache associativity are stored. The rest of the memory blocks would be naturally pushed out of the cache. As a result, while our analysis remains sound, the size of the cache summaries will remain constant wrt. to the cache capacity.

A combine and merge step are defined to generate the cache summary in a recursive manner, such that the cache summary of a node is generated based on the cache summary of its child nodes without affecting the soundness of the analysis. The combine-set and merge-set functions in Algorithm 1 the combine and merge steps for two cache-sets. The combine and merge steps for two cache summaries can be obviously implemented in a similar manner.

The combine-set function, presents the combine step to combine vertically two set summaries. This function preserves all the $[m, i]$ tuples from $M_1$ and for each $[m', i']$ tuple in $M_2$, if $m'$ is not in any tuple in $M_1$, adds the tuple $[m', i' + n_1]$ to the combined set summary. The relative age is increased by $n_1$ since the set summaries are combined vertically. The rest of the $[m', i']$ tuples in $M_2$ are not added, since $m'$ is already in a tuple in $M_1$. The merge-set function, presents the merge step to merge horizontally two set summaries. It preserves the memory blocks common on both the cache summaries with the maximum relative age.

**Algorithm 1 Combining and Merging Two Set Summaries**

**function** combine-set($\Delta_{s1}, \Delta_{s2}$)

1. Let $\Delta_{s1}$ be $(n_1, M_1) \land \Delta_{s2}$ be $(n_2, M_2)$
2. **foreach** $[m, i] \in M_2$ do
   1. if $[m, i] \notin \text{List1}$ and $n_1 + i \leq \text{Associateivity}$ then
   2. $M_1 := M_1 + (m, n_1 + i)$
3. **endfor**
4. return $(n_1 + n_2, M_1)$
**end function**

**function** merge-set($\Delta_{s1}, \Delta_{s2}$)

1. Let $\Delta_{s1}$ be $(n_1, M_1)$ \land $\Delta_{s2}$ be $(n_2, M_2)$
2. $M := \emptyset$
3. **foreach** $[m, i] \in M_1 \land [m, j] \in M_2$ do
   1. $M := M + [m, \max(i, j)]$
4. **endfor**
5. return $(\max(n_1, n_2), M)$
**end function**

**Algorithm 2 merge-path Function**

**function** merge-path($\Gamma_1, \Gamma_2, \delta_1, \delta_2$)

1. Let $\Gamma_1$ be $[t_1, \Upsilon_1, \sigma_1] \land \Gamma_2$ be $[t_2, \Upsilon_2, \sigma_2]$
2. $\delta := \delta_1 \land \delta_2$
3. if (max-time($\Gamma_1, \delta) \leq \text{max-time($\Gamma_2, \delta)$)$$)
   1. $\text{swap}(\Gamma_1, \Gamma_2)$, $\text{swap}(\delta_1, \delta_2)$
4. if (min-time($\Gamma_1, \delta) \geq \text{max-time($\Gamma_2, \delta)$)$$)
   1. $\{\text{Dominance Reached}\}$
5. return $\{\Gamma_1, \delta\}$

**for** each $m_i \in \Upsilon_1 \land \text{con-straint(}$

1. $\delta := \delta \land \{m_i \notin \text{cache}\}$
6. $t_i := t_i + \text{max-s} \and \Gamma_1 = \Gamma_1 - \{m_i\}$
7. if (min-time($\Gamma_1) \geq \text{max-time($\Gamma_2)$}$$)
   1. $\text{return} \{\Gamma_1, \delta\}$
8. **endfor**

**for** each $m_i \in \Upsilon_2 \land \text{con-straint(}$

1. if (min-time($\Gamma_1) \geq \text{max-time($\Gamma_2)$}$$)
   1. $\text{return} \{\Gamma_1, \delta\}$
2. $\text{Dominance Reached}$
3. **endfor**

The representative path and the dominating condition are generated with respect to a set-associative abstract cache for must analysis with the LRU replacement policy. However, it can be accordingly extended to other cache policies.

**D. Generating Representative Paths and Dominating Conditions**

The representative path and the dominating condition are generated with respect to a set-associative abstract cache for must analysis with the LRU replacement policy. However, it can be accordingly extended to other cache policies.
The merge-path function in Algorithm 2, presents the step to merge horizontally two representative paths and their respective cache dominating conditions. In the first step, the dominating conditions are merged (line 9) and based on the maximum possible timings of each of the representative paths the longer representative path is chosen (line 10 to 11). The merged dominating condition should contain the reason for the dominance of both representative paths (in their respective sub-trees) and the reason that the chosen representative path dominates the other one. As a result, both paths are compared and in case the first path’s minimum timing (based on the unresolved memory accesses in $\Gamma_1$ and the conditions in $\delta$) is more than the second path’s maximum timing, the first path dominates the second path and it is returned as the representative path (lines 12 and 13).

If dominance is not reached, for each of the memory blocks in $\Upsilon_1$, where there is no condition added to $\delta$, a condition of the form $m_i \not\in \text{cache}$ is added to $\delta$, which forces the access to $m_i$ in $\Gamma_1$ to be a cache miss and as a result, the timing for $\Gamma_1$ to increase. The condition is added to $\delta$ and $\Gamma_1$ is updated, with the hope that it would be enough to infer the dominance of the chosen representative path (line 14 to 18). If no, the process continues and if there is no more memory block left in $\Upsilon_1$, it will start adding constraints over the memory blocks in $\Upsilon_2$ and the dominance test is performed again (lines 19 to 22). Due to the static number of the memory accesses this process always terminate.

