Precise Cache Timing Analysis via Symbolic Simulation

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. In particular, we capture the situations where reuse is applicable by a concept of interpolation. 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 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: 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 an 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 estimations of memory access behavior are joined at the control flow merge points. This often results in an over-estimation of potential cache misses.
- An 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. This partial path captures a subset of all the control flow edges along which the micro-architectural state has been propagated. The partial path is defined as a propositional logic formula 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. 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 control flow graph (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 simulation framework where micro-architectural modeling, and systematic path-sensitivity are synergized. In our algorithm, loops are unrolled and summarized. The essence of our proposed method is that it is fully path-sensitive modulo summarization. This means that the only abstraction performed is within a loop iteration, and

---

1As 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.
2The partial paths are merged to true.
not across loop iterations. This leads to a precise inspection of the timing measured from the underlying hardware model, because the micro-architectural state can be tracked across the iterations. Clearly our method would produce very accurate WCET bounds since we can disregard infeasible states from the iterations.

```
for (i = 0; i < 100; i += 3) {
    if (c) { /* a */ }
else { /* b */ }
}
```

Fig. 1: An Academic Example

Let us consider an academic example in Fig. 1. For the sake of illustration, we assume that variable \( i \) is used in the conditional expression \( c \) and is not modified elsewhere (other than being incremented at the end of each iteration). We need a condition expression \( c \) such that in some iterations, block \( a \) will be executed while in other iterations, block \( b \) will be executed instead. Additionally, we assume a direct-mapped cache, where block \( a \) and block \( b \) access the memory block \( m1 \) and \( m2 \) respectively, and \( m1 \) and \( m2 \) map to the same cache set. In other words, the memory blocks \( m1 \) and \( m2 \) conflict in the cache.

Note that the execution of block \( a \) and block \( b \) are feasible, though in different iterations. A pure A1 approach such as [2] will have to conservatively declare that the fixed point must cache at the looping point contains neither \( m1 \) nor \( m2 \). Thus all the accesses are considered as misses, as if in the case when the conditional expression \( c \) is \((i \% 2 == 0)\).

On the other hand, [4] might improve the analysis by ignoring some "simple" infeasible states from being considered. For example, when the conditional expression \( c \) is \((i < 50)\), [4] can discover that an execution of \( b \) cannot be followed by an execution of \( a \) in the next iteration. In other words, this approach can discover that the access \( m2 \) is indeed persistent [5], [6]. Specifically, only the first access to \( m2 \) is a cache miss, the rest of \( m2 \) accesses are cache hits.

Now consider the case that \( c \) is bit more complicated, for example, \( c \equiv i \% 5 > 1 \). 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, the program is symbolically simulated and loops are unrolled. In the above example, we thus 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.

We note that, we are not the first to perform “integrated path and timing analysis” based on symbolic simulation. In fact, such idea dates back to the attempt by Lundqvist et al. [7]. An integrated method would theoretically give us ideal precision for WCET analysis. However, at the same time, scalability becomes seriously questionable. As an example, [7] does not scale to realistic programs. Thus our main contribution is a dynamic programming algorithm equipped with a powerful notion of reuse. Reuse in turn depends on the core concepts of interpolation and dominance.

In Section IV, we demonstrate both the accuracy and scalability of our algorithm on realistic benchmarks. In all of 13 example programs used, our analysis produced significantly more accuracy, averaging 33%, and with a manageable slowdown, averaging 38 times. 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. Overview Example

Consider the control flow graph (CFG) of a program fragment in Figure 2(a). Each node, shown as a rectangle, abstracts a basic block. Also shown is the program point \((1), (2), \ldots, (10)\) 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 \( m1 \) and \( m2 \) 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 \( m1 \) and \( m2 \) conflict with each other in the cache. We also assume a cache miss penalty of 9 cycles.

Next, in Figure 2(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 simulation of the program. It is important to note that each node in the tree has a context: one part is a collection of logic formulas denoting the relationships between the the program variables associated with each node in the path; the other part is to carry information about the cache. 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. Assume that none of basic blocks modifies the variable \( x \). Then, for example, the context of the leftmost node labelled \( s^{(1)} \) contains both the formula \( x = 0 \) and \( x > 1 \) (amongst possibly many other formulas arising from nodes 1, 2\(^{(1)} \), 4\(^{(1)} \), 5\(^{(1)} \), 7\(^{(1)} \), and therefore is equivalent to false. In other words, the leftmost path in Figure 2(b) is in fact an infeasible path. Note that we have not (fully) drawn the subtree below node \( 4^{(2)} \) in Figure 2(b).

