Context-sensitive Parametric WCET Analysis

Clément Ballabriga, Julien Forget, and Giuseppe Lipari

Université Lille, CRIStAL, UMR 9189, 59650 Villeneuve d'Ascq, France
{Clement.Ballabriga,Julien.Forget,Giuseppe.Lipari}@univ-lille1.fr

Abstract

In this paper, we propose a WCET analysis that focuses on two aspects. First, it supports context-sensitive hardware and software timing effects, meaning that it is sensitive to the execution history of the program and thus can account for effects like cache persistence, triangular loop, etc. Second, it supports the introduction of parameters in both the software model (e.g. parametric loop bounds) and the hardware model (e.g. number of cache misses). WCET computation by static analysis is traditionally handled by the Implicit Path Enumeration Technique (IPET), using an Integer Linear Program (ILP) that is difficult to resolve parametrically. We suggest an alternative tree-based approach. We define a context-sensitive CFG format to express these effects, and we provide an efficient method to process it, giving a parametric WCET formula. Experimental results show that this new method is significantly faster and more accurate than existing parametric approaches.

1998 ACM Subject Classification C.3 Real-time and embedded systems

Keywords and phrases Parametric, WCET, Real-time, Static analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2015.55

1 Introduction

In static WCET analysis methods, an upper bound to the WCET of a task is traditionally computed in three steps. First, the task code is statically analyzed to model the set of possible execution paths. Then, the hardware is taken into account by modeling the architectural effects: local effects (timings of basic blocks) and global effects (interactions between basic blocks). Finally, the WCET computation takes as input a program and its environment (hardware and software), and produces the WCET. A popular technique for doing this last step is IPET [14], in which the WCET computation is represented as an ILP problem solving.

With traditional WCET computation, if the program, input values, software environment or hardware platform changes, it is necessary to re-run the entire analysis. On the opposite, *parametric* WCET analysis takes a parametrized input, and produces a formula that depends on those parameters. If the parametric values change, it is possible to compute a new WCET simply by evaluating the formula on the new values. This offers several benefits, which we detail below.

First, parametric WCET avoids re-running the entire WCET computation each time there is a minor change to the program or hardware configuration. This is an important aspect, due to the increasing size of real-time systems and to the non-linear complexity of WCET analyses. Similarly, parametric WCET simplifies the analysis process when third-party software is involved, since the developer can provide a parametric WCET that can be adapted to the target system (software and hardware).

Second, many system parameters are only known at run-time: loop bounds that depend on input values, software and hardware state changes, operating system interference, etc. Using a parametric WCET, it is possible to evaluate the WCET formula at run-time and

15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015).
Editor: Francisco J. Cazorla; pp. 55–64

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

take better decisions accordingly. For instance, tighter WCET evaluation at run-time could benefit energy-aware scheduling techniques based on Dynamic Voltage and Frequency Scaling (DVFS) [12].

Finally, large execution time values may happen only very rarely, for instance for unlikely combinations of input data. By using parametric WCET, it is possible to design the system according to an upper bound that is safe for the vast majority of executions of the system, and then evaluate a parametric WCET formula at run-time to trigger an alternate less time-consuming computation when the formula returns a value exceeding the safe bound (and thus remain under the safe bound).

In this work we present a novel approach to parametric WCET analysis. Unlike the majority of existing approaches, our methodology is not based on ILP. To the best of our knowledge, it is the first to benefit from (1) a reasonable computing time, since it runs in polynomial time, and (2) a good precision thanks to the support of *context-sensitive effects* (persistent cache blocks, non-rectangular loops, branch prediction, etc.). Furthermore, in our approach the trade-off between computation time and precision is configurable: it is possible to increase (or decrease) the context sensitivity to improve the precision (or speed up the computation time).

2 Related works

In [1], a technique is presented to perform a partial, composable WCET analysis. This work addresses mostly the software and hardware modeling that occurs before the WCET computation proper. Results are presented for the instruction cache and branch prediction analysis, and loop bounds estimation. However, no solution is provided to perform the ILP computation parametrically.

