Symbolic WCET Computation

Clément Ballabriga, Julien Forget, Giuseppe Lipari

Symbolic WCET computation

CLÉMENT BALLABRIGA, JULIEN FORGET, and GIUSEPPE LIPARI, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL, France

Parametric Worst-case execution time (WCET) analysis of a sequential program produces a formula that represents the worst-case execution time of the program, where parameters of the formula are user-defined parameters of the program (as loop bounds, values of inputs or internal variables, etc).

In this paper we propose a novel methodology to compute the parametric WCET of a program. Unlike other algorithms in the literature, our method is not based on Integer Linear Programming (ILP). Instead, we follow an approach based on the notion of symbolic computation of WCET formulae. After explaining our methodology and proving its correctness, we present a set of experiments to compare our method against the state of the art. We show that our approach dominates other parametric analyses, and produces results that are very close to those produced by non-parametric ILP-based approaches, while keeping very good computing time.

Additional Key Words and Phrases: Worst-case execution time, symbolic evaluation

1 INTRODUCTION

A real-time system is usually represented as a set of tasks. Tasks are subject to timing constraints: typically, the execution of every instance of a periodic real-time task must be completed before its *deadline*. In order to guarantee the respect of timing constraints, first a *worst-case execution time* (WCET) analysis is performed off-line, which calculates an upper bound to the execution time of each task. Then, this information is used to perform a schedulability analysis and guarantee that every task will meet its deadline.

In this paper, we focus on WCET analysis. In WCET analysis, first the task code is analysed to model its set of possible execution paths. Then, the impact of the hardware architecture is taken into account: local effects (timing of basic blocks of code) and global effects (impact of processor pipeline, caches, and in general interactions between basic blocks). Finally, an upper bound to the execution time is computed by calculating the worst-case path, taking into account all effects. A popular technique for doing this, called Implicit Path Enumeration Technique (IPET), is to encode the problem as an Integer Linear Programming (ILP) problem that is then solved with standard techniques [16].

With traditional WCET analysis, if any of the program parameters is changed, it is necessary to re-run the analysis. Also, it is difficult to analyze the impact of different parameter values on the final WCET estimate. For example, the developer may be want to know the impact of the number of iterations of a certain loop on the WCET, the impact of the cache size, etc. To answer these questions, it would be necessary to run the analysis several times with different parameter values, which could be a very time consuming process.

An alternative approach is to calculate directly a *parametric WCET* formula instead of a constant value. If the parameter changes, it is possible to recompute the WCET by simply substituting the parameter value into the formula. Thus, it is possible to quickly explore the parameters space, which may be very useful in guiding developers at design

time. Similarly, parametric WCET simplifies the analysis process when third-party software is involved, since the developer can provide a parametric WCET along with the component, that can be adapted to the target system.

In addition, if the obtained formula is simple enough, it can be used to efficiently implement an *adaptive* real-time system. Indeed, 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. With traditional WCET analysis, adaptive features would rely on a pre-computed WCET table containing different WCET values for different parameter values. Instead, with parametric WCET analysis, we can compute off-line a WCET formula that depends on these parameters and instantiate this formula on-line, at which point parameter values become known. As a result, with low overhead, we obtain a tighter estimate of the task's WCET and take better scheduling decisions. This can for instance benefit energy-aware scheduling techniques based on Dynamic Voltage and Frequency Scaling (DVFS) [18].

Finally, large execution time values may happen only very rarely, for instance for unlikely combinations of input data. By using parametric WCET analysis, 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).

Contribution. In this paper, we propose a novel approach to parametric WCET analysis based on *symbolic computation* that greatly improves upon the state of the art on parametric WCET. Unlike the majority of existing WCET analysis algorithms, our methodology is not based on ILP: instead, we follow an approach based on symbolic computation of WCET formulae.

We start from a representation of the program as a *Control-Flow Graph* where nodes of the graph are basic blocks of code (the notion of CFG is recalled in Section 3). We transform the CFG into a *Control-Flow Tree* (CFT) (Section 4), because a tree is more amenable to be transformed into arithmetic (symbolic) formulae. To represent global effects, CFT nodes are annotated with *context-sensitive annotations* (Section 5): these annotations encode restrictions on the number of iterations of basic blocks when executed inside loops. They may be considered as the equivalent of ILP constraints in the IPET method [16]. We then move to the core method for generating a WCET formula. We first introduce the notion of *Abstract WCET* (Section 5.2) and how to compute it starting from an annotated CFT in the absence of parameters. Later, we introduce WCET parameters (Section 6) and we enunciate the rules for symbolic computation and simplification of *Abstract WCET formulae*. Finally, in Section 7 we present experimental data that compare our approach with the state of the art algorithms. We show that our algorithm produces results that are very close to those of non-parametric ILP-based approaches, while keeping very good computing time. We also show that simplified WCET formulae are very small, which implies low memory and execution time overhead in case of on-line formula evaluation. Finally, we show that our approach dominates other parametric WCET analyses. This paper focuses on the generic framework for symbolic WCET evaluation and only briefly outlines some applications in Section 6.2. More complex applications (e.g. data-cache analysis) are out of the scope of this paper and are subject to future work.

2 RELATED WORKS

Various existing works suggest using symbolic methods in WCET analysis. However, their goal differs from ours. For example, [4, 6, 8] use symbolic execution as a method to reduce the duration of the WCET analysis. In [21], the authors use symbolic states to model the effect of pipelines on the WCET. The objective of these papers is not to produce a parametric WCET formula.

In [2], 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 [11] presented a method for parametric ILP computation. The ILP solver presented in [11] (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. Experiments [7] have shown that PIPLib does not scale well when applied in the context of IPET. The MPA (Minimum Propagation Algorithm) [7] attempts to address these shortcomings. 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 [17]. In [10], the authors suggest a method to compute parametric WCETs using a tree-based approach. Our approach is also based on trees, but unlike [10] it can work directly on the binary code. Furthermore, our method can model timing effects in a more generic and accurate way thanks to context annotations (Section 5).

ParaScale [18] 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, the parameters in our method are not limited to loop bounds.

3 CONTROL-FLOW GRAPH

In this section we recall the definition of Control-Flow Graphs (CFG), the input model in our approach. The CFG is extracted from the binary code of the task under analysis.

Definition 1. *A Control Flow Graph (CFG) is a directed graph* $G =< \mathcal{B}, \mathcal{E} >$. *The set of vertices* $\mathcal{B}$ *corresponds to the set of basic blocks of the program represented by the CFG. A directed edge* $(b_i, b_j) \in \mathcal{E}$ *(where* $\mathcal{E} \subseteq \mathcal{B} \times \mathcal{B}$*), represents a valid succession of two basic blocks in the program execution. We denote by* $\mathsf{time}(b)$ *the worst-case execution time (WCET) of block* b.

An *entry node* is a node without incoming edges, and an *exit node* is a node without outgoing edges. We assume, without loss of generality, that a CFG has one single entry node and one single exit node (otherwise, it is always possible to add fictive entry and exit nodes with the corresponding edges). We also assume that each node is reachable from the entry node, and that the exit node is reachable from any node.

An *execution path* is a sequence of nodes (basic blocks): $p.b$ denotes a path whose last node is b; $p_1@p_2$ denotes the path consisting of path p_1 followed by path p_2. By abuse of notation, we also denote b the path consisting only of node b. ϵ denotes the empty path.

Definition 2. *Let* $G =< \mathcal{B}, \mathcal{E} >$ *be a CFG. Let* $p = b_1 \dots b_k$ *an execution path. We say that* p *is a* valid path *of* G *(or simply a path of* G*) iff:*

$$\forall i \in \{1, \dots, k\}, b_i \in \mathcal{B} \wedge \forall i \in \{1, \dots, k-1\}, (b_i, b_{i+1}) \in \mathcal{E}$$

If b_1 is an entry node of G and b_n is an exit node of G, then p represents a complete execution of the program represented by G.

Definition 3. *Let $p = b_1 \dots b_k$ an execution path. We have:* $\text{time}(p) \equiv \sum_{i=1}^{k} \text{time}(b_k)$

We introduce now a set of additional definitions concerning the CFG topology that will allow us to manipulate the CFG in the following sections.

Definition 4. *Let $G =< \mathcal{B}, \mathcal{E} >$. Let $b_i, b_j, h \in \mathcal{B}$ and let h be a loop header (see definition below).*

- *We say that b_i is a* predecessor *of b_j, and denote $b_i \rightarrow b_j$, iff $(b_i, b_j) \in \mathcal{E}$;*
- *We say that b_i* dominates *b_j, and denote $b_i \gg b_j$, iff all paths from the entry node to b_j go through b_i;*
- *b_k is the* immediate dominator *of b_i iff $b_k \gg b_i$, $b_k \neq b_i$ and there exists no $b_{k'}$ such that $b_{k'} \neq b_i$, $b_{k'} \neq b_k$, $b_{k'} \gg b_i$, $b_k \gg b_{k'}$;*
- *h is a* loop header *if it has at least one predecessor b_i such that $h \gg b_i$. We denote l_h the loop associated to header h;*
- *An edge (b_i, h) such that $h \gg b_i$ is called a* back-edge *of l_h;*
- *An edge (b_i, h) that is not a back-edge is called an* entry-edge *of l_h.*
- *The* body *of the loop of header h, denoted $body(h)$, is the set of all nodes b_i such that b_i belongs to a path P, where P starts with h, ends with a back-edge of l_h and does not go through any entry-edges of l_h.*
- *An edge (b_i, b_j) such that $b_i \in body(h)$ and $b_j \notin body(h)$ is called an* exit-edge *of l_h;*
- *An execution path of a loop l_h is a path $p = h.b_1 \dots b_n$, where there exists an exit edge (b_n, b_x) of l_h. Note that b_1, b_n, h may actually not be distinct. The number of iterations of l_h in p corresponds to the number of back-edges in p. The maximum number of iterations of the loop l_h, denoted by x_h, is the maximum of the number of iterations of any execution path of l_h.*
- *Let l_h, $l_{h'}$ be two loops of G. We say that l_h* contains *$l_{h'}$ and denote $l_{h'} \sqsubseteq l_h$ iff $h' \in body(h)$;*
- *The loop l_h* immediately contains *b_i iff $b_i \in body(h)$ and there exists no loop $l_{h'} \neq l_h$ such that $b_i \in body(h')$ and $l_{h'} \sqsubseteq l_h$.*
- *The set of loops of graph G is denoted L_G.*

We define two additional loops, that are not actually part of the represented program:

- $\top$ is such that for all $l \in L_G$, $l \sqsubseteq \top$. In other words, $\top$ is a fictive loop whose body is the whole CFG ($body(\top) = G$);
- $\bot$ is such that for all $l \in L_G$, $\bot \sqsubseteq l$. In other words, $\bot$ is a fictive empty loop ($body(\bot) = \emptyset$).