The core technical step in this paper is that of reuse, that is, to reuse the analysis of a subtree, say rooted at program point \( pp \), when encountering another subtree rooted at \( pp \). Thus re-analysis of the second subtree can be replaced by applying the reuse operation, which in general is far cheaper than performing (full) analysis.

To exemplify this, consider the possible reuse of the subtree \( t^{(1)} \) on \( t^{(2)} \). That is, even though we have depicted the subtree \( t^{(2)} \) in full, could we in fact have simply replaced this depiction and instead just indicated that \( t^{(2)} \) is a subtree whose WCET is the same as that of \( t^{(1)} \)? Note the WCET of \( t^{(1)} \) (which only contains one path) is 20 (= 5 + 10 + 0 + 5, since the access \( m2 \) is a hit). Clearly this is not usable in \( t^{(2)} \) (which has two paths). The important point here is the reason: it is that some infeasible path in \( t^{(1)} \) is in fact feasible in \( t^{(2)} \). We can now explain that, in general, we implement this first check for reusability in the form of an interpolant associated to a subtree. This is a formula which the candidate subtree must satisfy in order to be able to reuse the analysis of the interpolated tree. In Figure 2(b), we have indicated the interpolant for \( t^{(1)} \) under the heading \( Int \) and we used the formula \( x = 0 \).
We next discuss the second aspect of reuse, called the *dominating condition*. This concept is to cater for the fact the timing of a path, when caches are taken into consideration, can be dynamic.

Consider the pair of nodes $\langle 4 \rangle^1$ and $\langle 4 \rangle^{(2)}$, and 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 $\langle 4 \rangle^1$ is the (leftmost feasible path, in blue color): $\langle 4 \rangle^1, \langle 5 \rangle^1, \langle 7 \rangle^1, \langle 9 \rangle^1, \langle 10 \rangle^1$. The analysis of $\langle 4 \rangle^1$ 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). Call this distinguished path in $\langle 4 \rangle^1$ its representative path. Still in static timings (e.g. [8]), we can easily see that, if the same representative path is feasible in $\langle 4 \rangle^{(2)}$, that is the path: $\langle 4 \rangle^{(2)}, \langle 5 \rangle^2, \langle 7 \rangle^3, \langle 9 \rangle^3, \langle 10 \rangle^4$, then it is accurate to state that the WCET of $\langle 4 \rangle^{(2)}$ is that of $\langle 4 \rangle^1$.

However, in this paper, we deal with dynamic timings. That is, WCET of a subtree depends on its context. Clearly this a significant conceptual obstacle when dealing with the issue of reuse. In other words, in the example above about reusing the WCET $\langle 4 \rangle^1$ in $\langle 4 \rangle^{(2)}$, this reuse is, in general, unsound because we have not yet ascertained that the representative path in $\langle 4 \rangle^1$ is also a representative path in $\langle 4 \rangle^{(2)}$. This, in turn, is because we have not (yet) examined the context of $\langle 4 \rangle^{(2)}$ in relation to some property of the context of $\langle 4 \rangle^1$. This property is a “dominating condition” which we next exemplify.

In Figure 2(b), note that we have labelled $\langle 4 \rangle^1$ with “Dom: $m_1 \in \text{cache}”$: we have used the label “Int: true” to indicate the interpolant (which is a trivial one, in this case).

Let us refocus on $\langle 4 \rangle^1$ but this time considering the context. A walk through the representative path will show that this path remains representative in the subtree $\langle 4 \rangle^2$ if the context $m_1$ is in the cache. We can now define this to be the dominating condition for the representative path of $\langle 4 \rangle^1$. The key idea is that we can reuse the representative path of $\langle 4 \rangle^1$ on another subtree if the representative path, in the new context, is guaranteed to dominate all other feasible paths. Such guarantee is captured succinctly by the dominating condition.