Feautrier [8] presented a method for parametric ILP computation. A traditional ILP solver takes an ILP system as input, and provides a numerical solution corresponding to the maximization (or minimization) of an objective function. The ILP solver presented in [8] (called *PIPLib*), takes a parametrized ILP system as input, and produces a *quast* (quasi-affine selection tree). Once computed, this tree can be evaluated for any valid parameter values, without having to re-run the solver. However, this approach is computationally very expensive. In contrast, our approach has polynomial complexity, and it is therefore scalable, whereas the introduced pessimism is very small, as it will be shown in Section 6.

In theory, PIPLib could be applied for solving parametrically the ILP systems produced in the context of the IPET method [14]. However, experimentations [4] have shown that it does not scale well: the parametric ILP solving may become intractable for medium to large size programs. The MPA (Minimum Propagation Algorithm) [5, 4] attempts to address the shortcomings of PIPLib in the context of the IPET method. MPA takes as input the results of the software and hardware modeling analysis, and produces directly a parametric WCET formula. Compared with MPA, our method is significantly tighter because it takes into account various context-sensitive software and hardware timing effects.

In the past, many tree-based WCET computation methods have been presented [11]. In [7] the authors suggest a method to compute the WCET parametrically using a tree-based approach. Our parametric approach is also tree-based, but unlike the one presented in [7], it can work directly on the binary target (no source code needed). Furthermore, our method can model timing effects in a more generic and accurate way than existing tree-based approaches, and the trade-off between accuracy and computation time can be configured.

ParaScale [12] is an approach to exploit variability in execution time to save energy. By statically analyzing the tasks, a parametric WCET formula is given for loops in terms of the

loop iteration count. At run-time, before entering a loop, the formula is evaluated and the system dynamically scales the voltage and frequency of the processor. In comparison, our parametrization is not limited to loop bounds, but can be used for anything influencing the WCET (such as cache misses, branch predictor states, etc.). Furthermore our approach uses a more refined model for loops.

Finally, note that our method provides an alternative to the time-consuming ILP solving, thus our method is competitive even compared to non-parametric WCET analysis based on ILP.

3 Context-sensitive model

Our algorithm takes as input a *context-sensitive Control Flow Graph* (CFG) and produces as output a parametric formula. Let us motivate the need for the context-sensitive CFG as our algorithm input by using two examples.

First we consider the instruction cache analysis by categorization. In this approach, blocks can be categorized as *persistent* with respect to a loop (for the sake of simplicity, we assume that each basic block matches exactly a cache block), meaning that the block will stay in the cache during the whole execution of the loop (only the first execution results in a cache miss). In the CFG shown in Figure 1(a), if we assume that Block 3 is persistent, its execution time depends on whether it has already been executed or not. With IPET-based approaches, this information would be stored as an ILP constraint, but since we do not use IPET, we need to store it in the CFG itself.

As a second example, let us consider a triangular loop: a *for* loop $i = 1..10$, containing an inner *for* loop $j = i..10$. The maximum iteration count for each loop is 10, but the inner loop body can be executed at most $\sum_{i=1}^{10} i$ times. Once again, this cannot be expressed with the traditional CFG. In both examples, there is a block of code whose execution time depends on the number of times this block was executed after entering some loop containing it: this is what we call the *execution context*.

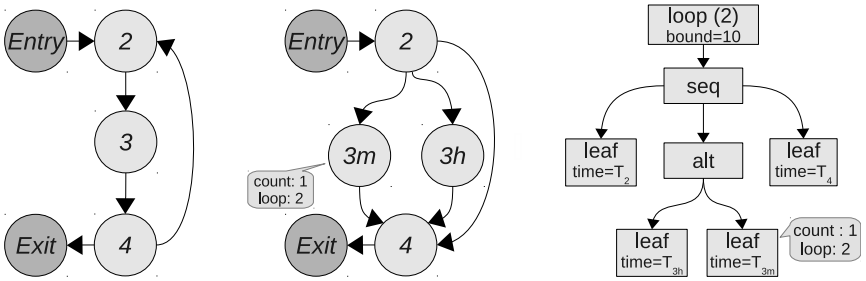
3.1 Context-sensitive CFG

In IPET-based WCET analyses, the various preliminary analyses (such as cache block categorization) produce ILP constraints. In our approach, instead we transform the program CFG into a *context-sensitive* CFG (and possibly make other graph transformations such as block duplication).