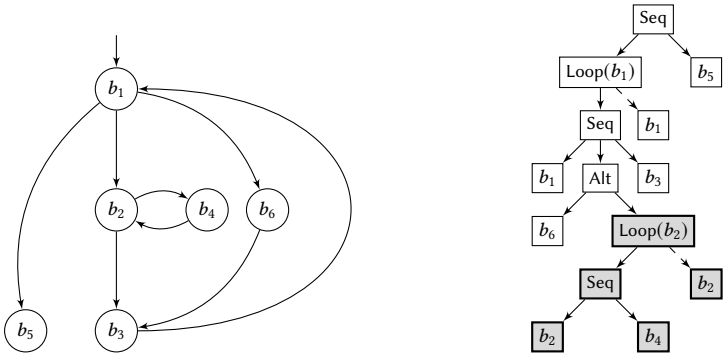
Property 1. *$(L'_G = L_G \cup \{\top, \bot\}, \sqsubseteq)$ is a lattice.*

Proof. Trivial due to the definition of $\top$ and $\bot$. □

In the following:

- $\sqcup : L'_G \times L'_G \rightarrow L'_G$ denotes the least upper bound, i.e. $l_1 \sqcup l_2$ is the least element of $\{l \in L'_G | l_1 \sqsubseteq l \wedge l_2 \sqsubseteq l\}$.
- $\sqcap : L'_G \times L'_G \rightarrow L'_G$ denotes the greatest lower bound, i.e. $l_1 \sqcap l_2$ is the greatest element of $\{l \in L'_G | l \sqsubseteq l_1 \wedge l \sqsubseteq l_2\}$.

Figure 1a shows of a simple CFG. Nodes b_1 and b_2 are loop headers. Loop l_{b_1} contains b_1, b_2, b_3, b_4, b_6, but it immediately contains only b_1 and b_3. (b_3, b_1) is a back-edge and (b_1, b_5) is an exit-edge for loop l_{b_1}. Loop l_{b_2} is contained within loop l_{b_1}. b_1 dominates all the other nodes of the CFG. b_1 is the immediate dominator of b_3.



(a) CFG representation (b) CFT representation

Fig. 1. A program with two nested loops.

4 CONTROL-FLOW TREE

We propose to translate the CFG into a *Control-Flow Tree*, which also represents the possible execution paths of a program but, thanks to its tree structure, is more prone to recursive WCET analysis than a CFG. A Control-flow Tree is similar to Abstract Syntax Trees used in programming languages compilation, except that it represents the structure of binary code. As such, it will be quite natural to represent the WCET of a CFT as an arithmetic expression (see Section 6).

4.1 Definition

The set of Control-flow Trees $\mathcal{T}$ is defined inductively as follows:

Definition 5. *Let $n, m \in \mathbb{N}^*$, $t_1, \ldots, t_n, \in \mathcal{T}^n$. A control-flow tree $t \in \mathcal{T}$ is one of:*

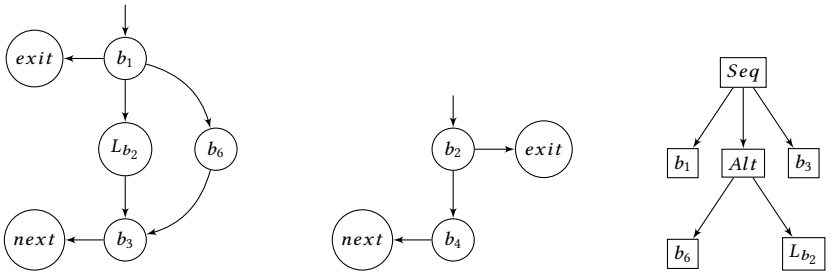
- *Leaf(b), which represents the execution of basic block $b \in \mathcal{B}$;*
- *Alt($t_1, \ldots, t_n$), which represents an alternative between the execution of trees $t_1, \ldots, t_n$;*
- *Loop(h, t_1, n, t_2), which represents a loop with header h, that repeats the execution of tree t_1, with a maximum number of iterations n, and exits from the loop executing the tree t_2;*
- *Seq($t_1, \ldots, t_n$), which represents a sequential execution of trees $t_1, \ldots, t_n$.*

As an example, Figure 1b shows the tree corresponding to the CFG of Figure 1a. In the following sections, we will use this example to describe the steps of the conversion from CFG to CFT. Our definition of loops considers that we repeat a sub-tree and then execute a different sub-tree when finishing the loop. This enables to represent a wide variety of loops: *for*, *while*, *do...while*, etc.

4.2 From CFG to Control-flow Tree

Algorithm 1 translates a loop of the CFG into a *Directed Acyclic Graph* (DAG) that represents the loop body. Algorithm 2 is the recursive procedure that generates the complete control-flow tree. It relies on Algorithm 1 to process the CFG loops.

Our control-flow tree construction method works only for CFGs that contain no irreducible loops (i.e. loops with multiples entries). In the general case, it is possible to transform CFGs with irreducible loops by using *node splitting* [14]



(a) DAG for loop l_{b_1} (b) DAG for loop l_{b_2} (c) CFT for the body of loop l_{b_1}

Fig. 2. From loop to DAG and CFT

algorithms. In [20] the authors show that it is possible to detect the set of irreducible loops in a CFG in $O(n^2)$. While the complexity of the node-splitting algorithm is not reported, the algorithm is meant to be executed only on irreducible loops, which usually constitute a small part of the analysed program.

4.2.1 Loop to DAG (Algorithm 1). The DAG produced for a loop l_h represents its body. In this DAG, inner loops are replaced by *hierarchical nodes*, which themselves correspond to separate DAGs. For instance, Figure 2a shows the DAG produced for loop l_{b_1} (the construction steps and the meaning of nodes *exit* and *next* are detailed below). L_{b_2} is a hierarchical node representing loop l_{b_2}. The DAG produced for loop l_{b_2} is shown in Figure 2b. In the remainder of this section, we use the example of Figure 2a to illustrate Algorithm 1.

Algorithm 1 constructs the DAG corresponding to a loop l_h. At line 2 the algorithm adds all nodes immediately contained in l_h to the DAG nodes. Any edge in the CFG between these nodes is added to the DAG edges (line 3). In our example, this corresponds to nodes b_1, b_3, b_6 and to edges (b_1, b_6) and (b_6, b_3).

Virtual *exit* and *next* nodes are created to represent, respectively, transferring control to the next iteration and exiting the loop (lines 4 and 5). For any back-edge (b_i, b_j) of l_h in the CFG, we add a corresponding edge in the DAG, from b_i to the virtual next node (line 6). Similarly, for any exit edge (b_i, b_j) of l_h in the CFG we add a corresponding edge in the DAG, from b_i to the virtual exit node (line 7). In our example, we have an edge (b_1, exit) and an edge (b_3, next).

Inner loops are handled by the *for* in lines 9–14. For each loop $l_{h'}$ directly in l_h, we create a *hierarchical node* $L_{h'}$. For each exit edge (b_i, b_j) of l'_h, an edge $(L_{h'}, b_j)$ is created (line 11) and for each entry edge (b_i, b_j), an edge $(b_i, L_{h'})$ is created (line 12). In our example, a hierarchical node L_{b_2} is created to represent the loop l_{b_2} (which is directly in loop l_{b_1}) and we also create edges (b_1, L_{b_2}) and (L_{b_2}, b_3).

We assumed in Section 3 that the whole CFG is the body of a (fictive) loop $\top$. Therefore, the whole CFG can also be transformed into a DAG using Algorithm 1. It produces a hierarchy of DAGs corresponding to the CFG containing only reducible loops.

Note that similar algorithms have been proposed in [22]. However, the most notable difference between the work presented in [22] and our approach, is that while our transformation may not preserve the semantics of the program, we guarantee that it does not decrease the execution time. On the contrary, the method proposed in [22] guarantees the preservation of the program semantics, but not the execution time.

Algorithm 1 Loop to DAG

1: **function** DAG($G =< \mathcal{B}, \mathcal{E} >, l_h \in L_G \cup \top$)
2: $\mathcal{B}_d = \{n | l_h$ immediately contains $n\}$
3: $\mathcal{E}_d = \{(b_i, b_j) | b_i \in \mathcal{B}_d \wedge b_j \in \mathcal{B}_d \wedge (b_i, b_j) \in \mathcal{E}\}$
4: $n \leftarrow$ new virtual node (next)
5: $e \leftarrow$ new virtual node (exit)
6: $sn \leftarrow \{(b_i, n) | \exists b_j, (b_i, b_j)$ back-edge of $l_h\}$
7: $se \leftarrow \{(b_i, e) | \exists b_j, (b_i, b_j)$ exit-edge of $l_h\}$
8: $(v, i, o) \leftarrow (\emptyset, \emptyset, \emptyset)$
9: **for** each loop $l_{h'}$ directly in l_h **do**
10: $L_{h'} \leftarrow$ new hierarchical node
11: $i_{h'} \leftarrow \{(L_{h'}, b_j) | \exists b_j, (b_i, b_j)$ exit-edge of $l_{h'}\}$
12: $o_{h'} \leftarrow \{(b_i, L_{h'}) | \exists b_i, (b_i, b_j)$ entry-edge of $l_{h'}\}$
13: $(v, i, o) \leftarrow (v \cup \{L_{h'}\}, i \cup i'_h, o \cup o'_h)$
14: **end for**
15: $d \leftarrow < \mathcal{B}_d \cup v \cup \{n, e\}, \mathcal{E}_d \cup i \cup o \cup \{sn, se\} >$
16: **return** (d, n, e)
17: **end function**

4.2.2 Tree construction (Algorithm 2). First, we introduce the notion of *forced passage nodes*, upon which the recursive structure of our algorithm relies. Intuitively, these correspond to the set of nodes that appear in every path to the end node of a DAG.