Now we can exemplify this reuse on the subtree $\langle 4 \rangle^2$. We first check if the context of $\langle 4 \rangle^2$ satisfies the interpolant computed after finishing $\langle 4 \rangle^1$. In this case, the interpolant is true, thus the check trivially holds. We then check that in the context of $\langle 4 \rangle^2$, indeed $m_1$ is in the cache. This condition also holds and so we can reuse the representative path of $\langle 4 \rangle^1$ in relation to some property of the context of $\langle 4 \rangle^{(2)}$. This property is the “dominating condition” which we next exemplify.

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 m1 at (3) 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: (4), (7), (10). Importantly it discovers that at (4), m1 must be in the cache. Thus the access to m1 at (6) is hit. However, at (7), AI has to conservatively declare that neither m1 nor m2 is in the cache. Consequently, the final worst case timings for the basic blocks that have some memory accesses are: (⟨2⟩, 19), (⟨3⟩, 24), (⟨5⟩, 24), (⟨6⟩, 11), (⟨8⟩, 33), (⟨9⟩, 19).

If we aggregate using a path-insensitive high-level analysis, the WCET estimate is 121 (= 10 + max(19, 24) + 10 + max(24, 11) + 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 A1+SAT. Whether [4] can achieve the precision we achieve highly depend on its merging strategies. In theory, [4] can to the extreme and choose not to merge at program point (7), thus it can estimate precisely the timing of basic block (9). 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 imprecise analysis, as we will demonstrate experimentally in Section IV.

III. GENERAL FRAMEWORK

We propose a symbolic simulation framework where micro-architectural modeling, and systematic path-sensitivity are synergized. Our symbolic simulation framework adopts the concepts of loop unrolling and summarizing, interpolation for reuse from [8]. In the analysis of loop-free programs our algorithm is fully path-sensitive. Additionally, for the programs with loops, we inherit from the loop handling mechanism in [8].

However, the framework from [8] does not take into account the effect of micro-architectural modelling on the WCET analysis. The timing of a basic block can differ up to an order of magnitude based on the internal state of the hardware. There are features in complex architectures which make the analysis of the WCET much more difficult. These features include caches, pipelines, etc. These features rely on some hidden state within the processor which will cause the number of possible states to grow exponentially. A precise WCET estimation needs to account the effect of the internal states of the hardware on the generated timing. Mehner et al. [1] show that the impact of caches on the real-time behaviour of programs is much more than the other features. This is the extension of our framework. In order to improve the estimated WCET, our algorithm integrates the effect of both the data and instruction caches on the obtained WCET. Additionally, the summarizations generated for the explored sub trees needed to be enhanced to be able to consider the effect of the caches on the obtained WCET. Our framework uses enhanced summarizations to be able to generate dynamic timing. Note that the summarizations in [8] where generating static timing.

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. We follow the standard semantics of abstract cache for must analysis, formally defined in [2].

Following [8], we model a program by a transition system. A transition system $P$ is a tuple $⟨L, l_0, \rightarrow⟩$ where $L$ is the set of program points, $l_0 \in L$ is the unique initial program point. Let $\rightarrow \subseteq L \times L \times Ops$, where Ops is the set of operations, be the transition relation that relates a state to its (possible) successors by executing the operations. 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 Vars. An assignment $x := e$ corresponds to assign the evaluation of the expression $e$ to the variable $x$. In the assume operator, assume$(cond)$, if the conditional expression cond evaluates to true, continues, otherwise it halts. We shall use $l \xrightarrow{op} l'$ to denote a transition relation from $l \in L$ to $l' \in L$ executing the operation op $\in Ops$.

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. One advantage of representing a program using transition systems is that the program can be executed symbolically in a simple manner.

**Definition 1 (Symbolic State).** A symbolic state $s$ is a quadruple $⟨l, c, σ, Π⟩$ where $l \in L$ corresponds to the concrete 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 the path condition Π is a first-order logic formula over the symbolic inputs which accumulates constraints the inputs must satisfy in order for an execution to reach this state (following some program path).

Let $s_0 \equiv ⟨l_0, c_0, σ_0, Π_0⟩$ 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 $s \equiv ⟨l, c, σ, Π⟩$, the evaluation $[\cdot]σ$ of an arithmetic expression $e$ in a store $σ$ is defined as usual: $[v]σ = σ(v)$, $[n]σ = n$, $[e + e']σ = [e]σ + [e']σ$, $[e - e']σ = [e]σ - [e']σ$, etc. The evaluation of conditional expression $[c]σ$ can be defined analogously. For convenience, the set of symbolic states are denoted as SymStates.