The context-sensitive CFG extends the standard CFG by allowing an (optional) context annotation associated with each node. For any CFG G, let L_G denote the set of its loops. A loop is defined as the set of cycles in the CFG sharing the same loop header. A loop header is defined as a node n having a predecessor p such that n dominates p. For any header h, we will note L_h the loop having the header h, and for any node n, we will note L_n the loop immediately containing n (i.e. L_n is the loop containing n such that there is no other loop containing n, and whose header is inside L_n). Furthermore, let us define a partial order on loops: $\forall (L_1, L_2) \in L_G \times L_G, L_2 < L_1$ if (and only if) L_2 is contained in L_1 (i.e. if there exists a path from the header of L_1 to itself, going through the header of L_2).

A context-sensitive CFG G is defined by $G =< \mathcal{B}_G, \mathcal{E}_G, \mathcal{A}_G >$, where $\mathcal{B}_G$ is the set of nodes (basic blocks), $\mathcal{E}_G$ is the set of edges, and $\mathcal{A}_G = L_G \times \mathcal{B}_G \times \mathbb{N}$ is the set of context annotations. We will note T_b the execution time of basic block b.

A context annotation $a = (a_l, a_b, a_n)$ represents a restriction on the set of feasible paths in the CFG: the basic block a_b (inside a_l) may be executed at most a_n times each time the loop a_l is entered.



(a) Standard CFG. **(b)** Context-sensitive CFG. **(c)** Expression tree.

Figure 1 Program representations.

To model our first example (cache persistence), we split Block 3 from Figure 1(a) as two (virtual) blocks $3h$ and $3m$, representing respectively the hit and miss. Then we add a context annotation to Block $3m$, as shown in Figure 1(b). This represents the fact that $3m$ can be executed only once per execution of the loop. To model the triangular loop example, we can add Annotation $(L_{outer}, body, \sum_{j=1}^{10} j)$, where L_{outer} represents the outer loop, and *body* is the block inside the inner loop.

These annotations are intended to be a generic tool to model many WCET-related effects (hardware, and software), therefore the exact way to generate those annotations will depend on the effect we want to model (and on the underlying analysis). However, the algorithm evaluating the context-sensitive CFG is generic and does not need to know which types of effects are represented by the annotations.

If a CFG node contains no annotation, its execution time is context insensitive (i.e. it is the same for all contexts). In this trivial case, its WCET is an integer. Otherwise, its execution time depends on the context. We define the context of a node n as the history of *context events* happening before the execution of n. The set of context events is defined as $V = \{exec(n) | n \in \mathcal{B}_G\} \cup \{loop(l) | l \in L_G\}$, where $exec(n)$ represents the execution of node n, and $loop(l)$ represents the entrance in loop l containing (not necessarily directly) n. Then, a context is defined as a list of context events. We denote an empty context as ϵ. We also use the classical list notation $\mathsf{head} \cdot \mathsf{tail}$ to denote a context whose first element is head, followed by list tail. For any context c and any integer k we note c^k the repetition k times of sequence c. We note $range[L_1, L_n]$ the sequence $loop(L_1) \cdot ... \cdot loop(L_n)$ where $\forall i, L_1 < L_i < L_n$. Similarly, the notation $range[L_1, L_n[$ corresponds to $loop(L_1) \cdot ... \cdot loop(L_{n-1})$. Finally, the set of contexts is noted C. For instance, the context of the third execution of the block $3m$ (in loop body) in Figure 1(a) would be $loop(L_2) \cdot exec(3m) \cdot exec(3m)$ (the loop represented in the figure is noted L_2, as its header is the basic block 2).