Definition 6. *Let $\mathcal{D}$ a DAG. Let* start *the start node and* end *an exit node of $\mathcal{D}$. The set of* forced passage nodes *of $\mathcal{D}$ towards e, denoted $forced(D, e)$, is defined as:*

$$forced(D, e) = \{n \in \mathcal{D} | \text{start} \gg n \wedge n \gg \text{end}\} \setminus \text{start}$$

The function MakeCFT described by Algorithm 2 builds recursively a control-flow tree from a DAG. Notice that this function takes as arguments a start node and an end node. This is because in some cases it is useful to build the control-flow tree representing paths between two arbitrary nodes that are different from the entry and exit nodes of the DAG (see the different recursive calls in the algorithm for details).

Function MakeCFT returns a Seq node. The list of children for this Seq node is contained in variable ch. We will call this Seq node the *current* sequential node.

We denote as $\mathcal{N}$ the set of forced passage nodes towards end. In the while loop (lines 7 to 19), the algorithm goes through $\mathcal{N}$ in reverse dominance order (i.e from the end to the start). Since we must pass through all nodes in $\mathcal{N}$, it is clear that each node in $\mathcal{N}$ must be a leaf child of the current sequential node (line 18). As an example, consider the tree obtained from the example of Figure 2a, which is represented in Figure 2c. During each iteration of loop l_{b_1}, we are forced to pass through b_1 and b_3, so $\mathcal{N} = \{b_1, b_3\}$. Therefore, the control-flow tree has a Seq node as root, with children b_1 and b_3, as well as an Alt node whose construction is explained below.

If there exists multiple possible paths between two adjacent forced passage nodes (line 10) then an Alt node must be added to the ch list. We construct a tree for each possible predecessor by recursively calling MakeCFT, and the Alt node contains these trees as children (lines 13 to 15). In our example, the node b_3 has two predecessors, L_{b_2} and b_6. The control-flow trees corresponding to these two predecessors are respectively Leaf(L_{b_2}) and Leaf(b_6).

In lines 20 to 25, the algorithm deals with inner loops. Inner loops have previously been added to the ch list as hierarchical Leaf nodes. Here, they are replaced by control-flow trees representing these loops. Such a tree is composed of two parts, in sequence. The first part is the loop body (line 22), representing all the iterations of the loop. The second part is the loop exit ex (line 23), which represents the paths from the last execution of the loop header, to the loop exit.

For instance, in Figure 1b the sub-tree depicted in gray replaces the hierarchical node L_{b_2}. The left part of this sub-tree corresponds to the body of loop l_{b_2}, while the right part (below the dashed edge) corresponds to the exit of loop l_{b_2}.

We note that in the algorithm, sometimes a single basic block can be represented by several Leaf nodes. When such duplication occurs, we rename the duplicated basic block(s) such that each Leaf node has an unique label. This guarantees that two different paths in the tree are always identified by different sequences of Leaf nodes.

Algorithm 2 DAG to control-flow tree

```
1:  function MakeCFT(𝒟, start, end)
2:      ch ← ∅
3:      𝒩 ← forced(𝒟, end)
4:      if start has no predecessors then
5:          ch ← {Leaf(start)}
6:      end if
7:      while 𝒩 ≠ ∅ do
8:          Pick c from 𝒩 such that ∀n ∈ 𝒩, n ≫ c
9:          𝒩 ← 𝒩 \ c
10:         if c has at least 2 predecessors then
11:             br ← ∅
12:             ncd ← imm. dominator of c in 𝒩
13:             for p in predecessors(c) do
14:                 br ← br ∪ MakeCFT(𝒟, ncd, p)
15:             end for
16:             ch ← ch ∪ Alt(br)
17:         end if
18:         ch ← ch ∪ Leaf(c)
19:     end while
20:     for all Leaf(c) ∈ ch, c representing l_h do
21:         (𝒟', n, e) ← DAG(l_h)
22:         bd ← MakeCFT 𝒟', h, n)
23:         ex ← MakeCFT 𝒟', h, e)
24:         Replace Leaf(c) by Loop(h, bd, x_h, ex)
25:     end for
26:     return Seq(ch)
27: end function
```

4.3 Execution paths in CFG and Control-flow Tree

We will now establish a correspondence between CFG execution paths and tree execution paths. This subsection contains the general idea and definitions. For a complete proof, see Appendix A.

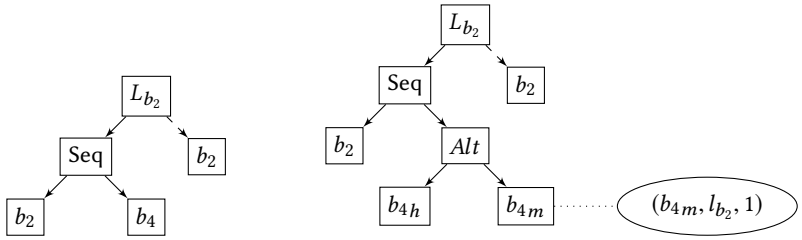
First, we denote gpaths(G, e) the function that, given a graph G and a node e, returns the set of execution paths $\{p_1, \ldots, p_k\}$ from the graph entry to the node e.

Second, a *tree execution path* is defined as a sequence of leaf nodes of the tree. We use the same notation for paths in the CFG and for paths in the tree, with the obvious correspondence between leaf nodes and basic blocks. The function tpaths(t) returns the set of tree execution paths of control-flow tree t. It is defined as follows:

Definition 7. *Let t be a Control-flow tree. The set of feasible execution paths of t, denoted* tpaths(t)*, is defined inductively as follows:*

$$
\begin{aligned}
\text{tpaths}(\mathit{Leaf}(b)) &= \{b\} \\
\text{tpaths}(\mathit{Seq}(t_1, \ldots, t_n)) &= \{p \mid \exists p_1 \in \text{tpaths}(t_1), \ldots, \exists p_n \in \text{tpaths}(t_n), p = p_1@\ldots@p_n\} \\
\text{tpaths}(\mathit{Loop}(h, t_b, n, t_e)) &= \{p \mid \exists p_1, \ldots, p_n \in \text{tpaths}(t_b), \exists p_e \in \text{tpaths}(t_e), p = p_1@\ldots@p_n@p_e\} \\
\text{tpaths}(\mathit{Alt}(t_1, \ldots, t_n)) &= \bigcup_{1 \le i \le n} \text{tpaths}(t_i)
\end{aligned}
$$

Let us denote $\mathcal{D}_s$ and $\mathcal{D}_e$ respectively the *start* and *exit* nodes of DAG D. Let G_e denote the exit node of G. The following theorem states the correctness of our translation from a CFG to a Control-flow Tree: any execution path in the CFG is also an execution path in the corresponding Control-flow Tree. However, some paths that are valid in the tree may not be valid in the CFG, therefore, the two representations are not equivalent. Still, this is *safe*, since the presence of additional paths in the CFT can only lead to an *over-approximation* of the WCET.



(a) Tree before annotation (b) Tree after annotation

Fig. 3. Context annotations

Theorem 4.1. *Let G be a CFG. Let $\mathcal{D} = DAG(G, \top)$ and let $t = \mathsf{MakeCFT}(\mathcal{D}, \mathcal{D}_s, \mathcal{D}_e)$. We have:*

$$\mathsf{gpaths}(G, G_e) \subseteq \mathsf{tpaths}(t)$$

Proof. See Appendix A for details. □

5 CONTEXT-SENSITIVE EXECUTION TIME

We now enrich the control-flow tree with *context annotations* designed to represent the result of extra-CFG analyses, that will help us reduce the pessimism in WCET estimation.

5.1 Context annotations

A context annotation constrains the conditions under which a sub-tree can be executed. In this work, annotations only represent constraints related to loops, which is usually the main source of WCET variability. Note that with IPET-based approaches, this information would be represented by an ILP constraint. We will detail the role of context annotations in parametric WCET in Section 6.

Definition 8. *A context annotation is a tuple (t, l, m), where t is a tree, l refers to an external loop (i.e., $l = Loop(h, t_b, x_h, t_e)$ is a loop such that t is contained within the loop body t_b), and m is the maximum number of times t can be executed each time l is entered. The null annotation is denoted by $(t, \top, \infty)$.*

Let $\mathsf{ann}(t)$ be the annotation on the root of tree t and let $\mathsf{ann}^*(t)$ be the set of annotations on all nodes in t (including root t).

We define $occ(\mathcal{P}, p)$ $occ : 2^{\mathcal{P}} \times \mathcal{P} \to \mathbb{N}$ as the function that returns the number of occurrences of any path in $\mathcal{P}$ inside path p.

Let t be a control-flow tree with context annotations. The previous definition of feasible execution paths is altered as follows:

$$\mathsf{tpaths}(Loop(h, t_1, n, t_2)) = \{p | \exists p_1, \ldots, p_n \in \mathsf{tpaths}(t_1), p_e \in \mathsf{tpaths}(t_2), p = p_1@\ldots@p_n@p_e,$$
$$\forall (t, l_h, m) \in \mathsf{ann}^*(t_1), occ(\mathsf{tpaths}(t), p) \leq m\}$$

We motivate the need to represent context-sensitive information by using two examples. First, 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 considered separately is 10, but the inner loop body can be executed at most $\sum_{i=1}^{10} i$ times. Knowing this information

will enable us to produce a tighter WCET estimation. To model this example, we have a Leaf(b) node representing the block inside the inner loop. This node has an annotation (Leaf(b), l_{outer}, 55) where l_{outer} represents the outer loop. This annotation represent the fact that, due to the triangular loop, the block b can be executed at most $\sum_{j=1}^{10} j = 55$ times in a complete execution of l_{outer}.

As a second example, 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). For instance, in the control-flow tree of Figure 3a, let us assume that the block corresponding to Leaf(b_4) is persistent. For every complete execution of loop l_{b_2}, b_4 can only cause a cache miss once. Thus the execution time of b_4 must account for the cache miss only once per complete execution of loop l_{b_2}. To model this example, we proceed in two steps. First, we modify the CFT by splitting the block b_4 from Figure 3a into two (virtual) leaves, representing respectively the cache hit and cache miss cases. This is shown in Figure 3b: Leaf(b_{4m}) corresponds to the miss and Leaf(b_{4h}) to the hit. Then, we add an annotation (b_{4m}, l_{b_2}, 1) to represent the fact that b_{4m} can be executed only once per execution of loop l_{b_2}.