Given a program point $l$, an operation $op \in Ops$, and a symbolic store $σ$, the function $acc(l, op, σ)$ denotes the sequence of memory block accesses by executing op at the symbolic state $s \equiv ⟨l, c, σ, Π⟩$. While the program point $l$ identifies the instruction cache access, the sequence of data accesses are obtained by considering both $op$ and $σ$ 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 [6] for loading memory ranges into the cache, 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 1.** In our symbolic execution framework, both add and load instructions are modelled as assignments. While the
former involves no data access, the later is used to load to 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 \(\langle \mathcal{L}, l_0, \rightarrow \rangle\), a transition system, and a symbolic state \(s \equiv \langle l, c, \sigma, \Pi \rangle \in \text{SymStates}\), the symbolic execution of transition \(\tau : l \overset{op}{\rightarrow} l'\) returns another symbolic state \(s'\) defined as:

\[
\begin{align*}
    \Gamma & \equiv \begin{cases} (l', c', \sigma, \Pi \land \text{cond}) & \text{if } op \equiv \text{assume(\text{cond})} \\ (l', c', \sigma[x \mapsto [c]\sigma], \Pi) & \text{if } op \equiv x := e \end{cases} \\
\end{align*}
\]

where \(c'\) is the new abstract cache, derived from \(c\) and the sequence of accesses \(\text{acc}(l, op, \sigma)\) using the standard the update function from the abstract cache semantics for must analysis from \([2]\) where \(c' \equiv U(\text{acc}(l, op, \sigma), c)\).

Abusing notation, the execution step from \(s\) to \(s'\) is denoted as \(s \overset{\tau}{\rightarrow} s'\). For convenience, when there is no ambiguity, we just refer to the symbolic state \(s\) using the tuple \(\langle l, c, s\rangle\) where \(s\) is the constraint component of the symbolic state \(s\), obtained by projecting the symbolic store onto the set of program variables. A path \(\pi \equiv s_0 \rightarrow s_1 \rightarrow \ldots s_m\) is feasible if \(s_m \equiv \langle l, c, s\rangle\) and \(s\) is satisfiable. Otherwise, the path is called infeasible and \(s_m\) is called an infeasible state. Here we query a theorem prover to another symbolic state \(s\) and \(s\) is always infeasible, \(s_m\) is the representative path, and \(s_m\) is the path that \(\pi\) is post-dominates \(s\).

For the purpose of WCET analysis, we also include in the set of program variables the special timing variable \(t\). Our analysis computes a sound and accurate bound for \(t\) in the end, across all feasible paths of the program. Note that \(t\) is always initialized to 0 and the only operations allowed upon it are concrete increments. Given a symbolic state \(s \equiv \langle l, c, \sigma, \Pi \rangle\) and a transition \(\tau : l \overset{op}{\rightarrow} l'\), the amount of increment at \(s\) by executing \(\tau\) will be evaluated by the execution time of the low-level instructions and the access time of \(\text{acc}(l, op, \sigma)\) on the cache state \(c\). The timing variable \(t\) is not used in any other way.

**B. Summarization and Reuse**

Given the definitions in the previous Section, a symbolic execution tree can be obtained in an obvious manner. However, it is not possible to naively constructing 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 key challenge remain: how to make it scale.

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 II, for each finite subtree, already analyzed, the reuse condition is captured succinctly using an interpolant and a dominating condition. Given a new symbolic state, if it satisfies the interpolant, it follows that all the infeasible paths discovered in the previously analyzed subtree will be maintained at the reuse point. On the other hand, if the state satisfies the dominating condition, the dominating path, therefore longest, in the analyzed subtree, 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 \([9]\).

**Definition 3.** [Interpolant]. Given two first-order logic formulas \(F\) and \(G\) such that \(F \models G\), then there exists a Craig interpolant \(H\) denoted as \(\text{Intp}(F,G)\), 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 Craig 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.

\([8], [10]\) are pioneer works applying Craig interpolation for program analysis. In the context of program analysis, the problem is formulated so that scalability can be achieved by reusing previously computed sub-results. 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.