Algorithm 3 combine-path function

```plaintext
function combine-path(t_ins, seq, $\Gamma$, $\delta$)
    Let $\Gamma$ be \{t_s, $\Upsilon$, $s_1$\}
    \[(23)\] $t_s = t_s + t_{\text{ins}}$
    \[(24)\] foreach $m_i \in \text{seq}$ do
        \[(25)\] if $m_i \not\in \Upsilon$ and relative-age($m_i$, $\Upsilon$) < Associativity then
            \[(26)\] $\Upsilon = \Upsilon - m_i$, $t_s = t_s + \text{hit}$
        \[(27)\] add-to-begginning($m_i$, $\Upsilon$)
    \end{for}
    \[(28)\] else
    \[(29)\] foreach $m_i \in \text{seq}$ do
        \[(30)\] if $\{m_i \in \text{cache}\} \in \delta$ then
            $\delta = \delta - \{m_i \in \text{cache}\}$
        \end{for}
    \end{for}
    \[(32)\] return $\{\Gamma, \delta\}$
\end{function}
```

The combine-path function in Algorithm 3, presents the step added to combine vertically two paths. The execution time of the instructions ($t_{\text{ins}}$) is added to the static time ($t_s$) at line 23. Moreover, for each of the accesses in the respective transition (seq), if the memory block is in $\Upsilon$ and its relative-age is less than the associativity, the memory block is removed from $\Upsilon$. $t_s$ is increased by hit latency and the memory block is added back to the beginning of the list. This is to replicate the order of loading memory blocks in a list, but in a reverse order (line 26 and 27). Otherwise, the memory block is added to $\Upsilon$ and its access time is calculated at the time of reuse and based on the incoming cache context (line 28). Note that the list of the unresolved memory accesses for the combined representative path will remain an ordered list. In the next step, the combined dominating condition is updated. The update is quite simple, for each memory block $m_i$ in seq, if there is a condition of the form $m_i \in \text{cache}$, the condition is removed, since due to the presence of $m_i$ in seq, this condition will always be true and it is removed from $\delta$.

III. A Example

Consider the control flow graph (CFG) of a program fragment in Figure 3(a). Each node, shown as a rectangle, abstracts a basic block. Also shown is the program point ((1), (2), \cdots, (10)) at the block entry, the static timing (timing of the corresponding instructions while assuming that all memory accesses are hits), and the memory access sequence. For brevity, we might use interchangeably the identifying program point when refer to a basic block. We shall consider accesses to memory blocks $m_1$ and $m_2$ only. Two outgoing edges signify the 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. We also assume a cache miss penalty of 9 cycles.

Next, in Figure 3(b), we depict a symbolic execution tree of the program. The nodes, shown as circles, represent the program points, with superscripts to distinguish the multiple occurrences. Each path denotes a symbolic execution of the program. Each node is associated with a symbolic state $s \equiv (l, c, [s])$, which preserves the context reaching the node. While the context is not explicitly shown in the Figure (since the basic blocks are shown only abstractly), we shall make use of some obvious properties of the context of some nodes. Asssume that none of the basic blocks modifies the variable $x$. Then, for example, the context of the leftmost node, labeled (8)$^{(1)}$ contains both the formula $x = 0$ and $x > 1$ (amongst possibly many other formulas arising from nodes (1), (2), (4)$^{(1)}$, (5)$^{(1)}$, (7)$^{(1)}$, and therefore is equivalent to false. In other words, the leftmost path in Figure 3(b) is in fact an infeasible path. Note that we have not (fully) drawn the subtree below node (4)$^{(2)}$ in Figure 3(b).

The core technical step in this paper is reuse. To exemplify this, consider the possible reuse of the subtree (7)$^{(1)}$ on (7)$^{(2)}$. That is, even though we have depicted the subtree (7)$^{(2)}$ in full, could we in fact have simply replaced this depiction and instead just indicated that (7)$^{(2)}$ is a subtree whose WCET is the same as that of (7)$^{(1)}$? The WCET of (7)$^{(1)}$ (which only contains one path) is $20 \leq 5 + 10 + 0 + 5$, since the access $m_2$ is a hit. The summation generated from the analysis of node (7)$^{(1)}$ is \{(7), $\Gamma$, $\Delta$, $\Psi$, $\delta\}$, where the stored interpolant.
is $\Psi : x = 0$, the stored dominating condition is an empty set containing no constraints, since the subtree beneath ($\gamma$) contains only one feasible path ($\delta : [\Gamma]$. Moreover, the stored representative path is $\Gamma : [20, \{m_2\}, \{x \leq 1\}]$, where 20 is the timing of the corresponding instructions, $\{m_2\}$ is the sequence of the memory accesses in the representative path and $\{x \leq 1\}$ is the set of the constraints along the representative path. For brevity, we do not go through the details regarding the stored abstract transformer $\Delta$ in this example.

The summarization generated from the analysis of node ($\gamma$) clearly is not usable in ($\gamma$) (which has two paths). The important point here is the reason: it is that some infeasible path in ($\gamma$) is in fact feasible in ($\gamma$). We defined this first check for reusability in the form of the interpolant $\Psi$ associated to the subtree. As a result, node ($\gamma$) is expanded and the subtree beneath ($\gamma$) is analyzed.