Due to context annotations, some structurally feasible paths are now unfeasible. As an example, in the tree of Figure 3b, path $\{b_2, b_{4m}, b_2, b_{4m}, b_2\}$ is feasible if we ignore annotations. However, taking context annotations into account, this path it is not.

Context annotations are intended to be a generic tool to model various 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). Furthermore, as shown with the cache example above, it may be necessary to modify the CFT to represent some constraints. In the future, we might use other CFT transformations to represent other types of constraints (not necessarily only duplication).

5.2 Abstract WCET

Due to context annotations, the WCET of a segment of code that is executed iteratively can vary at each iteration. We introduce the concept of *abstract WCET* to represent the set of WCETs associated with a tree node. Abstract WCETs are defined using *multi-sets*, a generalization of sets where multiple instances of the same element are allowed. The number of instances of some element k in the multiset is denote $m(k)$ and called its *multiplicity*. In our context, we consider that the smallest element of the multiset has an implicit infinite multiplicity. We recall below some definitions on multi-sets:

Definition 9. *Let $\mathbb{N}^\#$ denote the set of multi-sets over $\mathbb{N}$. Let $\eta, \eta' \in \mathbb{N}^\#$ and let $n \in \mathbb{N}$. The following operations are defined on multi-sets:*

- *$\eta[n]$, denotes the $(n+1)$-th greatest element of η, i.e. $|\{k|k \in \eta, k > \eta[n]\}| \leq n < |\{k|k \in \eta, k > \eta[n]\}| + m(\eta[n])$. For instance, if $\eta = \{4, 3, 3\}$ then $\eta[0] = 4$, $\eta[1] = \eta[2] = 3$, ...;*
- *$\eta|_n$ denotes the multi-set that contains the n greatest elements of η (i.e. $\eta[0], \ldots, \eta[n-1]$) and an infinite number of zeros;*
- *$\eta \uplus \eta'$ is a modified version of the traditional multi-set sum, which we will denote $\uplus_{trad}$. Like $\uplus_{trad}$, $\uplus$ sums multiplicities. The difference is as follows. Let min_η, $min_{\eta'}$ denote respectively the smallest elements of η and η'. Then, we have: $\eta \uplus \eta' = \eta \uplus_{trad} \eta' \backslash \{k|k \leq max(min_\eta, min_{\eta'})$. So for instance, $\{8, 8, 4\} \uplus \{9, 8, 3, 2\} = \{9, 8, 8, 8, 4\}$;*
- *$\eta \otimes k$ denotes the multi-set for which each member has k times the multiplicity it has in η;*
- *$\eta'' = \eta \oplus \eta'$ is the multi-set such that: $\forall i \in \mathbb{N}$, $\eta''[i] = \eta[i] + \eta'[i]$.*

The notion of *abstract WCET* is now defined as follows:

Definition 10. *For any tree t, its abstract WCET is a pair $\alpha = (l, \eta)$, where l is a loop and η is a multi-set over $\mathbb{N}$. The presence of an integer n in η means that the code associated with t may have an execution time n, but only once, each time l is entered.*

For instance, in our cache example from Figure 3b, the abstract WCET computed for the Alt node would be $(l_{b_2}, \{\mathsf{time}(b_{4m}), \mathsf{time}(b_{4h}), \mathsf{time}(b_{4h}), \mathsf{time}(b_{4h}), \ldots\})$, meaning that the WCET of that node is $\mathsf{time}(b_{4m})$ for the first iteration of loop l_{b_2} and then it is $\mathsf{time}(b_{4h})$ for all subsequent iterations of the loop. Note that, if we exit and re-enter the loop, the WCET of the Alt node will again be $\mathsf{time}(b_{4m})$, then $\mathsf{time}(b_{4h})$, $\mathsf{time}(b_{4h})$, etc.

The abstract WCET for an expression $t \in \mathcal{T}$ is computed by applying the evaluation function $\gamma : \mathcal{T} \to L_G \times \mathbb{N}^{\#}$, defined below, using helper function $\omega(t)$:

$$\gamma(t) = (l, \eta) \qquad \text{where } l = l_1 \sqcap l_2, \eta = \eta_1|_n, (l_1, \eta_1) = \omega(t) \text{ and } (t, l_2, n) = \mathsf{ann}(t).$$

$\omega(t)$ computes the abstract WCET without considering the annotation on the root node of t, and then $\gamma(t)$ computes the abstract WCET resulting from the application of the annotation over t (if any). Notice that, if no annotation is defined over t, then $\mathsf{ann}(t) = (t, \top, \infty)$; as a consequence $l \sqcap \top = l$, $\eta|_\infty = \eta$, and $\gamma(t) = \omega(t)$.

We now define $\omega(t)$ for the different cases. First, when $t = \mathsf{Leaf}(b)$, the WCET of the basic block b is repeated an infinite number of times. In formula:

$$\omega(t) = (\top, \{\mathsf{time}(b)\} \otimes \infty)$$

The idea behind the processing of Alt nodes is based on the following observation: the worst-case scenario for multiple executions of the Alt node may involve execution of different children. Therefore, we need to merge the multi-sets resulting from $t_1, \ldots, t_n$. In formula, when $t = \mathsf{Alt}(t_1, \ldots, t_n)$:

$$\omega(t) = (l_1 \sqcap \cdots \sqcap l_n, \eta_1 \uplus \cdots \uplus \eta_n) \qquad \text{where } (l_1, \eta_1) = \gamma(t_1), \ldots, (l_n, \eta_n) = \gamma(t_n).$$

Example 5.1. Let us consider an Alt node with two children t_1 and t_2, such that $\gamma(t_1) = (l, \{5, 4, 2, 1\})$ and $\gamma(t_2) = (l, \{6, 2\})$. The first time the Alt node is executed, the WCET will be 6 (from t_2), the second time it will be 5 (from t_1), then 4, and so on. As such, we compute the abstract WCET for the Alt node by taking the union of the multi-set components of the two children abstract WCET. Therefore, in our example, $\omega(t) = (l, \{6, 5, 4, 2, 2\})$.

When $t = \mathsf{Seq}(t_1, \ldots, t_n)$, we make the following observation: for any n, the worst-case time for n executions of the Seq node is equal to the worst-case time for n executions of t_1 plus the worst-case time for n executions of t_2 and so on. In formula:

$$\omega(t) = (l_1 \sqcap \cdots \sqcap l_n, \eta_1 \oplus \cdots \oplus \eta_n) \qquad \text{where } (l_1, \eta_1) = \gamma(t_1), \ldots, (l_n, \eta_n) = \gamma(t_n).$$

Example 5.2. Let us consider a Seq node with two children t_1 and t_2, such that $\gamma(t_1) = (l, \{5, 4\})$ and $\gamma(t_2) = (l, \{2, 1\})$. The first time the Seq node is executed, its WCET will be $5+2 = 7$, the second time it will be $4+1 = 5$. As such, we compute the abstract WCET for the Seq node by adding elements of corresponding ranks. In the example, $\omega(t) = (l, \{7, 5\})$.

When $t = \mathsf{Loop}(h, t_1, x_h, t_2)$, let $(l_1, \eta_1) = \gamma(t_1)$ and $(l_2, \eta_2) = \gamma(t_2)$. Two different cases must be considered[1]. If l_h is the loop component of the abstract WCET of t_1 (case $l_1 \equiv l_h$), then the execution time of t_1 is a fixed value. In this case, the worst-case time for one execution of the Loop node is always the worst case execution time for x_h executions of the loop body t_1.

[1]Notice that, by definition of context annotation, it is not possible to have $l_h \equiv l_2$.

Otherwise, l_1 represents a loop that contains the currently processed Loop node. As in the previous case, the worst-case execution time for one execution of the Loop node is the worst-case execution time for x_h executions of the loop body t_1. However, since l_1 refers to an outer loop, successive executions of the Loop node yield different execution times, and these are summed together in groups of x_h elements.

To summarize, in formula:

$$\omega(t) = \begin{cases} (l_2, (\{\sum_{i=0}^{x_h-1} \eta_1[i]\} \otimes +\infty) \oplus \eta_2) & \text{if } l_h \equiv l_1 \\ (l_1 \sqcap l_2, \eta \oplus \eta_2) & \text{otherwise} \end{cases}$$

$$\text{where } (l_1, \eta_1) = \gamma(t_1) \text{ and } (l_2, \eta_2) = \gamma(t_2) \text{ and } \eta[i] = \sum_{j=i\cdot x_h}^{i\cdot x_h + x_h - 1} \eta_1[j].$$

Example 5.3. Let $\gamma(t_1) = (l_h, \{5, 4, 3\})$ (case $l_h \equiv l_1$), let the loop bound $x_h = 2$ and let t_2 be empty. Then the execution time for one execution of the loop is always $5 + 4 = 9$ (the sum of the x_h first ranks of the multi-set) and we have $\omega(t) = (\top, \{9\} \otimes \infty)$.

Example 5.4. Let $\gamma(t_1) = (l_1, \{5, 4, 3, 2\})$ (case $l_h \not\equiv l_1$), let the loop bound $x_h = 2$ and let t_2 be empty. Then the first execution of the loop will yield execution time $5 + 4 = 9$ (the sum of the first x_h ranks of the multi-set), while the second execution will yield execution time $3 + 2 = 5$ (the sum of the subsequent x_h ranks of the multi-set). Therefore, $\omega(t) = (l_1, \{9, 5\})$.

Notice that we make pessimistic simplifications concerning the loop component in the computation of ω and γ. Consider, the computation for $t = \mathsf{Alt}(t_1, \ldots, t_n)$ for instance. The WCET of $t_1, \ldots, t_n$ may depend on different loops, but keeping track of all these loops in the WCET of t would be very complex. So, as a simplification, we only keep track of the greatest lower bound of these loops (the loop that most immediately contains t). This is also true in other cases. However, this approximation is safe (see the proof of Theorem B for details) and has a low impact on WCET over-approximation (see Section 7).

5.3 From abstract to concrete WCET

We will now detail how to evaluate the WCET of a tree t inside a loop l. Suppose that t is executed n times and that its abstract WCET is $\gamma(t) = (l, \eta)$. The execution time for each individual execution of t depends on the number of times it was executed after the last time l was entered. Let e be the number of times l was entered, and let us assume that the n execution of t are distributed uniformly across all e executions of l (this is a realistic assumption because our computation method ensures that iterating every loop to the maximum results in the longest execution time).