**Definition 4.** [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 \(\ell_2\) wrt. an incoming symbolic state \(s\). The summarization of this subtree is defined as the tuple \(\langle \ell_1, \ell_2, \Omega, \Delta, \Psi, \delta \rangle\), where \(\Omega\) is the representative path, \(\Delta\) is an abstract transformer capturing the abstract input-output relation between variables and the cache state at \(l_1\) and \(l_2\). Moreover, \(\delta\) is a dominating condition, and finally, \(\Psi\) is an interpolant, i.e., a condition under which this summarization can be safely reused.

By definition, the abstract transformer \(\Delta\) \([8]\) will be the abstraction of all feasible paths (wrt. the incoming symbolic state \(s\), also called the incoming context) from \(\ell_1\) to \(\ell_2\). In our implementation, the abstract transformer for the program variables is computed using the polyhedral domain. On the other than, the abstract transformer for the abstract cache component, which we name it cache summary, is computed. This abstract transformer is the summary of the memory accesses between the program points \(pc_1\) and \(pc_2\), such that if applied to \(c_1\) (the cache state at \(pc_1\)), will generate \(c_2\) (the cache state at \(pc_2\)), as if we had explored the paths between \(pc_1\) and \(pc_2\). Note here that, in general, abstract transformer is not a functional relation.

The cache summary is generated in a recursive manner meaning that the cache summary of a node is generated based on the cache summary of its children nodes. A cache summary for a set-associative cache is a list of \(N\) set summaries, where \(N = \text{Capacity}/(\text{Block Size} \times \text{Associativity})\). The set summary is a tuple \((\text{list}, n)\) which contains first the list of memory blocks loaded to the cache in a sub tree and second the number \(n\) which denotes the maximum relocation of the items already in the cache. Each memory block in \(\text{list}\) has a location \(\text{loc}\) attached to it which indicates the location that the block should be positioned when the summary is applied to a cache state. The cache summaries are combined and merged recursively. In the process of generating the cache summaries only the memory blocks which their location is less than or equal to the cache associativity are stored. The rest will
be naturally, pushed out of the cache. As a result, with this combination method, while our analysis remains sound, the size of the cache summaries will remain constant compared to the size of the sub-trees.

The **representative path**, is the sequence of the program points demonstrating the longest path in an explored sub-tree. The representative path is stored in the summarization and can replay the sequence of accesses dynamically at the time of reuse. As a result the timing obtained for the representative path is dynamic and can alter based on the incoming context. It should be noted that in order to store the representative path we do not need to store the whole sequence in the representative path and only the ones needed to check the feasibility of the representative path should be stored. Moreover, some of the accesses to the memory can be resolved to always cache hits or always cache/misses. These access can be resolved regardless of the input cache state\(^3\). These sources of time remain fixed during the analysis and their timing are stored statically in the representative path. Summarization of a sub tree is important for summarizing loop iterations. However, we need to define the summarization of a program point too.

**Definition 5.** [Summarization of a Program Point]. A summarization of a program point \(\ell\) is the summarization of all paths from \(\ell\) to \(\ell'\) (wrt. 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 \(\ell'\).

In the framework presented in [8], [11], the reuse of a summarization is performed as long as the interpolant is satisfied and the representative path is feasible. However, we needed to extend the conditions for reuse since the representative path can be used to obtain the timing of a sub tree while it remains the longest path in a sub-tree. In other words, the representative path should dominate all other paths in the sub-tree with regard to the incoming context. In order to reach this confidence the concept of **dominating condition** is introduced.

The dominating condition is an extension over the concept of witness path from [8]. With the extension, the enhanced dominating condition now checks if the representative path is feasible in the incoming context and if it still dominates all the other paths with regard to the incoming context.

**Definition 6** (Dominating Condition (\(\bar{\delta}\))). The dominating condition for a representative path is the set of conditions that while satisfied guarantee that the respective representative path is feasible and dominates all other paths inside a sub-tree.

The dominating condition with regard to the cache state assures the soundness of reusing a summarization by checking two conditions on the incoming cache state at the time of reuse. First, it assures that the list of the memory blocks in the incoming cache state is a superset of the list of the memory blocks in the cache state of the previously explored sub tree. Even if the list of memory blocks are a superset of the memory blocks of in the cache state of the previously explored sub tree, since with the LRU cache policy, we need to make sure that the memory blocks remain in the cache at least for the same amount of time as before and are not pushed out of the cache too early. So, secondly, the cache dominating condition checks that the location of the memory blocks in the incoming cache are at most the same as the location of the memory blocks in the cache state at the time that the summarization was generated. The dominating condition over the cache state will perform these checks by the help of the \(loc\) function. The \(loc\) function maps the memory blocks to their locations in the cache state. For example by the help of the \(loc\) function and a constraint like \(loc(m_i) \geq \text{Associativity}\) we can reason that a certain memory block like \(m_i\) is in the cache or not.