Continuing the analysis, consider the pair of nodes ($\delta$) and ($\delta$). We will show how reuse can in fact take place here, and why so. Take note, without proof, that the “longest” path in the subtree rooted at ($\delta$) is (the long path, in blue color): ($\delta$), ($\delta$), ($\delta$), ($\delta$), ($\delta$), ($\delta$), ($\delta$). The analysis of ($\delta$) therefore is given by this path, and its WCET is $54 = 10 + 15 + 9 + 5 + 10 + 0 + 5$, since the first access to $m_2$ is a miss while the second one is a hit. The summarization generated from the analysis of node ($\delta$) is $\{\{4, \Gamma, \Delta, \Psi, \delta\}, \{x \leq 0 \land x \leq 1\}$, where the interpolant (which is a trivial one, in this case) is “$\Psi$: true”. The representative path is $\Gamma \equiv [45, [m_2, m_2], \{x = 0 \land x \leq 1\}]$, where 45 is the timing of the corresponding instructions, $\{m_2, m_2\}$ is the sequence of the memory accesses in the representative path and $\{x = 0 \land x \leq 1\}$ is the set of the constraints along the representative path. Since the second access to $m_2$ always will be a hit, the representative path is stored in the form of $\Gamma \equiv [45 + 0, [m_2], \{x = 0 \land x \leq 1\}]$.

Let us focus on the other two feasible paths emerging from ($\delta$). The timing of the corresponding instructions and the sequence of the memory accesses for the first path ($\delta$), ($\delta$), ($\delta$), ($\delta$), ($\delta$), ($\delta$), ($\delta$) is $44 + 0, [m_1, m_2]$ (since the second access to $m_1$ is an always hit). The instructions timing and the sequence of the memory accesses for the second path is $39, [m_1, m_2]$. Comparing the representative path with these two paths, the representative path will dominate the other two paths while the access to $m_1$ is a hit. As a result, the stored dominating condition at node ($\delta$) is “$\delta : [m_1 \in \text{cache}”$.

Now we can exemplify this reuse on the subtree ($\delta$). We first check if the context of ($\delta$) satisfies the interpolant computed after finishing ($\delta$). In this case, the interpolant is true, thus the check trivially holds. We then check that in the context of ($\delta$), indeed $m_1$ is in the cache. This condition also holds and so we can reuse the representative path of ($\delta$), yielding the same timing of 54. We remark here that in this particular example, the timing of the path from ($\delta$) to ($\delta$) ($\delta$) is the same, since $m_1$ is present in the cache in both cases. However, in practice, when the memory accesses are more diverged, we often observe the reuse of different concrete timing, though for the same representative path.

Finally, we easily arrive at the WCET of the entire tree, thus the entire example program, to be 88 cycles ($10 + 15 + 9 + 54$, since the access to $m_1$ at ($\delta$) is a 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: ($\delta$), ($\delta$), ($\delta$). Importantly, it discovers that at ($\delta$), $m_1$ must be in the cache. Thus the access to $m_1$ at ($\delta$) is hit. However, at ($\delta$), AI has to conservatively declare that neither $m_1$ nor $m_2$ is in the cache. Consequently, the final worst case timings for the basic blocks that have some memory accesses are: ($\delta$), ($\delta$), ($\delta$), ($\delta$), ($\delta$), ($\delta$), ($\delta$), ($\delta$).

If we aggregate using a path-insensitive high-level analysis, the WCET estimate is $121 = 10 + \max(19, 24) + 10 + \max(24, 9) + 5 + \max(33, 19) + 5$. Aggregating using a fully path-sensitive high-level analysis will give the WCET of 97 $= 10 + 24 + 10 + 24 + 5 + 19 + 5$. In both cases, the resulting estimates are not as precise as our estimate.

We then now discuss briefly about [4], namely AI+SAT. Whether [4] can achieve the precision we achieve highly depend on its merging strategies. In theory, [4] can go to the extreme and choose not to merge at program point ($\delta$), thus it can estimate precisely the timing of basic block ($\delta$). This means that [4] would need to consider all the 8 paths, constituting the full symbolic execution tree. In general the tree is exponential in size. While our algorithm is equipped with the concept of reuse to mitigate such explosion, [4] has none. As such the approach needs to merge frequently, resulting in impractic precision, as we will demonstrate experimentally in Section V.

## IV. The Integrated Symbolic Execution Algorithm

In this section, our algorithm shown in Algorithms 4 and 5 is presented. Our key function, Analyze, takes as inputs a symbolic state $s \equiv (\ell, c, [\delta])$, and the transition system of the input program $P$. It then performs the analysis and returns the summarization for the program point $\ell$ as in Def. 11. The worst case execution bound is then achieved from the representative path of the summarization of the root of the symbolic execution tree and the initial cache state $c$.

### Base Cases: Analyze handles 4 base cases. First, when the symbolic state $s$ is infeasible (line 33), no execution needs to be considered. Note that here path-sensitivity plays a role since only provably executable paths will be considered. As a result, the returned representative path is $[-\infty, \text{-}, \text{-}, \text{false}]$. Second, $s$ is a terminal state (line 34). Here $\text{Id}$ refers to the identity function, which keep the values of variables unchanged. The ending point of a loop is treated similarly in the third base case (line 36). The last base case, lines 38-39, is the case that a summarization can be reused. We have demonstrated this step, in the end of Section II-B.