Definition 11. *Let t be a control-flow tree and let $\gamma(t) = (l, \eta)$. The concrete WCET of t in the scenario where t is executed n times and the loop l is executed e times, where e and n are strictly positive and n is a multiple of e, is computed as: $\sum_{i=1}^{n} (\eta \otimes e)[i]$*

This definition applies to any node of the tree. To compute the WCET of a complete program represented by tree t, we apply the formula with $n = e = 1$, since we are only interested in one execution of the program. The WCET of the program is thus computed as $\sum_{i=1}^{1} (\eta \otimes 1)[i] = \eta[1]$. The following theorem establishes the soundness of our WCET evaluation method.

Theorem 5.5. *Let G a CFG. Let $\mathcal{D} = DAG(G, \top)$ and let $t = \mathsf{MakeCFT}(\mathcal{D}, \mathcal{D}_s, \mathcal{D}_e)$. Let $(l, \eta) = \gamma(t)$. We have:* $\forall p \in \mathsf{gpaths}(G, G_e), \mathsf{time}(p) \leq \eta[1]$

Proof. See Appendix B for details. □

6 SYMBOLIC COMPUTATION

In this section we study the problem of computing the abstract WCET of a tree when some parameters of the tree are unknown (loop bounds for instance, but not only). We show that, using simple syntactic sugaring, our definition of $\omega(t)$ produces formulae akin to arithmetic expressions. Then we rely on existing work on symbolic computation of arithmetic expressions to simplify abstract WCET formulae. The simplification step is mainly useful in case of on-line formula evaluation. It reduces memory overhead (since formulae must be part of the embedded code) as well as execution time overhead (since formulae must be evaluated at each task instanciation).

6.1 Abstract WCET formulae

First, we introduce several operators on abstract WCET, which act as syntactic sugar, to be able to express WCET computation as arithmetic computation.

Definition 12. *Let t_1 and t_2 be two control-flow trees. We define a set of operations on abstract WCET such that:*

$$\begin{aligned}
\omega(t_1) \oplus \omega(t_2) &= \omega(\mathit{Seq}(t_1, t_2)) \\
\omega(t_1) \uplus \omega(t_2) &= \omega(\mathit{Alt}(t_1, t_2)) \\
(\omega(t_1), \omega(t_2), h)^{x_h} &= \omega(\mathit{Loop}(h, t_1, x_h, t_2)) \\
\omega(t_1)_{\downarrow(h,n)} &= \gamma(t_1) \quad (\mathit{where}\ \mathsf{ann}(t_1) = (t_1, l_h, n)) \\
n \odot (l, \eta) &= (l, \eta') \quad (\mathit{where}\ \forall i, \eta'[i] = \eta[i] \times n) \\
k^{\infty} &= \{k\} \otimes \infty
\end{aligned}$$

Furthermore, we let $\theta \equiv (\top, 0^{\infty})$. We define the following grammar to represent the set of formulae $\mathcal{W}$ corresponding to the computation of the abstract WCET of a control-flow tree ($w \in \mathcal{W}$):

$$\begin{array}{lcl}
w & ::= & const \mid id \mid w_{\downarrow(h,it)} \mid w \oplus w \mid w \uplus w \mid (w, w, b)^{it} \\
h & ::= & b \mid id \\
it & ::= & i \mid id
\end{array}$$

The simplest formula is a constant abstract WCET value ($const \in (L_G \times N^{\#})$). A formula can also be a variable corresponding to an unknown WCET value (id). A formula can also be the sum ($w \oplus w$), the product ($w \uplus w$) or the repetition of two formulae ($(w, w, b)^{it}$). Finally, a formula can also consist of the application of an annotation to a formula ($w_{\downarrow(h,it)}$). The factor of a repetition and the factor of an annotation (it) can either be a constant integer value (i) or a variable (id). The loop header of an annotation (h) can either be a basic block name (b) or a variable (id).

6.2 Symbolic values

As we can see, several elements of these formulae can be symbolic values (denoted by id), i.e. variable parameters: symbolic WCET value (w), symbolic loop iteration bound (it), symbolic loop header (h). Let us now illustrate how these

symbolic values can be used to model various WCET variation sources. A simple example is the case where the number of iterations of a loop depends on an input of the system. The WCET of the loop is statically evaluated to $(\omega_1, \omega_2, h)^n$, where n is a symbolic value. The value of n is computed dynamically and the WCET of the loop is deduced from this value.

As a second example, we discuss how to perform a modular WCET analysis, in the case where the program contains a call to a dynamic library. Assume for instance that the library call is in the *else* branch of an $if - then - else$ and that the $then$ branch has a constant WCET of 5. The WCET is statically evaluated to $(\top, \{5\} \otimes \infty) \uplus \omega$, where ω is a symbolic value. We perform a separate analysis on the different programs the dynamic library call can correspond to, so we obtain a different WCET for each possibility. At program execution, we replace ω by the WCET corresponding to the library that is actually called and deduce the program WCET.

As a last example we discuss how to take into account the results of an instruction cache analysis. Let us consider the execution of a multi-task system with a non-preemptive scheduler. In such a system, though the hardware provides no means to consult the exact cache state, it can be approximated to an abstract cache state using the techniques of [2]. In some cases, the category of a block, that is to say whether the execution of the block will result in a miss or in a hit, depends on the content of the cache at the beginning of the execution of the task containing it. As a consequence, the block category cannot be determined statically, however it can be determined dynamically based on the abstract cache state at the beginning of the task execution. In Figure 3b, we have shown how to use context annotations to model a persistent block. Similarly, to model a block with a non-static category, we split the block into a *hit* and a *miss* alternative, and add annotations on both alternatives. So the WCET formula for this block will be: $(\omega_{hit})_{\downarrow h,n_1} \uplus (\omega_{miss})_{\downarrow h,n_2}$, where n_1 and n_2 are symbolic values. At the beginning of the task execution, we determine the values of n_1 and n_2 based on the abstract cache content and deduce the task WCET. A similar approach can be used to take into account data-cache analysis and branch prediction.

More generally, we believe that symbolic WCET evaluation is a powerful generic tool with many potential applications. The focus of this paper however, is to present the general framework. Potential applications will be the subject of future work.

Concerning the limitations of our approach, currently we cannot specify constraints relating different symbolic values, which may prevent some simplifications in WCET formula. For instance, a single parameter in the program external context (e.g. the data-cache size) may introduce several separate symbolic values in the WCET formula (e.g. the WCET of each basic-block whose WCET is impacted by the data-cache size will become a symbolic value). Handling such related symbolic values is clearly also an important topic for future work.

A second limitation is that some extra-CFG analyses information may be difficult to represent using context-annotations, such as for instance the results of CCG analysis [15].

6.3 Formula simplification

When variables appear in a WCET formula, we cannot reduce the formula to a constant abstract WCET value. However, in many cases the formula can be transformed into a simpler, yet equivalent formula. For instance, we have: $(x \oplus 2 \odot x) \oplus 3 \odot x \oplus y = 6 \odot x \oplus y$

Figure 4 lists all the rewriting rules we use in order to simplify WCET formulae. Most of them are direct transpositions of integer arithmetic simplification rules [9] to the case of WCET formulae. We make the following comments:

Associativity.

$$(w_1 \oplus w_2) \oplus w_3 \mapsto w_1 \oplus w_2 \oplus w_3 \quad (1)$$
$$w_1 \oplus (w_2 \oplus w_3) \mapsto w_1 \oplus w_2 \oplus w_3 \quad (2)$$
$$(w_1 \uplus w_2) \uplus_3 \mapsto w_1 \uplus w_2 \uplus w_3 \quad (3)$$
$$w_1 \uplus (w_2 \uplus w_3) \mapsto w_1 \uplus w_2 \uplus w_3 \quad (4)$$

Commutativity.

$$(w_1 \oplus w_2) \mapsto (w_2 \oplus w_1) \text{ if } w_2 \lhd w_1 \quad (5)$$
$$(w_1 \uplus w_2) \mapsto (w_2 \uplus w_1) \text{ if } w_2 \lhd w_1 \quad (6)$$

Distributivity.

$$(cst_1 \oplus w_3) \uplus (cst_2 \oplus w_3) \mapsto (cst_1 \uplus cst_2) \oplus w_3 \quad (7)$$

Neutral element.

$$w_1 \oplus \theta \mapsto w_1 \quad (8)$$
$$w_1 \uplus \theta \mapsto w_1 \quad (9)$$

Multiplication.

$$0 \odot w_1 \mapsto \theta \quad (10)$$
$$(k_i \odot w_1) \oplus w_1 \mapsto (k_i + 1) \odot w_1 \quad (11)$$

Annotation.

$$\theta_{\downarrow(h,it)} \mapsto \theta \quad (12)$$
$$w_1{\downarrow(h,it)} \oplus w_2{\downarrow(h,it)} \mapsto (w_1 \oplus w_2)_{\downarrow(h,it)} \quad (13)$$

Loop.

$$(w_1, w_2, b)^{it} \mapsto (w_1, \theta, b)^{it} \oplus w_2 \quad (14)$$

Fig. 4. Abstract WCET formula rewriting rules

- We rely on an order relation $\lhd$ on formulae, so as to ensure that the commutativity rules can only be applied in one direction for two given formulae. Classically, the order relation is defined based on the syntactic structure of the formulae (see e.g. [9] for details);
- Distributivity is applied in reverse order and only to factor constant terms;
- Concerning the annotation rewriting rule, the strategy consists in reducing the number of annotation applications;
- Concerning the loop rule, since we have no rule for combining loops, we only extract the loop exit tree from the loop;
- Combination of constant formulae is not detailed here but is applied as well. For instance, $(l, 2^\infty) \oplus (l, 3^\infty)$ is simplified to $(l, 5^\infty)$.

Let $\mathcal{R}$ denote the rewriting system consisting of all of these rewriting rules. Let w_1, w_2 two WCET formulae. We write $w_1 \mapsto_{\mathcal{R}} w_2$, or simply $w_1 \mapsto w_2$ when w_1 rewrites to w_2 using a single rule of $\mathcal{R}$. We write $w_1 \mapsto_* w_2$ when w_1 rewrites to w_2 using a sequence of rules of $\mathcal{R}$. Let ρ denote a *variable mapping*, that is to say a set of substitutions of the form $id \to v$ where id is an identifier and v is a value. Let $\rho(w)$ denote the result of the substitution of variables of w by their values in ρ. We assume that ρ maps identifiers to values of the correct type, meaning that it maps WCET identifiers to WCET values, loop identifiers to loop headers and integer identifier to integer values. We say that ρ is a *complete mapping* with respect to formula w when it maps all variables of w to a value.