The dominating conditions and the representative paths are generated recursively. In the first step, based on the maximum possible timings of the representative paths the longer representative path is chosen and reported as the representative path. Secondly, the dominating condition should be updated such that it can guarantee that while satisfied, the chosen representative path will always dominate all the other paths in the sub tree. The cache dominating condition should contain the reason for dominance of the both representative paths and the reason that the chosen representative path dominates the other one. We start with the conjunction of the dominating condition of the two representative paths, hoping it would be enough to infer the dominance of the chosen representative path. If yes, we will return the conjunction of the cache dominance conditions. Otherwise, we will start strengthening the cache dominating condition by the help of the \(loc\) function over the memory blocks in the cache up to the point that the chosen representative path dominates the other path. Due to the static number of the memory accesses this process always terminate.

Moving back to the symbolic simulation framework, a new node \(j\), such that \(i\) and \(j\) associate to the same program point \(\ell\), will not be further expanded while the following conditions are satisfied and the timing for node \(j\) will be generated from the representative path at node \(i\) and the incoming context of node \(j\):

1. Its incoming context over the \(s_j\) is less general than a previously stored interpolant \(\bar{s}_i\) in the program dominating condition i.e. \(s_j = \bar{s}_i\)
2. The representative path is still feasible in the incoming program context.
3. The dominating condition is still satisfied in the incoming context.

With this, we finish this section that, by adopting the concepts of interpolation and enhancing or introducing the concepts of representative path, dominating condition and cache summary, our framework became capable of generating summaries which in turn can be used to generate the timing of an analysed program based on both the program variables and the cache state.

IV. **EXPERIMENTAL EVALUATION**

We used a 2.3 GHz Pentium core i7 machine with 4 GB of RAM. We implemented our WCET analysis on top of the LLVM IR [12], due to the simplicity and wide use in literature. The LLVM instructions are simulated for a reduced instruction set computing (RISC) architecture with the size of 4 bytes. Additionally, the cache settings for the experiments were adopted from the 2014 WCET tool competition [13], ARM9 target processor.

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

---

\(^3\)Similar to the idea of the Always Hit/Always Miss/Persistent categories in the AI framework presented in [2]
### Table I: The results of our analysis, Unroll\(_d\), compared to AI+SAT⊕Unroll\(_s\) and AI+SAT⊕ILP

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AI+SAT⊕ILP</th>
<th>AI+SAT⊕Unroll(_s)</th>
<th>Unroll(_d)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s) WCET</td>
<td>Time(s) WCET</td>
<td>Time(s) WCET vs AI+SAT⊕Unroll(_s) vs AI+SAT⊕ILP</td>
</tr>
<tr>
<td>tcas</td>
<td>0.26 1427</td>
<td>6.2 1212</td>
<td>11.32 1116</td>
</tr>
<tr>
<td>nsichneu</td>
<td>74.38 85125</td>
<td>1744 66308</td>
<td>2544.59 48388</td>
</tr>
<tr>
<td>lir</td>
<td>0.08 17977</td>
<td>1.1 16109</td>
<td>1.5 15597</td>
</tr>
<tr>
<td>fft1</td>
<td>8 1.68 219707</td>
<td>1.82 95194</td>
<td>3 92084</td>
</tr>
<tr>
<td>ud</td>
<td>5 0.98 16705</td>
<td>0.98 15792</td>
<td>1.36 12262</td>
</tr>
<tr>
<td>compress</td>
<td>4.86 218836</td>
<td>122 30215</td>
<td>143.06 28190</td>
</tr>
<tr>
<td>ndes</td>
<td>4.8 304519</td>
<td>16.2 174416</td>
<td>26.38 154247</td>
</tr>
<tr>
<td>fly(_b)y(_w)ire</td>
<td>0.62 12161</td>
<td>8.3 9751</td>
<td>16.44 8751</td>
</tr>
<tr>
<td>edn</td>
<td>0.82 437298</td>
<td>550 437298</td>
<td>715.24 320858</td>
</tr>
<tr>
<td>cnt</td>
<td>10 0.08 21935</td>
<td>0.28 21935</td>
<td>0.32 19345</td>
</tr>
<tr>
<td>matmult</td>
<td>20 1.1 822348</td>
<td>4.34 822348</td>
<td>3.08 573378</td>
</tr>
<tr>
<td>jfdctint</td>
<td>8 0.06 20332</td>
<td>0.84 20332</td>
<td>1.54 17572</td>
</tr>
<tr>
<td>fdct</td>
<td>8 0.02 17442</td>
<td>0.06 17442</td>
<td>0.12 14572</td>
</tr>
</tbody>
</table>