### Expanding to the next programming point: Line 50 depicts the case when transitions can be taken from current program point $\ell$, and $\ell$ is not a loop starting point. Here we call TransStep to move recursively to next program points. TransStep implements the traversal of transition steps emanating from $\ell$, denoted by outgoing($\ell$, $P$), by calling Summarize recursively and then compunds the returned summarizations into a summarization of $\ell$. For each $t$ in TransSet, TransStep extends the current state with the transition. The resulting child state is then given as an argument in a recursive call to Analyze (line 63). From each summarization of a child returned by the call to Analyze, the algorithm computes a summarization, contributed by that particular child to the parent as in lines 64-66. Finally, all of these summarizations will be compounded using the JoinHorizontal function (line
Algorithm 4 Integrated WCET Analysis Algorithm

function Analyze(s, P)
    Let s be \langle \ell, c, [s]\rangle
    (33) if ([s] \equiv false) return \langle \ell, [-\infty, [], false], false, false, [], []\rangle
    (34) if (outgoing(\ell, P) = \emptyset) return \langle \ell, [0, [\Gamma]], true, [\Gamma]\rangle
    (35) if (loop_head(\ell, P)) return \langle \ell, [0, [\Gamma]], true, [\Gamma]\rangle
    (36) if (loop_end(\ell, P)) return \langle \ell, [0, [\Gamma]], true, [\Gamma]\rangle
    (37) S := \langle \ell, [\Gamma, \Delta, \Psi, \delta] \equiv \text{memoed}(s)\rangle
    (39) if ([s] \equiv \Psi \land [\Gamma] \not\equiv false \land \check{\delta}(c)) return S\rangle
    (40) if (loop-head(\ell, P))
        S1 := \langle [..., \Gamma1, \Delta1, ...] \rangle := TransStep(s, P, entry(\ell, P))
    (41) if ([\Gamma1] \equiv false)
        \text{S} := \text{JoinHorizontal}(S1, \text{TransStep}(s, P, exit(\ell, P)))
    (43) else
        S := \text{transStep}(s, P, outgoing(\ell, P))
    (44) \text{return } S\rangle

end function

Handling Loops: Lines 41-49 handle the case when the current program point \ell is a loop head. Let entry(\ell, P) denote the set of transitions going into the body of the loop, and exit(\ell, 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 41). When the returned representative path is false, it means that we cannot go into the loop body anymore, so we proceed to the exit branches. The returned summarization is compound (using JoinHorizontal) with the summarization of the previous unrolling attempt (line 43). Otherwise, if some feasible paths are found by going into the loop body, we use the returned abstract transformer to produce a new continuation context, (line 44), so that we can continue the analysis with the next iteration (line 45). The returned information is then compound (lines 46 - 48) with the first unrolling attempt. Our algorithm can be reduced to linear complexity because these compound summarizations of the inner loop(s) can be reused in later iterations of the outer loop. Finally, we finish this section by presenting a proof of concept for the framework presented in this paper.

Theorem 1 (Sound WCET Estimation). The generated time by the summarization at the root of the symbolic tree is a sound over-approximation of the WCET of the respected program.

Proof: The framework presented in this paper performs a depth-first traversal of the symbolic tree. In all the steps in the depth-first traversal except reuse the only performed action is widening the context, thus, the only effect of these steps is over-approximating the WCET. These steps do not affect the soundness of the analysis. In case we can show that the analysis remains sound with the reuse step, we can conclude that the generated time by the summarization at the root of the symbolic tree is an over-approximation on the worst-case execution time of the longest path in the symbolic execution tree. As a result, the generated time will be a sound over-approximation of the WCET of the analyzed program.

In the reuse step, the summarization of a node i which is denoted by \langle \ell, \Gamma, \Delta, \Psi, \delta\rangle is reused to generate the WCET at a new node j, such that i and j associate to the same program point \ell. The reuse step indicates that the representative path \Gamma will remain the longest path w.r.t. the incoming state (\ell, c, \sigma, \Pi), if first the incoming context is less general than the stored interpolant (\Psi \equiv \delta). Secondly, the representative path is still feasible in the incoming context [\Gamma] \equiv \Delta and finally the dominating condition is still satisfied in the incoming cache context check(\delta, c) \equiv \Delta.

Assume that the reuse is not safe. As a result there should be a feasible path in the subtree beneath \ell, which its execution time is more than the execution time of the representative path \Gamma w.r.t. to the context reaching node j. Let us call this path \Gamma'. This can be true in two cases:

1) \Gamma' is infeasible in the subtree beneath i.
2) The execution time of \Gamma' was less than the execution time of the representative path \Gamma w.r.t. to the context reaching node i.

Otherwise, \Gamma' was chosen as the representative path in the final algorithm.
summarization of node \( \Gamma' \).

Let us examine the first case. If \( \Gamma' \) was infeasible in the subtree beneath \( i \) and feasible in the subtree beneath \( j \), the interpolant test would fail and the summarization was not reused in the first place. So, \( \Gamma' \) can not be infeasible in the subtree beneath \( i \).

Moving to the second case, where the execution time of \( \Gamma' \) was less than the execution time of the representative path \( \Gamma \) w.r.t. to the context reaching node \( i \). In this case the dominating condition test would fail. Since, dominating condition guarantees that the representative path \( \Gamma \) remains the longest path w.r.t. the new cache state (the cache state reaching node \( j \)). As a result, this is a contradiction and there can not exist a feasible path \( \Gamma' \) in the subtree beneath \( j \), where its execution time is more than the execution time of the representative path \( \Gamma \) w.r.t. to the context reaching node \( j \).

As a result, the representative path remains the longest path at the reuse point and the WCET of a subtree can be generated from the representative path and the incoming cache state with out any effect on the soundness of the analysis.

By that, we prove that the reuse step maintains the soundness of our analysis and the the generated time by the summarization at the root of the symbolic tree is an over-estimation on the worst-case execution time of the longest path in the symbolic execution tree and a sound over-approximation of the WCET of the analyzed program.


table

V. EXPERIMENTAL EVALUATION

We used an Intel Core i5 @ 3.2Ghz processor having 4Gb RAM for our experiments and built our system upon CLP(R) [17] 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, maintains the general CFG of the program. The LLVM instructions are simulated for a RISC architecture. We use Clang 3.2 [18] to generate the IR. The data and instruction cache settings in our experiments were adopted from [19] for ARM9 target processor. The cache configuration used in our experiments was a write-through, with no-write-allocate, 4-way set associative L1 cache (4KB size for instruction cache and 4KB size for data cache) with LRU replacement policy. The cache miss latency was 10 cycles and cache hit latency was set to 0 cycles.