Lemma 6.1. *Let $w_1, w_2 \in \mathcal{W}$. Let ρ a complete variable mapping of w_1. We have:*

$$w_1 \mapsto_* w_2 \Rightarrow \rho(w_1) = \rho(w_2)$$

Proof. We must prove that, for each rewriting rule, the formula on the left of the rule is equivalent to the formula on the right. Most rules are trivial to prove and rely on arithmetic properties on integer multi-sets. We only detail the proof for rules on annotations and loops.

Rule 13. Let $(l_1, \eta_1) = w_1$ and $(l_2, \eta_2) = w_2$.

$$\begin{aligned}(w_1 \oplus w_2)_{\downarrow(h,it)} &= (l_1 \sqcap l_2, (\eta_1 \oplus \eta_2)|_{it}) = (l_1 \sqcap l_2, \eta_1|_{it} \oplus \eta_2|_{it}) \\ &= (l_1, \eta_1|_{it}) \oplus (l_2, \eta_2|_{it}) = w_1{\downarrow(h,it)} \oplus w_2{\downarrow(h,it)}\end{aligned}$$

Rule 14. Let $(l_1, \eta_1) = w_1$ and $(l_2, \eta_2) = w_2$.

By definition of the ω function on Loop nodes, we see that the computation result for $(w_1, w_2, b)^{it}$ is of the form $(\langle loop \rangle, \langle expression \rangle \oplus \eta_2)$. Therefore, let us define (l, η) such that $(w_1, w_2, b)^{it} = (l, \eta \oplus \eta_2)$.

If $l_1 = b$ then:

$$(w_1, w_2, b)^{it} = (l_2, \eta \oplus \eta_2) = (\top, \eta \oplus 0^\infty) \oplus (l_2, \eta_2) = (w_1, \theta, b)^{it} \oplus w_2$$

If $l_1 \neq b$ then:

$$\begin{aligned}(w_1, w_2, b)^{it} &= (l_1 \sqcap l_2, \eta \oplus \eta_2) = (l_1 \sqcap l_2, \eta \oplus 0^\infty) \oplus (l_2, \eta_2) \\ &= (l_1, \eta \oplus 0^\infty) \oplus (l_2, \eta_2) = (w_1, \theta, b)^{it} \oplus w_2.\end{aligned}$$

This concludes the proof. □

The following Lemma states that recursive applications $\mathcal{R}$ to a given formula w eventually reach a fixed-point and always produce the same formula w'.

Lemma 6.2. *$\mathcal{R}$ is convergent.*

Proof. $\mathcal{R}$ is convergent if it *terminates* and it is *confluent*. The reader can refer to [1] for more detailed definitions and proof strategies that we use here.

Termination. We note that for each rule $l \mapsto r$ of $\mathcal{R}$, we have either of the following properties:

- Let $op(w)$ denote the sum of the number of operators $\oplus, \uplus, \odot, |$ in w. Then, we have $op(l) < op(r)$ (for the following rules: distributivity, neutral element, multiplication with an integer, annotation);
- The number of parenthesis is less in l than in r (for associativity rules);
- $l \lhd r$ (for commutativity rules);
- Let us extend op by defining $op((w_1, w_2, h)^k) = (k + 1) * (op(w_1) + op(w_2))$. Then $op(l) < op(r)$ (for loop rules).

Based on these properties, we can define a strict order relation $\prec$ on formulae such that, for each rule $l \mapsto r$ we have $l \prec r$. As a consequence $\mathcal{R}$ terminates.

Confluence. As $\mathcal{R}$ terminates, we only need to prove that its overlapping rules are locally confluent. Two rules $l_1 \mapsto r_1$ and $l_2 \mapsto r_2$ overlap if there exists a sub-term s_1 of l_1 (resp. s_2 of l_2) that is not a variable, and a unifier (a term substitution) u such that $u(s_1) = u(l_2)$ (resp. $u(s_2) = u(l_1)$). Unification is applied after renaming variables such that $Vars(l_1) \cap Vars(l_2) = \emptyset$. For instance, rules 1 and 2 overlap: we have two different possible sequences of re-writings for formula $(w_1 \oplus (w_2 \oplus w_3)) \oplus w_4$:

$$\begin{aligned}(w_1 \oplus (w_2 \oplus w_3)) \oplus w_4 &\mapsto (w_1 + w_2 + w_3) + w_4 \text{(rule 2)} \\ &\mapsto w_1 + w_2 + w_3 + w_4 \quad \text{(rule 1)}\end{aligned}$$

$$\begin{aligned}(w_1 \oplus (w_2 \oplus w_3)) \oplus w_4 &\mapsto w_1 + (w_2 + w_3) + w_4 \text{(rule 1)} \\ &\mapsto w_1 + w_2 + w_3 + w_4 \quad \text{(rule 2)}\end{aligned}$$

As both sequences produce the same formula, these overlapping rules are locally confluent.

We do not detail the proof for the remaining overlapping rules, since it is very similar to the case we just presented. We only list them below:

$(w_1 \uplus (w_2 \uplus w_3)) \uplus w_4$	(3 and 4)
$(w_1 \oplus w_2) \oplus w_3 \quad \text{if } w_2 \lhd w_1$	(1 and 5)
$w_1 \oplus (w_2 \oplus w_3) \quad \text{if } w_3 \lhd w_2$	(2 and 5)
$(w_1 \uplus w_2) \uplus w_3 \quad \text{if } w_2 \lhd w_1$	(3 and 6)
$w_1 \uplus (w_2 \uplus w_3) \quad \text{if } w_3 \lhd w_2$	(4 and 6)
$(cst_1 \oplus (w_3 \oplus w_4)) \uplus (cst_2 \oplus (w_3 \oplus w_4)$	(2 and 7)
$(cst_1 \oplus w_3) \uplus (cst_2 \oplus w_3) \quad \text{if } w_3 \lhd cst_1 \vee w3 \lhd cst_2$	(5 and 7)
$(w_1 \oplus \theta) \oplus w_2$	(1 and 8)
$w_1 \oplus (\theta \oplus w_2)$	(2 and 8)
$(w_1 \uplus \theta) \uplus w_2$	(3 and 9)
$w_1 \uplus (\theta \uplus w_2)$	(4 and 9)
$w_{1\downarrow(h,it)} \oplus w_{2\downarrow(h,it)} \quad \text{if } w_2 \lhd w_1$	(13 and 5)

This concludes the proof. □

To summarize, we enumerate below the steps of the computation of the WCET of a program with our approach. Steps 1 to 4 correspond to the computation of the parametric WCET formula. Steps 5 and 6 correspond to the computation of the actual WCET for some specific parameter values:

(1) Translate the program CFG to a CFT t;
(2) Add extra-CFG analyses results as context annotations;
(3) Compute $w = \gamma(t)$;
(4) Simplify w into w' using rewriting rules;
(5) Replace parameters by their values and obtain w'', with $w'' = (l, \eta)$;
(6) Return $\eta[1]$.

7 EXPERIMENTS

The benchmarks we selected for our experiments are summarized in Table 1. For each benchmark, we mention its source (ML for Mälardalen, TB for TACleBench, or PB for PapaBench), provide a short description of the kind of algorithm it performs and specify the function whose WCET is analyzed. We only introduce one parameter per benchmark because precision is independent of the number of parameters in our approach. The analyses have been executed on a PC with an Intel core i5 3470 at 3.2 Ghz, with 8 Gb of RAM. Every benchmark has been compiled with ARM crosstool-NG 1.20.0 (gcc version 4.9.1) with -O1 optimization level.

The results of our experiments are shown in Table 2. First, we detail the size of the WCET formulae computed by our approach. Column *CFG* shows the number of basic blocks in the CFG. Column *Initial* shows the size (the number of

Bench	*Source*	*Parameter*	*Algorithm*	*Function*
matmult	ML	Matrix size	Matrix multiplication	*Initialize (twice)*
cnt	ML	Matrix size	Matrix sum	*Sum*
fft	TB	Number of samples	FFT	*main*
compress	ML	Data size	Data compression	*main*
lift	TB	Number of sensors	Factory lift control	*main*
adpcm	ML	Trigo. computation steps	ADPCM encoding	*main*
aes_enc	TB	Data size	AES encryption	*main*
powerwindow	TB	Sensor data input size	Car window control	*main*
fbw	PB	Task activaction count	fly-by-wire	*main*
audiobeam	TB	Audio source count	Audio beamforming	*main*
mpeg2	TB	Video resolution	MPEG2 decoding	*main*

Table 1. Benchmarks summary

		Formula size		*Time (ms)*			*Pessimism (%)*			
Bench	*CFG*	*Initial*	*Final*	*Common*	*Us*	*ILP*	*Us*	*Min*	*Max*	*MPA*
matmult	111	130	5	1105	1	0	0.01	0.00	3.88	0.31
cnt	153	284	3	2278	2	8	0.15	0.00	3.59	30.4
fft	391	453	8	2968	4	16	0.00	0.00	1.51	-
compress	694	906	3	4760	11	40	0.02	0.01	0.03	-
lift	814	1799	5	5130	19	40	1.51	0.05	2.29	-
adpcm	2032	2211	3	10688	67	272	0.01	0.01	0.33	-
aes_enc	2205	2651	2	4914	30	260	0.04	0.03	0.04	-
powerwindow	3738	4453	24	45702	224	4192	0.01	0.01	1.43	-
fbw	10612	27251	2	36940	1198	8960	2.62	0.03	7.05	-
audiobeam	12299	47248	37	56566	1222	12824	0.12	0.00	0.49	-
mpeg2	38612	1658109	3	267332	12221	> 1 week	-	-	-	-

Table 2. Benchmarking results

operands) of the WCET formula before simplification, while Column *Final* shows the formula size after simplification. In most cases, the size of the non-simplified formula, which also corresponds to the size of the CFT, is close to the size of the CFG. Differences are due to the presence of structure-breaking instructions (such as *goto*, *break*, *continue*, *return* in the middle of a function), which force basic block aliasing in the CFG to CFT conversion algorithm. This is especially true for the *mpeg2*, and to a lesser extent for *lift*, *audiobeam*, and *fbw* benchmarks. For all benchmarks, the size of the simplified formula is very small and is related to the number of loops whose iteration count depends on the parameter.