- **Unroll\(_d\)** implements the WCET analysis presented in this article. It performs WCET analysis in one integrated phase, coupling the micro-architectural analysis with path analysis. Moreover, Unroll\(_d\) is able to generate dynamic timings from the reused summarizations.

- **AI+SAT⊕ILP** is the implementation of the WCET analysis from [4], which benefits from the state-of-the-art method in micro-architectural modelling combined with ILP formulation. The ILP formulation was only provided with the set of the loop bounds of the benchmarks and no information on the infeasible paths where provided to the ILP formulation. We shall call this the baseline analysis.

- **AI+SAT⊕Unroll\(_s\)** is an implementation of a hypothetical algorithm. This WCET analysis was constructed to benefit from the state-of-the-art micro-architectural modelling method from [4], and the state-of-the-art path analysis method from [8]. The WCET estimations generated by AI+SAT⊕Unroll\(_s\) will be the most precise result which the modular approaches performing micro-architectural modelling and the path analysis in separate phases can reach. AI+SAT⊕Unroll\(_s\) generates static timings from the reused summarizations. So the purpose of this analysis is to show that our algorithm, in Unroll\(_d\), still surpasses a direction combination of two superior existing methods.

For all our experimental results, we measure the WCET improvement as \(\frac{B - U}{U} \times 100\%\), where \(U\) is the WCET obtained using our analysis, and \(B\) is the WCET obtained using the baseline approach.

The set of the benchmarks used in the experiments can be seen in table I. In order to have a realistic set of experiments we have performed our experiments on three groups of benchmarks:

1) **Benchmarks with lots of Infeasible Paths:** The first group contains “tcas” and “nsichneu” from [14]. “tcas” is a loop-free program with many infeasible paths and “nsichneu” is an auto-generated code which contains a long loop with many infeasible paths which iterates twice. Due to the large number of infeasible paths in these benchmarks, the analysis of these benchmarks can determine the precision of our proposed analysis compared to the other two analyses.

2) **Benchmarks with Complicated Loops and Infeasible Paths:** This group of benchmarks contains standard programs from [14] and “fly by wire” from [15] which are mainly used for timing analysis. The benchmarks in this group contain complicated loops and infeasible paths.

3) **Benchmarks with Simple Loops:** This group contains a set of academic programs from [14] which do not contain infeasible paths or complicated loops but contain memory accesses which might resolve to a range of memory addresses.

Each group of the benchmarks in table I, are separated from the other groups by the double lines. An important note is that since some of the benchmarks from [14], they have variable initializations which can affect the timings generated by the micro-architectural modelling in the WCET analysis. In order to be able to have a fair comparison between the three WCET analyses we have removed the initializations which did not have affect on the feasibility of the paths.

Now we each group of the benchmarks separately. In the first group, the WCET estimated by Unroll\(_d\) compared to AI+SAT⊕ILP in average is improved by 32% which increases up to 43% for “nsichneu”. Moreover, the WCET obtained by Unroll\(_d\) compared to AI+SAT⊕Unroll\(_s\) in average is improved by 17%. Note that, “nsichneu” contains a long loops with many infeasible paths. The improvement of the WCET estimation comes from the detection of infeasible paths. Moreover, “tcas” is a loop-free program and the improvement in the obtained WCET is coming directly from the infeasible path detection. We highlight that, although the infeasible paths detected by the Unroll\(_d\) and AI+SAT⊕Unroll\(_s\) are the same, there still is an average of 17% improvement in the estimated WCET of Unroll\(_d\) compared to AI+SAT⊕Unroll\(_s\). This clearly comes from the use of dynamic timing.