Table I presents the results of our experiments on three WCET analysis algorithms:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unroll_d</strong>:</td>
<td>implements the algorithm presented in this paper, performing WCET analysis in one integrated phase.</td>
</tr>
<tr>
<td><strong>AI+SAT⊕ILP</strong>:</td>
<td>implements the algorithm in [4], i.e., a state-of-the-art method in micro-architectural modeling combined with ILP formulation for WCET aggregation. In order for ILP to work, we provide “standard” loop bounds for all the benchmarks.</td>
</tr>
<tr>
<td><strong>AI+SAT⊕Unroll_s</strong>:</td>
<td>implements a hypothetical algorithm. This WCET analysis was constructed to benefit from the state-of-the-art micro-architectural modeling in [4] as well as the state-of-the-art path analysis in [8]. AI+SAT⊕Unroll_s generates static timing for each basic block before aggregating via a path analysis phase. So the purpose of comparing to this algorithm is to show that our framework, Unroll_d, surpasses a direct combination of two superior existing methods.</td>
</tr>
</tbody>
</table>

For comparison purpose, we calculate the WCET improvement as \( \frac{B-U}{U} \times 100\% \), where \( U \) is the WCET obtained using our analysis algorithm, and \( B \) is the WCET obtained using the baseline approach. The last two columns in Table I present the improvement of Unroll_d over the other two analyses.

The three groups of the benchmarks used in our experiments are presented in Table I. Each group is separated by a double line:

**Benchmarks with lots of Infeasible Paths:** This group contains statemate and nsichnu from Mälardalen benchmarks [20] and teas, a real life implementation of a safety critical embedded system. teas 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, nsichnu and statemate are programs which contain loops of big-sized bodies with many infeasible paths. These benchmarks are often used to evaluate the scalability of path-analysis algorithms.

**Benchmarks with Simple Loops:** This group contains a set of academic programs from [20]. Though, the loops in these benchmark programs are considered to be simple, they contain memory accesses which might be resolved to a range of memory addresses, leading to the imprecision in the low-level analysis approaches that employ a fixed point computation.

**A. Discussion on Precision**

The generated WCET by Unroll_d for the first group of benchmarks, compared to AI+SAT⊕ILP, in average is improved by 34%, which increases up to 43% for nsichnu. Additionally, the WCET obtained by Unroll_d compared to AI+SAT⊕Unroll_s in average is improved by 17%. Comparing the generated WCET for nsichnu and statemate, it can be seen that part of the improvement comes from detection of infeasible paths (i.e., improvement of AI+SAT⊕Unroll_s over AI+SAT⊕ILP). However, by employing dynamic timing the improvement is increased in Unroll_d.

For a loop-free program like teas, we demonstrate that by using interpolation, we do not necessitate full enumeration of paths. The improvement in the generated WCET for teas is purely from infeasible path detection, but now, importantly, performed in one integrated phase. We highlight that, although the infeasible paths detected by Unroll_d and AI+SAT⊕Unroll_s are the same, there still is an average of 17% improvement in the WCET generated by Unroll_d compared to AI+SAT⊕Unroll_s. This is because infeasible path detection in Unroll_d is reflected in the micro-architectural modeling, by avoiding cache state merging at the join points in the CFG.

For the standard benchmarks in the second group, the WCET obtained by Unroll_d compared to AI+SAT⊕ILP is improved by 48% on average, and peaks at 94%. Moreover, the WCET obtained by Unroll_d compared to AI+SAT⊕Unroll_s is improved by 9% on average, and reaches the maximum of 14%. compress and nedes exhibit the most significant improvement compared to AI+SAT⊕ILP. Our investigation reveals that some infeasible paths in these two benchmarks contain loops, which due to infeasibility do not contribute to the WCET of these benchmarks. By
detecting these infeasible paths, the generated WCET improves dramatically. On the other hand, comparing the results of Unroll_d and AI+SAT⊕Unroll_s, the generated WCET for ndes and fly-by-wire is improved more compared to the other benchmarks. Further investigation reveals that these two benchmarks have memory accesses which are resolved to address ranges in the low-level analysis for AI+SAT⊕Unroll_s. Such scenarios significantly affect the precision of AI+SAT, which ultimately is still a fixed-point algorithm.

Furthermore, the effect of such memory accesses can be seen in the results of the analyses of the benchmarks in the third group. The estimated WCET by Unroll_d compared to the other two analyses is improved 18% on average, and peaks at 28%. These benchmarks do not contain many infeasible paths nor complicated loops and that is the reason why AI+SAT⊕Unroll_s is not able to improve the precision for these benchmarks. However, these benchmarks contain memory accesses which are resolved to address ranges in the fixed point computation. In contrast, Unroll_d maintains its precision. This is because Unroll_d can precisely resolve the memory accesses by performing loop unrolling with dynamic timing instead of employing a fixed point calculation to handle loops.

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

B. Discussion on Scalability

The two largest benchmarks in our experiments, 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⊕Unroll_s is 31% and 40%, respectively, while the precision is improved by 27%.

In the next section, we will elaborate on the point that similar to the symbolic simulation algorithm in [8], our integrated analysis remains super-linear with regard to both time and space complexity.

C. Analysis of Benchmarks with Complicated Loops

We have performed a set of experiments on benchmarks containing nested or complicated loops from [20]. These benchmarks are analysed not only to show the ability of our analyser to generate tight WCET for the benchmarks, but also to maintain the super-linearity of the symbolic execution WCET analyser in the integrated WCET analysis.