Then, we compare our approach with an IPET approach. Comparison is performed according to two criteria: WCET analysis time, and pessimism of the resulting WCET. 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 [19] and the instruction cache is modeled using cache categorization [12]. The target instruction cache used in the analysis has 64 Kbytes, 16 ways, and blocks of 16 bytes. We chose a small cache to highlight the impact of the cache on the execution time for such small benchmarks. The instruction cache miss latency was assumed to be 10

cycles. Each benchmark is analyzed as a standalone task, 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 rely on OTAWA (version 1.0), an open source WCET computation tool [3]. These steps are common to the IPET approach and to our approach. For the remaining steps, in the case of the IPET approach, we use GNU lp_solve ILP solver [5]. Our approach was coded in Python, and executed with PyPy 2.4.0. We took the mean time for 1000 executions of our algorithm, to compensate for PyPy's slow start speed. To compare the WCET estimates, we instantiate our WCET formula by assigning to the parameter the constant value used in the IPET experiment.

The *Common* column represents the time spent by OTAWA for the preliminary steps (common to IPET and our approach), while the *Us* (our approach) and *ILP* columns correspond to the time spent for the remaining steps. The WCET evaluation time is essentially linear in the size of the CFT in our approach and noticeably lower than the evaluation time for the IPET approach. Notice that lp_solve did not find a solution for mpeg2 after one week of execution time. Furthermore, let us emphasize that computing the WCET for different parameter values with the IPET approach requires to run the whole analysis (*Common+ILP*) for each parameter value, while we only need to do the analysis (*Common+Us*) once and then instantiate the formula for each parameter value.

WCET pessimism is measured in comparison with the IPET result. The *Us* column represents the value of the pessimism with our approach for a fixed value of the parameter (the same value as the one used for the IPET approach). The *Min* and *Max* columns represent respectively the minimum pessimism and maximum pessimism (in percentage) for varying values of the parameter between 1 and 1000. We observed that, in general, the percentage of pessimism decreases with the value of the parameter, approximately with an hyperbolic shape. The pessimism of our approach is much lower than that of the MPA approach (results extracted from [7] are reported in column MPA). It is also extremely low compared to the IPET approach. Pessimism in our approach can be attributed to the following causes: (1) the reduced expressiveness of our CFT annotations (as opposed to ILP constraints) and (2) paths existing in the CFT but not in the CFG. Experiments show that the amount of pessimism does not depend on the size of the CFG.

8 CONCLUSION

In this paper we have presented a novel technique for parametric WCET analysis, which follows a completely new approach based on symbolic computation of WCET formulas. Experiments show very promising results: execution time is lower than the traditional non-parametric IPET technique and over-approximation of the WCET (compared to IPET) is extremely low.

Symbolic WCET computation has many advantages: it greatly reduces the time for analysing the parameters-space of system; it is modular and easily permits to separately analyse different modules of a system; it allows to efficiently compute the WCET on-line, thus paving the way to the use of our methodology in adaptive systems.

One of the main limitations of our method is that it is not possible to specify constraints relating different parameters, which may prevents some simplifications in the formulae. Furthermore, some constraints used in IPET (i.e. some types of unfeasible paths) cannot be easily represented with context annotations. We plan to extend context annotations in future works to solve these issues.

REFERENCES

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[2] 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'09)*, volume 10, pages 1–12, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] 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*, pages 35–46. Springer Berlin Heidelberg, Waidhofen/Ybbs, Austria, 2010.

[4] Bilel Benhamamouch, Bruno Monsuez, and Franck Védrine. Computing wcet using symbolic execution. In *Proceedings of the Second International Conference on Verification and Evaluation of Computer and Communication Systems*, VECoS'08, pages 128–139, Swinton, UK, 2008. British Computer Society.

[5] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system, May 1 2004.

[6] Armin Biere, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. The Auspicious Couple: Symbolic Execution and WCET Analysis. In *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30, pages 53–63, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[7] S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric wcet calculation. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA'09.*, pages 13–21, Beijing, China, Aug 2009. IEEE.

[8] Duc-Hiep Chu and Joxan Jaffar. Symbolic simulation on complicated loops for wcet path analysis. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 319–328, New York, NY, USA, 2011. ACM.

[9] J.S. Cohen. *Computer Algebra and Symbolic Computation: Mathematical Methods.* Number vol. 1 in Ak Peters Series. Peters, Natick, MA, USA, 2002.

[10] 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)*, pages 36:1–36:53, Washington, DC, USA, 2002. IEEE.

[11] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, 1988.

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

[13] Matthew S Hecht and Jeffrey D Ullman. Flow graph reducibility. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, Denver, CO, USA, 1972. ACM.

[14] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, November 1997.

[15] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 254–263, Washington, DC, USA, 1996. IEEE.

[16] Y-TS Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 298–307, Pisa, Italy, 1995. IEEE.

[17] 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. *IEEE Transactions on Software Engineering*, 21(7):593–604, 1995.

[18] 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 *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 232–242, San Antonio, TX, USA, Dec 2005. IEEE.

[19] Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. In Per Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers II*, volume 5470 of *Lecture Notes in Computer Science*, pages 222–241. Springer-Verlag, Berlin, Heidelberg, 2009.

[20] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. A new algorithm for identifying loops in decompilation. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, pages 170–183, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] Stephan Wilhelm and Björn Wachter. Symbolic state traversal for wcet analysis. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 137–146, New York, NY, USA, 2009. ACM.

[22] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *Network and Distributed System Security (NDSS), ISOC*, San Diego, CA, USA, 2015. Internet Society.

A CFG TO CFT

In this appendix, we prove the correctness of our translation from a CFG to a CFT. Namely, we prove that any valid path in the CFG is also a valid path in the CFT.

A.1 Execution paths in a hierarchical DAG

We have already defined the set of feasible execution paths for a CFG ($\text{gpaths}(G, e)$) and for a CFT ($\text{tpaths}(t)$). We will now define the function $\text{dpaths}(D)$ that returns the set of feasible paths of a *hierarchical* DAG $\mathcal{D}$. Since a DAG

is a particular case of graph, gpaths() can also be applied to a DAG, however, an important difference between both functions is that dpaths() explores recursively the sub-paths of hierarchical nodes appearing in the DAG.

Definition 13. *Let $\mathcal{D}$ be a DAG. The set of execution paths of $\mathcal{D}$ is defined as:*

$$\mathsf{dpaths}(\mathcal{D}, e) = \bigcup_{p \in \mathsf{gpaths}(\mathcal{D}, e)} \mathsf{spaths}(p)$$

where

$$\mathsf{spaths}(p.n) = \begin{cases} \{q = q_1@q_n | q_1 \in \mathsf{spaths}(p) \wedge q_n \in \mathsf{vpaths}(n)\} & \textit{(if } n \textit{ is hierarchical)} \\ \{q = q_1.n | q_1 \in \mathsf{spaths}(p)\} & \textit{(otherwise)} \end{cases}$$

$$\mathsf{spaths}(\epsilon) = \{\epsilon\}$$

and

$$\mathsf{vpaths}(L_h) = \{p = p_1@...@p_{x_h}@p_e | \forall i, 1 \leq i \leq x_h, p_i.h_{next} \in \mathsf{dpaths}(\mathcal{D}_h, h_{next}) \wedge p_e.h_{exit} \in \mathsf{dpaths}(\mathcal{D}_h, h_{exit})\}$$

where $\mathcal{D}_h, h_{next}$ and h_{exit} are respectively the DAG, the next node and the exit node corresponding to hierarchical node L_h.

A.2 Transformation correctness

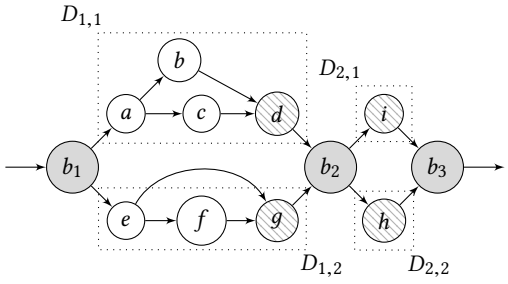


Fig. 5. Decomposing the DAG

We will proceed in two steps: first we will establish a correspondence between DAG execution paths and tree execution paths, then between CFG execution paths and DAG execution paths.

We will now present a graph decomposition technique on which our proof relies. Let $\mathcal{N} = \{b_1, \ldots, b_n\}$ denote the set of forced passage nodes of $\mathcal{D}$ towards $\mathcal{D}_e$. Then, $\mathcal{D}$ can be decomposed into a set of DAGs $\mathcal{D}_{i,j}$, where $j = 1, \ldots, i_k$ is the j-th predecessor of b_{i+1}. DAG $\mathcal{D}_{i,j}$ contains all nodes between b_i (excluded) and the j-th predecessor of b_{i+1} (included), and all related edges. If b_i is a hierarchical node, we denote Df_i the DAG representing the corresponding loop (if b_i is a basic block, Df_i is not defined).

Figure 5 shows such a decomposition. In this example, the forced passage nodes are shown in gray, and their predecessors are represented by a striped pattern. The DAG is decomposed into sub-DAGs $D_{1,1}$, $D_{1,2}$, $D_{2,1}$ and $D_{2,2}$ (plus a single node DAG for each forced passage node).

Lemma A.1. *Let $\mathcal{D}$ be a DAG. Let $t = \mathsf{MakeCFT}(\mathcal{D}, \mathcal{D}_s, \mathcal{D}_e)$. We have:*

$$\mathsf{dpaths}(D, \mathcal{D}_e) \subseteq \mathsf{tpaths}(t)$$

Proof. The proof is done by induction on the graph decomposition presented above. The base of the induction corresponds to the case where $\mathcal{D}$ consists only of a chain of forced passage basic blocks. Due to the definition of basic blocks though, this chain would always consist of a single basic block. Thus proving the induction base is trivial.

Let us now prove the induction step. Let $t_{i,j} = \mathsf{MakeCFT}(\mathcal{D}_{i,j}, \mathcal{D}_{i,j_s}, \mathcal{D}_{i,j_e})$, for any appropriate values of i and j. Let $tfb_i = \mathsf{MakeCFT}(\mathcal{D}f_i, \mathcal{D}f_{i_s}, \mathcal{D}f_{i_n})$, and let $tfe_i = \mathsf{MakeCFT}(\mathcal{D}f_i, \mathcal{D}f_{i_s}, \mathcal{D}f_{i_e})$.