The benchmarks in the second group show the maximum improvement in the measured WCET. The WCET obtained by Unroll\(_d\) compared to AI+SAT⊕ILP is improved by 43.5% on average, and peaks at 87%. Moreover, the WCET obtained by Unroll\(_d\) compared to AI+SAT⊕Unroll\(_s\) is improved by 9%
on average, and peaks at 22%. Some of the benchmarks in this group "fir", "fft1", "compress" 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 "ud". These two benchmarks, while containing complicated loops, possess the least improvement between the benchmarks in this group. This can confirm that the provided loop bounds to the ILP formulation were quite tight. On the other hand, "fft1" and "compress" show the maximum improvement of the wcet estimation by the Unroll_d analysis compared to AI+SAT ⊕ILP. Our investigation reveals that some infeasible paths in these two benchmarks contain loops, which due to the infeasibility of the paths do not contribute to the WCET of these benchmarks. By detecting these infeasible paths, the measured WCET improves dramatically. However, the fixed point computation is unable to eliminate this effect. This can be observed as a contribution of our method in precisely detecting long infeasible paths which are not detectable by the fixed point computation. Moreover, the same observation is made in "ndes" which falls in the third place of the highest improvement in the comparison of Unroll_d and AI+SAT ⊕ILP.

In the comparison of the results of Unroll_d and AI+SAT ⊕Unroll_s for this group, the WCET estimation of "ud", "ndes" and "fly-by-wire" is improved more compared to the other benchmarks. Our observation reveals that these three benchmarks have more memory accesses which are resolved to address ranges in the memory. Unroll_d benefits from dynamic timing which can more effectively cope with such sources of imprecision.

Furthermore, the effect of these 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 for the benchmarks in this group is improved 19% on average, and peaks at 30%. The benchmarks in this group do not contain infeasible paths or complicated loops, but contain memory accesses which are resolved to address ranges in the memory. While, our analysis, Unroll_d, is able to remain precise in the analysis of these benchmarks, both of the AI+SAT ⊕ILP and AI+SAT ⊕Unroll_s analyses are unable to confront this source of imprecision. For these benchmarks, the estimated WCET by Unroll_d receives its improvements only from the improvement in the precision of memory accesses.

The WCET estimations from Unroll_d prevail the results from the other two analyses in all benchmarks. The WCET estimations from Unroll_d are improved 32% on average compared to AI+SAT ⊕ILP and 14% on average compared to AI+SAT ⊕Unroll_s. These improvements clearly uphold our proposal that performing the WCET analysis in one integrated phase with the use of dynamic timing in the summarizations will result in the maximum precision.

On the other hand, "nischneu", containing many paths is used as an indicator of the scalability of the WCET tools, and we can state that our method is scalable since it is able to analyse "nischneu".

Finally, we would like to indicate that the result of our analyses, Unroll_d, is plainly demonstrating the importance of performing the WCET analysis in one integrated phase for the precision of the WCET estimation. The improvement in the precision comes from the fact that the effect of infeasible path detection can be reflected in the micro-architectural modelling.

V. RELATED WORK

WCET analysis has been the subject of much research, and substantial progress has been made in the area (see [16], [17] 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) [18]. However, the work does not scale due to a huge number of generated ILP constraints. Subsequently, the abstract interpretation (AI) [19] framework for micro-architectural modeling, proposed in [2], has made an important step towards scalability. The solution has been proved scalable and it has also been applied in commercial WCET tools (e.g., [20]). 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 [21] and [5]. The later article extends their previous AI framework [2] approach to analyze data caches. Recently, an enhancement on the data cache analysis is presented by Huynh et. al. 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 [20], 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. On the other hand, 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.

Program flow analysis has also been well investigated by the WCET research community (e.g. [8], [11], [22]–[25]). These approaches aim to improve the WCET analysis at the program path level. Among them, 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 simulation algorithm in [8] does not consider the effect caches on the estimated WCET. Given the effect of caches on the basic block timings – making the timings dynamic – [8] is no longer applicable. 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.

The idea of coupling micro-architectural modelling with path analysis to achieve high precision dates back to [7]. This work, however, has no mechanism other than path merging in order to control the state space blow-up. 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.

VI. 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 simulation 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. It follows that the algorithm is the most precise. 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 reasonable time.

REFERENCES