Table II, presents the results of the comparison of our symbolic execution analysis, Unroll_d, with AI+SAT⊕ILP and AI+SAT⊕Unroll_s for these benchmarks. In this table, the WCET estimated by Unroll_d compared to AI+SAT⊕ILP in average is improved by 50% and peaks at 97% for expint and insertsort. From the benchmarks in this group, bubblesort, expint, fft1, fir, insertsort, janne_complex and ud contain complicated loops which the number of the iterations of the loops is related to the context reaching the loop. In order to have a fair comparison between the three analyses, we provided the ILP formulation with the tightest possible loop bounds. Our aim was to eliminate the effect of the loop bounds for complicated loops on the estimated WCET. We like to elaborate on the results of the analyses for fir and bubblesort. These two benchmarks, while containing complicated loops, possess the least improvement. It is noteworthy, that the generated LLVM IR for insertsort, showing the best improvement in the generated WCET, is slightly different from the CFG of the C code. This has resulted in a much more costly path with less flow to receive maximum flow in the ILP formulation. The AI+SAT⊕Unroll_s analysis is able to tackle this issue in the high level analysis, but the issue still remains in the low-level analysis due to the fixed point computation. However, Unroll_d is able to generate a more precise WCET estimation.

The estimated WCET in all of these benchmarks show a significant improvement. Part of the improvement comes from the ability to find the exact number of loop iterations for complicated loops. This improvement is shared between our integrated analysis, Unroll_d, and AI+SAT⊕Unroll_s. However, comparing Unroll_d and AI+SAT⊕Unroll_s, the estimated WCET in average is improved by 12% and peaks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>C</th>
<th>LLVM</th>
<th>Time(s)</th>
<th>WCET</th>
<th>Time(s)</th>
<th>WCET</th>
<th>Time(s)</th>
<th>WCET</th>
<th>vs AI+SAT⊕ILP</th>
<th>vs AI+SAT⊕Unroll_s</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>400</td>
<td>736</td>
<td>0.84</td>
<td>1427</td>
<td>9.07</td>
<td>1212</td>
<td>21.36</td>
<td>1112</td>
<td>8.25</td>
<td>22.07%</td>
<td></td>
</tr>
<tr>
<td>nsichneu</td>
<td>4253</td>
<td>12879</td>
<td>161.58</td>
<td>85845</td>
<td>504.88</td>
<td>66808</td>
<td>709.03</td>
<td>48388</td>
<td>27.57</td>
<td>43.63%</td>
<td></td>
</tr>
<tr>
<td>statemate</td>
<td>1276</td>
<td>3345</td>
<td>13.89</td>
<td>12382</td>
<td>248.41</td>
<td>9101</td>
<td>358.94</td>
<td>7644</td>
<td>16.01</td>
<td>38.27%</td>
<td></td>
</tr>
<tr>
<td>ndes</td>
<td>231</td>
<td>1755</td>
<td>1.145</td>
<td>304369</td>
<td>37.95</td>
<td>174266</td>
<td>38.92</td>
<td>148368</td>
<td>14.86</td>
<td>51.25%</td>
<td></td>
</tr>
<tr>
<td>fly-by-wire</td>
<td>267</td>
<td>2459</td>
<td>1.32</td>
<td>12171</td>
<td>10.97</td>
<td>9761</td>
<td>11.16</td>
<td>8751</td>
<td>10.35</td>
<td>28.10%</td>
<td></td>
</tr>
<tr>
<td>adpcm</td>
<td>879</td>
<td>2876</td>
<td>4.82</td>
<td>39088</td>
<td>106.53</td>
<td>33676</td>
<td>118.92</td>
<td>31574</td>
<td>6.24</td>
<td>19.22%</td>
<td></td>
</tr>
<tr>
<td>compress</td>
<td>508</td>
<td>1334</td>
<td>9.18</td>
<td>478191</td>
<td>179.43</td>
<td>31665</td>
<td>204.82</td>
<td>28670</td>
<td>9.46</td>
<td>94.00%</td>
<td></td>
</tr>
<tr>
<td>edn</td>
<td>285</td>
<td>1226</td>
<td>1.47</td>
<td>437158</td>
<td>53.28</td>
<td>437158</td>
<td>676.11</td>
<td>321028</td>
<td>25.63</td>
<td>25.63%</td>
<td></td>
</tr>
<tr>
<td>cnt</td>
<td>267</td>
<td>269</td>
<td>0.17</td>
<td>21935</td>
<td>0.29</td>
<td>21935</td>
<td>0.44</td>
<td>19355</td>
<td>11.76</td>
<td>11.76%</td>
<td></td>
</tr>
<tr>
<td>matmult</td>
<td>163</td>
<td>286</td>
<td>1.75</td>
<td>874348</td>
<td>5.38</td>
<td>874348</td>
<td>6.5</td>
<td>621488</td>
<td>28.92</td>
<td>28.92%</td>
<td></td>
</tr>
<tr>
<td>jfdctint</td>
<td>375</td>
<td>693</td>
<td>0.08</td>
<td>20332</td>
<td>1.02</td>
<td>20332</td>
<td>1.43</td>
<td>17572</td>
<td>13.57</td>
<td>13.57%</td>
<td></td>
</tr>
<tr>
<td>fdct</td>
<td>239</td>
<td>831</td>
<td>0.08</td>
<td>17442</td>
<td>0.05</td>
<td>17442</td>
<td>0.13</td>
<td>14572</td>
<td>16.45</td>
<td>16.45%</td>
<td></td>
</tr>
</tbody>
</table>

TABLE I: Comparing our Algorithm (Unroll_d) to the State-of-the-art
at 31% for insertsort. The extra improvement comes from the integrity of the WCET analysis.