3.2 Expression tree

The execution time of a node depends on the node itself and on the context. Therefore, it is possible to compute an unique execution time for any $(context, node)$ pair. To provide a practical and easily computable approach, we will first convert the context-sensitive CFG into an *expression tree*, then evaluate this tree to produce the WCET. To perform this conversion, an algorithm similar to the one presented in [6] can be used. An expression contains a set $\mathcal{T}$ of tree nodes which can be of type *alt*, *seq*, *loop* and *leaf*. A leaf node n represents a single basic block, and has an attribute storing the basic block execution time (noted n_{time}). A seq node represents a sequence of child nodes, and an alt node represents a choice

(alternative) between child nodes. A loop node represents a loop in the program, has exactly one child node (representing the loop body), and has an attribute (noted n_b) representing the maximum iteration count. In addition, each context annotation in the original CFG is copied to the corresponding leaf node in the tree[1]. Figure 1(c) shows the tree that would be generated from the context-sensitive CFG represented in Figure 1(b).

3.3 Abstract WCET values

An abstract WCET represents a set of possible execution times for a tree node, along with conditions required for that execution time to occur. In our cache example, the abstract WCET value computed for the alt node would contain two execution times (miss case, and hit case) and indicate that the miss case can occur only the first time Block 3 is accessed.

First, we introduce the concept of *context mapping*. A context mapping $m = (m_{loop}, m_{time})$ is a pair whose first element (m_{loop}) represents a loop, and second element (m_{time}) represents an execution time. The set of context mappings is noted $M = L_G \times \mathbb{N}$. We define a total non-strict order on M such that $\forall (m,n) \in M \times M, m \geq n \iff m_{time} \geq n_{time}$. An abstract WCET $\alpha = (\alpha_{map}, \alpha_{other})$ is a pair, in which the first element (α_{map}) is a multiset over M, and the second element (α_{other}) is an execution time. The set of abstract WCETs is noted $A = M^{\#} \times \mathbb{N}$, where $M^{\#}$ is the sets of multisets over M. For any multiset m over M, we note $1_m : M \to \mathbb{N}$ its multiplicity function. Furthermore, we define the $\max_k : M^{\#} \to M^{\#}$ function, returning the k greatest elements of $M^{\#}$ (context mappings with same time but different loop are considered equivalent by this function).

An abstract WCET α is computed for each node in the tree. Informally, the presence of a context mapping m in α_{map} means that the node may have an execution time of m_{time}, but only once each time m_{loop} is entered. The α_{other} value represents the default execution time of the node (used whenever no other time can be used due to context). We lift the concept of context over CFG nodes to tree nodes. The new set of contexts is defined as $V' = \{exec(n) | n \in \mathcal{T}\} \cup \{loop(l) | l \in L_G\}$.

The abstract WCET of a tree node n provides a mapping from contexts to execution times. We define two functions, $eval : C \times A \times \mathcal{T} \to \mathbb{N}$, and $next : C \times A \times \mathcal{T} \to \mathbb{N}$. Let n be a tree node, α its associated abstract WCET, and c a context for the tree node n. The expression $eval(c, \alpha, n)$ gives the WCET corresponding to the execution of the event sequence (loop entrances, and executions of n) represented by c, while $next(c, \alpha, n)$ gives the WCET increase caused by a subsequent execution of n after the event sequence represented by c (which is not necessarily equal to the WCET of this last execution of n). The function $next$ is defined such that $\forall \alpha, next(c, \alpha, n) = eval(c \cdot exec(n), \alpha, n) - eval(c, \alpha, n)$, and $eval$ is defined as follows:

$$eval(c, \alpha, n) = \alpha_{other} \times \max(0, i - |\Theta|) + \sum_{t \in \max_n(\Theta)} t$$

where i is the number of $exec(n)$ present in c, and Θ is such that $\forall m \in M, 1_\Theta(m) = 1_{\alpha_{map}}(m) \times |\{k | c_k = loop(m_{loop})\}|$. The general idea behind the $eval$ function is to account for the execution time of n for each $exec(n)$ present in the context, while ensuring that each context mapping is not used more times than the number of times its corresponding loop is entered, and using α_{other} to provide a time for n when no context mapping can be