Finally, we like to note one other contribution of our method, which is performing loop unrolling in super-linear time. Our analysis shares this contribution with $\text{AI+SAT}@\text{Unroll}_s$. It is reached through depth-wise loop compression, i.e. reusing across the loop iterations and generating and using compound summarizations where many loop iterations can be summarized at once. This summarization can be used later to avoid exploring large portions of loops. By comparing the size parameter of the benchmarks in table II with the analysis time it can be clearly seen that the super-linear complexity of the brute-force loop unrolling remains in our integrated analysis. For example, in $\text{ns}$ the complexity is $O(n^4)$ while the analysis is performed in a super-linear complexity.

### VI. Related Work

WCET analysis has been the subject of much research, and substantial progress has been made in the area (see [22], [23] for surveys of WCET). Micro-architectural modeling, especially with emphasis on caches, has been an active research topic in WCET analysis. Initial works on instruction cache modeling used integer linear programming (ILP) [24]. However, the work does not scale due to a huge number of generated ILP constraints. Subsequently, the abstract interpretation (AI) [25] framework for micro-architectural modeling, proposed in [2], made an important step towards scalability. The solution has been proved scalable and it has also been applied in commercial WCET tools (e.g., [26]). For most existing WCET analyzers, AI framework has emerged to be the basic approach used for micro-architectural modeling. Additionally, the static analysis for the timing effect of data cache has been investigated in [27], [5], and in [6].

A recent approach [3], [4] has shown some promising results by combining abstract interpretation (AI) with verification technology – model checking and/or SAT solving – for WCET analysis. These works, while possess some forms of infeasible path discovery, essentially still employ a fixed point computation. Therefore, they give a worst-case timing for each basic block, even though the timings of a basic block in different iterations of a loop can diverge significantly.

Similarly, most commercial tools, such as [26], use abstract interpretation (AI) framework for low-level analysis. The worst-case timing for each basic block is then aggregated using the ILP formulation to give the final WCET estimate. In other words, low-level and high-level analyses are performed separately. The immediate benefit is that these tools scale impressively and are applicable to a wide range of input programs. In contrast, our method performs the WCET analysis in one integrated phase, yielding the ideal precision. The main contribution of our paper is to show that integrated analysis can be made scalable, at least for a class of realistic benchmarks. Thus, our method can be employed for applications where precise WCET analysis is pivotal.

We now discuss a number of high-level analysis techniques. Program flow analysis has been well investigated by the WCET research community (e.g. [8], [28]–[32]), among others. These approaches aim to improve the WCET analysis at the program path level, but still suffer from the imprecision in the fixed point computation of the worst-case timing for each basic block in the low-level analysis.

Among the above-mentioned, the most important related work is [8]. Chu et al. [8] proposed a path sensitive loop unrolling algorithm coupled with the concept of interpolation for reuse for scalability. However, the symbolic execution algorithm in [8] works at the level of path analysis only. Given the effect of caches on the basic block timings – making the timings dynamic – [8] alone is no longer effective enough. Our work adopts the concept of interpolation for reuse from [8] and extend it with the concept of dominance. Specifically, by capturing the dominating condition, this paper enables reuse, now under the existence of caches. Another recent approach is [33], where the WCET analysis problem is formulated in the form of an optimization modulo theory (opt).

#### TABLE II: The results of our analysis, $\text{Unroll}_d$, compared to $\text{AI+SAT}@\text{Unroll}_s$ and $\text{AI+SAT}@\text{ILP}$ to Illustrate the Super-linearity of Loop Unrolling

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parameter</th>
<th>AI+SAT, ILP</th>
<th>AI+SAT, Unroll_s</th>
<th>Unroll_d</th>
<th>WCET vs AI+SAT, Unroll_s</th>
<th>WCET vs AI+SAT, ILP</th>
</tr>
</thead>
<tbody>
<tr>
<td>fir</td>
<td>0.09</td>
<td>17977</td>
<td>0.25</td>
<td>14247</td>
<td>0.34</td>
<td>13957</td>
</tr>
<tr>
<td>ud</td>
<td>2.08</td>
<td>33515</td>
<td>1.87</td>
<td>15792</td>
<td>1.77</td>
<td>12132</td>
</tr>
<tr>
<td>janne_complex</td>
<td>0.13</td>
<td>1718</td>
<td>0.11</td>
<td>1219</td>
<td>0.16</td>
<td>1119</td>
</tr>
<tr>
<td>expint</td>
<td>0.14</td>
<td>453451</td>
<td>19.38</td>
<td>16831</td>
<td>28.16</td>
<td>16791</td>
</tr>
<tr>
<td>fft</td>
<td>8</td>
<td>6.43</td>
<td>177059</td>
<td>5.33</td>
<td>74524</td>
<td>3.36</td>
</tr>
<tr>
<td>fft</td>
<td>16</td>
<td>6.43</td>
<td>266821</td>
<td>12.07</td>
<td>143342</td>
<td>10.74</td>
</tr>
<tr>
<td>fft</td>
<td>32</td>
<td>6.43</td>
<td>403179</td>
<td>45.62</td>
<td>280188</td>
<td>48.13</td>
</tr>
<tr>
<td>bubblesort</td>
<td>25</td>
<td>0.14</td>
<td>46706</td>
<td>0.43</td>
<td>46655</td>
<td>0.82</td>
</tr>
<tr>
<td>bubblesort</td>
<td>50</td>
<td>0.14</td>
<td>187656</td>
<td>1.72</td>
<td>187605</td>
<td>3.27</td>
</tr>
<tr>
<td>bubblesort</td>
<td>100</td>
<td>0.14</td>
<td>752681</td>
<td>9.4</td>
<td>752630</td>
<td>19.34</td>
</tr>
<tr>
<td>ns</td>
<td>5</td>
<td>4.94</td>
<td>100416</td>
<td>3.04</td>
<td>100416</td>
<td>0.57</td>
</tr>
<tr>
<td>ns</td>
<td>10</td>
<td>4.94</td>
<td>1506020</td>
<td>3.34</td>
<td>1506020</td>
<td>1.12</td>
</tr>
<tr>
<td>ns</td>
<td>20</td>
<td>4.94</td>
<td>2338610</td>
<td>4.31</td>
<td>2338610</td>
<td>2.71</td>
</tr>
<tr>
<td>insertsort</td>
<td>25</td>
<td>0.12</td>
<td>68969</td>
<td>0.41</td>
<td>10124</td>
<td>0.64</td>
</tr>
<tr>
<td>insertsort</td>
<td>50</td>
<td>0.12</td>
<td>294994</td>
<td>1.94</td>
<td>21124</td>
<td>3.48</td>
</tr>
<tr>
<td>insertsort</td>
<td>100</td>
<td>0.12</td>
<td>1219540</td>
<td>11.95</td>
<td>43124</td>
<td>26.54</td>
</tr>
</tbody>
</table>
problem. This work, sharing the same motivation as [8], uses Craig interpolants to avoid state explosion in solving the op
task. This approach is formulated for loop-free programs, thus unlike [8], it is not suitable for analyzing programs with
nested loops.

Additionally, WCET squeezing [34], presents a post-
analysis method which can integrate ILP-based methods with a
CEGAR refinement loop. It checks the worst-case path
for feasibility by SMT-solving; if the path is infeasible, an
additional ILP constraint is generated to exclude the infeasible
worst-case path.

We also mention parametric methods for WCET anal-
ysis [35], [36]. These methods use a polyhedral abstract
interpretation with symbolic integer-linear programming to
generate parametric WCET expressions. The constraints used
within their approach differ from the standard IPET model.
These expressions in general can become quite complicated
and might need manual simplification to solve them.

The idea of coupling micro-architectural modeling with
path analysis to achieve high precision dates back to [10].
However, to control the state space blow-up, this work has no
mechanism other than path merging. As a result, merging is
performed frequently. It then forfeits the intended precision,
while at the same time, does not scale to realistic benchmarks.
Our work, to the best of our knowledge, is the first scalable
method which integrates micro-architectural modeling with
program path analysis.

In the end, we would like to mention [37], which computes
response times of tasks scheduled by non-pre-emptive fixed-
priority scheduling where execution times depend on history.
This can be similar to the concept of cache summary which
we presented in this article. However, the concept of cache
summary used in this paper foresights the sequence of memory
accesses in a sub-tree based on a similar sub-tree, explored in
the past, which can be quite different from the idea of using
history for task scheduling.

VII. CONCLUSION

We have presented an algorithm for the WCET analysis of
programs with consideration of a cache micro-architecture. At
its core, the algorithm is a symbolic execution, which preserves
the program’s operational semantics in detail, down to the
cache. The only abstraction performed is that the analysis
of one loop iteration is summarized; importantly, the analysis
proceeds precisely across loop iterations. The key challenge,
scalability, was obtained by using a custom notion of reuse.
In realistic benchmarks, it was shown that the extreme attempt
at precision in fact pays off because there was a significant
increase in precision, and this was obtained in a reasonable
time.

REFERENCES

address spaces in real-time operating systems,” in 23rd RTSS ‘02.
prediction by separate cache and path analyses,” RTS 18(2/3) ‘00.
ment of cache timing analysis via model checking,” in RTSS ‘11.
[4] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, “Precise micro-
architectural modeling for wcet analysis via ai++,” in 19th RTAS ‘13.
real-time systems,” in LCTES ‘98.
analysis for WCET estimation,” in RTAS ‘11.
dynamic programming with ad-hoc constraints,” in AAAI ‘08.
for wcet path analysis,” in EMSOFT ‘11.
analysis,” in SAS ‘12.
method based on cycle-level symbolic execution,” RTS ‘99.
WCET ‘06.
presence of timing anomalies,” in WCET ‘09.
sors for software timing analysis,” in RTSS ‘04.
enhanced learning,” in RV ‘14.
interpolation,” in VMCAI ‘07.
language and system,” ACM TOPLAS 14(3) ‘92.
“Papabench: a free real-time benchmark,” in WCET ‘06.
analysis, J. RTS ‘00.
of methods and survey of tools,” TECS ‘08.
model for static analysis,” in POPL ‘77.
[26] “aIT Worst-Case Execution Time Analyzers,” URL http-
:
databases and set-associative caches,” in RTS ‘97.
[28] Y. S. Li and S. Malik, “Performance analysis of embedded software
using implicit path enumeration,” SIGPLAN Not.30(11) ‘95.
[29] A. Ermedahl and J. Gustafsson, “Deriving annotations for tight calcu-
lation of execution time,” in Euro-Par ‘97.
for embedded system C programs,” in WORDS ‘05.
with assertions,” in EMSOFT ‘13.
worst-case execution time by optimization modulo theory and a clever
encoding of program semantics,” in RTNS’14.
[34] J. Koop, L. Kovács, and J. Zwirnmayr, “Wcet squeezing: on-demand
feasibility refinement for proven precise wcet-bounds,” in RTNS ‘13.
analysis.” in WCET ‘03.
parametric wcet calculation,” JSA 57(6) ‘11.
[37] B. Andersson et al., “Non-preemptive scheduling with history-
dependent execution time,” in ECRTS ‘12.