[1] It is possible to extend the CFG annotations to sub-graphs representing if or loop structures (as opposed to single blocks), this is not described here for the sake of brevity.

used. The *eval* function may compute an over-approximation for some contexts, however this representation is quite compact, with many contexts (leading to the same execution time) described by a single context mapping value.

4 WCET computation

Let us define some notations that we will use in this section. The binary operator $\uplus$ is defined such that for any multiset pair (m, n), $\forall x, 1_{m \uplus n}(x) = 1_m(x) + 1_n(x)$. Similarly to the set-builder notation, we define the multiset-builder notation $[n|n \in m \wedge pred(n)]$, representing the multiset containing all the elements in m satisfying $pred(n)$. For any element e in a multiset, and any $n \in \mathbb{N}$, we will note $e \otimes n$ the multiset containing only the element e, with a multiplicity of n.

4.1 Node evaluation

We note ω the evaluation function, taking as input a tree node and producing an abstract WCET. Once the tree evaluation is finished, and we have the abstract WCET corresponding to the root node *root*, the concrete WCET is $\omega(root)_{other}$. We detail below a simple way to compute function ω.

For a leaf node n, if the node has no context annotation then $\omega(n) = (\emptyset, n_{time})$, otherwise, $\omega(n) = (\{(a_l, n_{time})\} \otimes a_n, 0)$, where $(a_l, a_b, a_n) \in A$ is the context annotation associated with the leaf node (a_b is the basic block represented by leaf node n).

For a seq node n with two children n_1, n_2 (this can be extended to an arbitrary number of children, since the operation is associative), we have $\omega(n) = (map, other)$ such that:

$$other = \omega(n_1)_{other} + \omega(n_2)_{other} \quad map = \biguplus_{l \geq L_n} \mathcal{S}(\omega(n_1), \omega(n_2), range[l; L_n])$$
$$\text{with } \mathcal{S}(\alpha, \alpha', c) = \biguplus_{i=0}^{m-1} (l, next(c \cdot exec(n_1)^i, \alpha, n_1) + next(c \cdot exec(n_2)^i, \alpha', n_2))$$
$$\text{and } m = \max(|\alpha_{map}|, |\alpha'_{map}|)\,.$$

The general idea is to match context mappings from both children, and add their times. For example, if $\omega(n_1) = ([(L_1, 10), (L_1, 8)], 5)$ and $\omega(n_2) = ([(L_2, 4)], 3)$, and $L_2 < L_1$, then $\omega(n) = ([(L_1, 10+4), (L_1, 8+3), (L_2, 5+4)], 5+3)$.

For a alt node n with two children n_1, n_2 (this can be generalized in the same way as the seq node), $\omega(n) = (map, other)$, such that:

$$other = \max(\omega(n_1)_{other}, \omega(n_2)_{other})$$
$$map = [m|m \in (\omega(n_1)_{map} \uplus \omega(n_2)_{map}) \wedge m_{time} > other]$$

For any loop node n representing loop l, with body n_1, $\omega(n) = (map, other)$ such that:

$$other = eval(\sigma(l, n_1, n_b), \omega(n_1), n_1) \quad map = \biguplus_{l' > l} \biguplus_{i=1}^{j} (l', \tau(i, n_1, n_b, range[l'; l[))$$
$$\text{with } \tau(i, n, b, c) = eval(c \cdot \sigma(l, n, b)^i, \omega(n), n) - eval(c \cdot \sigma(l, n, b)^{n-1}, \omega(n), n)$$
$$\text{and } \sigma(l, n, k) = loop(l) \cdot exec(n)^k, \quad j = min(\{i|\tau(i, \alpha, n_1, b, c) \leq other\})\,.$$

For example, if we consider a loop l having a body with abstract WCET $\omega(n_1) = ([(L_1, 8), (L_1, 7)], 6)$ and loop bound 5, inside an outer loop L_1, then $\omega(n) = ([(L_1, 8+7+6\times3)], 6\times5)$.

The abstract WCET value corresponding to the seq node of Figure 1(c) is $([(L_2, T_2 + T_{3m} + T_4)], T_2 + T_{3h} + T_4)$: the first time the seq node is executed after entering loop 2, its WCET value is $T_2 + T_{3m} + T_4$, however for subsequent executions it is $T_2 + T_{3h} + T_4$. The abstract WCET value corresponding to the loop node is $(\emptyset, 10 \times (T_2 + T_4) + 9 \times T_{3h} + T_{3_m})$: each time we enter the loop, a miss will occur at most once (Block $3m$) and other iterations will be hits (Block $3h$). Note that, since there is no context mapping in the abstract WCET (the multiset is empty), it is equivalent to a static WCET.

4.2 Approximations

In the presence of many context annotations, tree nodes can have many possible execution times, thus the evaluation can produce very large context mappings. Using measurement results from various experiments, we have observed that most of the time, the function that maps iteration counts to execution times can be tightly over-approximated by a much simpler piecewise linear function. The presence of a straight-line section in this piecewise linear function means that there is a group of context mappings in the corresponding abstract WCET value, with a time approximately equal to the slope of the straight-line section. Such a group of i context mappings can be merged into one context mapping with a multiplicity of i, and a time equal to the maximum time of the former group.

Using such an approximation, we lose some precision but we reduce the amount of data we will have to process. We can only merge context mappings referring to the same loop. Therefore, in order to increase the merging possibilities, for any context mapping $(L_1, t) \in M$, it is possible to replace it safely by (L_2, t), with $L_2 < L_1$. Of course this would cause a loss of precision, but it allows more merging and reduce the complexity of the evaluation.

The greater the number of context mappings we use, the greater the precision and the analysis time. Therefore, when evaluating the tree, at each node we may need to use heuristics to decide when and how to perform an approximation. For the experiments performed in this paper, we use a simple (yet quite effective) strategy: we perform an approximation whenever the number of distinct context mappings exceeds a certain (user-configurable) threshold.

5 Parametric WCET computation

Our tree-based computation is the first step for parametric computation. It can be made parametric in a much easier way than an ILP computation (while it is true that ILP solving can be done parametrically, it is way more costly than our method), as we will show in this section. For instance, considering the example of Figure 2 and assuming that the loop bound is not known statically, we will show how to create a parametric WCET in terms of the loop bound value and how to obtain the concrete WCET once the loop bound value is known.

We extend our expression tree model, to enable the introduction of parameters that represent information unknown at static analysis time:

- We introduce a new node type: a *param* node n has an attribute n_{param} representing a parameter identifier. Such a node can represent any type of (statically unknown) expression sub-tree. It can be used to perform modular analysis (to represent the abstract WCET for a separately-analyzed library call for which we do not have the code).
- In loop nodes, the maximum iteration count, which was previously stored as an integer, can now be a parameter identifier, to represent a statically unknown loop bound.
- In any context annotation (a_l, a_b, a_n) associated to a node, the values a_l and a_n can now be parameter identifiers. This could be used to support parametric cache categories.

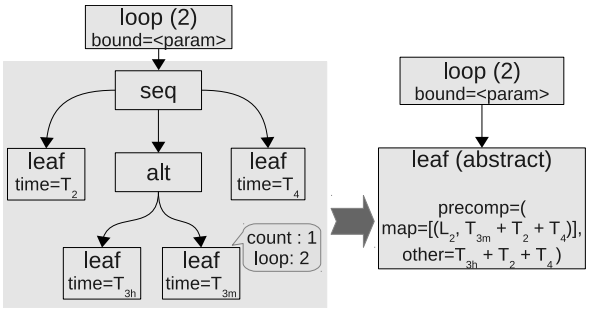


Figure 2 Partial evaluation.

5.1 Partial evaluation

The parametric WCET computation starts with a partial evaluation phase, which precomputes as much as possible from the parametric tree, and produces a simplified parametric tree.

We introduce two separated kinds of leaf nodes: a leaf node n can either be concrete (associated with a basic block), or abstract (with an attribute $n_{precomp}$ of type A, abstract WCET). The abstract leaf node holds the result of a precomputed sub-tree. The evaluation function ω for an abstract leaf node is defined such that $\omega(n) = n_{precomp}$.

A node is *parametric* if it (or any of its descendants) contains a parameter. To partially evaluate the parametric tree, we select any non-parametric non-abstract node n, remove it (and its descendants), and replace it by an abstract leaf node n' such that $n'_{precomp} = \omega(n)$. This is repeated until the tree contains only parametric nodes, and abstract leaf nodes. Some optimizations can also be applied (not detailed here) to remove unneeded nodes.

In our example, the result of the partial evaluation is shown in the right side of Figure 2, containing only one parametric loop node, and an abstract leaf node holding the abstract WCET corresponding to the loop body: the first time the loop body is executed, its time is $T_{3m} + T_2 + T_4$, subsequently its time is $T_{3h} + T_2 + T_4$.

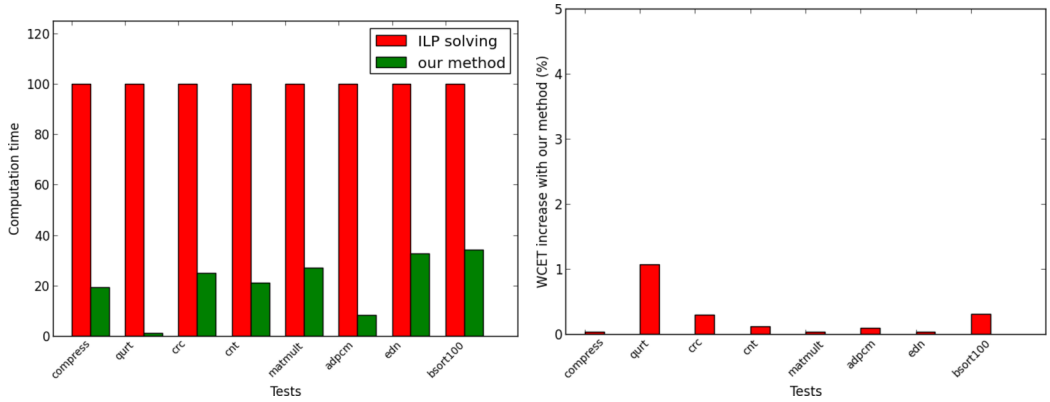
5.2 Parameter instantiation

The next step is the parameter instantiation, which takes as input a simplified parametric tree and parameter values. To do the parametric instantiation, we replace, in the parametric tree, each param node n by the value of n_{param}, and each parameter p present in a loop bound or in a context annotation by their value. Once the resulting tree contains no parametric node, it can be evaluated using the method from Section 4, producing the WCET.

In our example, if we want to instantiate the simplified tree with a loop bound of 10 for L_2, we replace $< param >$ with 10. Let us call r this partial result after replacement. We can evaluate it, giving the abstract WCET $\omega(r)$, from which we can get the concrete WCET: $\omega(r)_{other} = T_{3m} + T_{3h} \times 9 + 10 \times (T_2 + T_4)$.

6 Experiments

To evaluate the performance of our approach, we compared it to existing IPET-based WCET analysis. The target hardware is an ARM processor with a set-associative LRU instruction cache (the data cache is not taken into account). The processor pipeline is analyzed with the exegraph method [13] and the instruction cache is modeled using cache categorization [9]. We perform our analysis on a subset of the Mälardalen benchmarks [10], used as standalone



(a) Analysis time comparison. **(b)** Pessimism measurement.

Figure 3 Experimental results.

tasks, without any modeling of the operating system. To perform the preliminary steps of the WCET analysis (program path analysis, CFG building, loop bounds estimation, pipeline and cache modeling), we used OTAWA, an open source WCET computation tool [2]. Then, we compare our approach with an ILP approach (using the GNU lp_solve ILP solver [3]), by running both on the result produced by OTAWA.

We first compare the ILP solving time and the time taken by our tree evaluation (Figure 3(a)). The times are normalized so that the ILP time is always 100. The measurements do not include the preliminary WCET analyses (performed by OTAWA) as they are common to both approaches. On average, our approach reduces the analysis time by a factor of 15.7. The running time of our approach is polynomial in the tree size as long as we merge context mapping groups once their size exceeds a fixed threshold.

Our approach uses simplifications that discard information, which introduces pessimism. To quantify it, we choose a large loop in each benchmark, and create a parameter representing its iteration count. We run our parametric analysis on each benchmark, instantiate the result for each possible parameter value, and we evaluate the pessimism by comparing the result to the value obtained by IPET. Figure 3(b) shows the average WCET increase for each benchmark. We can see that the pessimism increase is very reasonable (on average 0.25%).

The pessimism can be attributed to two main causes: (1) the reduced expressiveness of our annotated CFG format (as opposed to an ILP system), and (2) the approximations performed during our computation (such as the simplification by merging context mappings presented in Section 4.2). We cannot express all types of infeasible paths, such as mutually exclusive paths, although we plan to support this in the future.

7 Conclusion

In this paper, we have presented a new approach to perform the final step of WCET computation, which replaces the ILP solving phase of the IPET method. Instead of computing a single, fixed WCET value, our method computes a WCET formula that may depend on parameters, such as, for instance, loop bounds, architectural entry states (cache or branch predictor state), or system environment (for example, preemption count). The WCET formula can be evaluated once the parameter values are known. Additionally, it has a significantly faster computation time, even compared to non-parametric WCET computation

methods, while retaining good precision, and supports various WCET features such as branch prediction, cache analysis, and non-rectangular loops.

The main limitation of our method is the diminished expressiveness (compared to ILP approaches), for instance in presence of certain types of infeasible paths. In future works, we plan to work on this issue, and enable our context-sensitive CFG format to represent all characteristics found in modern WCET analyses.

References

1 Clément Ballabriga, Hugues Cassé, and Marianne De Michiel. A Generic Framework for Blackbox Components in WCET Computation. In *9th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2009.

2 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*. Springer, 2010.

3 Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system. `http://lpsolve.sourceforge.net/5.5/`.

4 S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric wcet calculation. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2009.

5 Stefan Bygde. *Parametric WCET Analysis*. PhD thesis, Mälardalen University Press, 2013.

6 Cristina Cifuentes. A structuring algorithm for decompilation. In *XIX Conferencia Latinoamericana de Informática*, 1993.

7 Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *14th Euromicro Conference on Real-Time Systems (ECRTS)*, Washington, DC, USA, 2002. IEEE Computer Society.

8 Paul Feautrier. Parametric integer programming. *RAIRO Operations Research*, 22, 1988.

9 Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2):163–189, 1999.

10 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks – past, present and future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010.

11 Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An Accurate Worst Case Timing Analysis for RISC Processors. In *Real-Time Systems Symposium (RTSS)*, pages 97–108, 1995.

12 S. Mohan, F. Mueller, W. Hawkins, M. Root, C. Healy, and D. Whalley. Parascale: exploiting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *Real-Time Systems Symposium (RTSS)*, 2005.

13 Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. In *Transactions on High-Performance Embedded Architectures and Compilers II*, volume 5470 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009.

14 Yau tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Real-Time Systems Symposium (RTSS)*, 1995.