We must now prove the induction step: assuming Inclusions 15, 16, 17, prove Inclusion 18.

$$\forall i, j, \mathsf{dpaths}(\mathcal{D}_{i,j}, \mathcal{D}_{i,j_e}) \subseteq \mathsf{tpaths}(t_{i,j}) \tag{15}$$

$$\forall i, \mathsf{dpaths}(\mathcal{D}f_i, \mathcal{D}f_{i_n}) \subseteq \mathsf{tpaths}(tfb_i) \tag{16}$$

$$\forall i, \mathsf{dpaths}(\mathcal{D}f_i, \mathcal{D}f_{i_e}) \subseteq \mathsf{tpaths}(tfe_i) \tag{17}$$

$$\mathsf{dpaths}(\mathcal{D}, \mathcal{D}_e) \subseteq \mathsf{tpaths}(t) \tag{18}$$

To simplify the notation, we will assume that each time a variable named i is introduced in some equation in the proof, it is constrained to $1, \ldots, n$. Similarly, when j is introduced, it is constrained to $1, \ldots, i_k$.

For any path p in $\mathsf{dpaths}(\mathcal{D}, \mathcal{D}_e)$, p can be expressed as $p = pf_1@p_1@pf_2@\ldots@p_{n-1}@pf_n$, where the pf_i terms are the path segments corresponding to the execution of forced passage nodes, and p_i terms are the path segments corresponding to the execution between these forced passage nodes.

For all i, if b_i is a basic block, then let $tf_i = \mathsf{Leaf}(b_i)$. Otherwise, let $tf_i = \mathit{Loop}(tfb_i, tfe_i)$.

Let us show that $\forall i$, $pf_i \in \mathsf{tpaths}(tf_i)$. If b_i is a basic block, then we have $pf_i = \{b_i\} \in \mathsf{tpaths}(tf_i)$. If b_i is a hierarchical node, then we have $pf_i \in \mathsf{vpaths}(b_i)$. Thanks to induction hypothesis, $\mathsf{dpaths}(\mathcal{D}f_i, \mathcal{D}f_{i_n}) \subseteq \mathsf{tpaths}(tfb_i)$ and $\mathsf{dpaths}(\mathcal{D}f_i, \mathcal{D}f_{i_e}) \subseteq \mathsf{tpaths}(tfe_i)$. Due to the definition of $\mathsf{vpaths}(b_i)$, $pf_i \in \mathsf{tpaths}(tf_i)$.

We have $\forall i, \exists j, p_i \in \mathsf{dpaths}(\mathcal{D}_{i,j}, b_{i+1})$. Thus, thanks to the induction hypothesis, $\forall i, \exists j, p_i \in \mathsf{tpaths}(t_{i,j})$. Thanks to the definition of the function tpaths() on the Alt node, we have $\forall i, p_i \in \mathsf{tpaths}(\mathsf{Alt}(t_{i,1}, \ldots, t_{i,i_k}))$.

As a consequence, we have $p \in \mathsf{tpaths}(\mathsf{Seq}(tf_1, \mathsf{Alt}(t_{1,1}, \ldots, t_{1_k}), \ldots, \mathsf{Alt}(t_{n-1,1}, \ldots, t_{n-1,k}), tf_n))$.

Now, we must prove that this corresponds to the structure of the tree built by our algorithm. By examining the algorithm, we see that t is a Seq node, whose children list alternates between Leaf nodes representing the forced passage nodes, and Alt nodes (line 16) corresponding to possible paths between forced passage nodes.

The tree representing the forced passage node b_i is either $\mathsf{Leaf}(b_i)$, if b_i is a basic block (line 18), or $\mathsf{Loop}(tfb_i, tfe_i)$, otherwise (line 21-24). The definition of this tree is thus that of tf_i.

Furthermore, each child tree of one of the Alt nodes represents the paths between a *forced passage* node, and a predecessor of the next *forced passage* node (the test at line 4 prevents the double counting of the *forced passage* nodes).

Therefore, we have $t = \mathsf{Seq}(tf_1, \mathsf{Alt}(t_{1,1}, \ldots, t_{1_k}), \ldots, \mathsf{Alt}(t_{n-1,1}, \ldots, t_{n-1,k}), tf_n)$. As a consequence, $p \in \mathsf{tpaths}(t)$ and finally $\mathsf{dpaths}(\mathcal{D}, \mathcal{D}_e) \subseteq \mathsf{tpaths}(t)$. □

Now we can proceed to the final correctness theorem.

Theorem A.2. *Let G be a CFG and let G_e denote the exit node of G. Let $\mathcal{D} = DAG(G, \top)$ and let $t = \mathsf{MakeCFT}(\mathcal{D}, \mathcal{D}_s, \mathcal{D}_e)$. We have:*

$$\mathsf{gpaths}(G, G_e) \subseteq \mathsf{tpaths}(t)$$

Proof. Let $D = DAG(G, \top)$. All we need to prove now is that $\mathsf{gpaths}(G) \subseteq \mathsf{dpaths}(\mathcal{D}, \mathcal{D}_e)$. The problem of reducing the CFG into a hierarchy of DAGs is a classical problem in compiler theory. Our method is similar to the one described in [13], so we take its correctness for granted. □

B WCET CORRECTNESS

In this appendix, we show that the WCET obtained with our approach is greater than the execution time of any feasible path in the CFT. Since we also proved that any paths of the CFG is also a path of the CFT obtained by our translation, these two properties ensure that the WCET computed by our approach is greater than the execution time of any feasible path in the CFG, which establishes the correctness of our approach.

Let $\mathsf{eval}(\eta, e, n) \equiv \sum_{i=1}^{n} (\eta \otimes e)[i]$. We want to prove that the WCET estimation for the program, provided by function eval, is an upper bound on the execution time of any path in the tree t. The proof strategy is the following:

- We first define a property of the abstract WCET on a control-flow tree. The property is verified only if the abstract WCET is a valid representation of the tree's many possible execution times;
- We then show that our function γ provides an abstract WCET which verifies the property mentioned above;
- Finally, we show that this property implies that the WCET estimation for the program is an actual upper bound.

We start by introducing an helper function prep (for *path repetition*). It is a generalization of tpaths() that computes all the paths in n repetitions of t, considering that an external loop l of t has been entered e times:

Definition 14. *Let* $\mathsf{prep}(t, e, n)$ *be defined as follows:*

$$\begin{aligned}\mathsf{prep}(t, e, n) = \{p | \exists p_1, \ldots, p_n \in \mathsf{tpaths}(t), p = p_1 @ \ldots @ p_n,\\ \forall (t', l, m) \in \mathsf{ann}^*(t), l \notin t \implies occ(\mathsf{tpaths}(t'), p) \leq e \cdot m\}\end{aligned}$$

If t is the whole program, then $\mathsf{prep}(t, 1, 1) = \mathsf{tpaths}(t)$ (in that case, there is no loop containing t, so all annotations in $\mathsf{ann}^*(t)$ refer to loops inside t).

We are now ready to state our predicate.

Definition 15. *$V(t, \eta)$ is a predicate representing the fact that η is a valid abstract WCETs for control-flow tree t:*

$$V(t, \eta) \equiv \forall e, n \in \mathbb{N}, p \in \mathsf{prep}(t, e, n), \mathsf{time}(p) \leq \mathsf{eval}(\eta, e, n)$$

This property is actually a generalization of the property we want to prove, i.e. that $\mathsf{eval}(\eta, 1, 1) = \eta[1]$ is a correct upper bound for any possible execution of a tree t.

Then, the following theorem states that the function γ computes an abstract WCET that satisfies the property V.

Theorem B.1. $\forall t \in \mathcal{T}, (l, \eta) = \gamma(t) \implies V(t, \gamma(t))$.

First, we state a property on γ that will be useful later during the proof.

Lemma B.2. *Let $\gamma(t) = (l, \eta)$. Then:*

$$\forall (t', l', m) \in \mathsf{ann}^*(t), l' \notin t \implies l \sqsubseteq l'.$$

Proof. By definition of γ and ω, l is always computed as the intersection between external loops. So, it can never happen that l refers to a loop that is more external than a loop contained within an annotation in t. □

We prove the theorem by induction on the structure of the control-flow tree. We start by proving that, if the property is valid for the result of ω, then it is also valid for the result of γ.

Lemma B.3. *Let t be a control-flow tree, and let $\mathsf{ann}(t) = (t, l_1, k)$ be its annotation. Let t' be the same tree on which the annotation on t has been replaced by the empty annotation $(t', \top, \infty)$. Let $\omega(t') = (l', \eta')$ and let $\gamma(t) = (l, \eta)$. Then:*

$$V(t', \eta') \implies V(t, \eta)$$

Proof. Clearly, $\gamma(t') = \omega(t') = \omega(t)$ because function ω does not consider the annotation on the root of t.
For all $e, n \in \mathbb{N}$, let $M = \max(e \cdot k, n)$.

(1) by definition, $\mathsf{prep}(t, e, n) = \mathsf{prep}(t', e, M)$;
(2) by definition, $\mathsf{eval}(\eta, e, n) = \mathsf{eval}(\eta', e, M)$.

From item 1, it follows that $\forall p \in \mathsf{prep}(t, e, n)$ we have also that $p \in \mathsf{prep}(t', e, M)$.
From $V(t', \eta')$, it follows that $\mathsf{time}(p) \leq \mathsf{eval}(\eta', e, M)$. From item 2, $\mathsf{eval}(\eta', e, M) = \mathsf{eval}(\eta, e, n)$ which proves the lemma. □

To prove Theorem B.1, we consider each case of the inductive definition of the CFT separately (Seq, Alt, Loop).

Lemma B.4. *Let $t = Seq(t_1, t_2)$, and let $(l, \eta) = \gamma(t)$, $(l_1, \eta_1) = \gamma(t_1)$, and $(l_2, \eta_2) = \gamma(t_2)$. Then,*

$$\forall e, n \in \mathbb{N}, \quad V(t_1, \eta_1) \wedge V(t_2, \eta_2) \implies V(t, \eta)$$

Proof. Let t' be the same tree as t but without the annotation on the root node and let $(l', \eta') = \omega(t')$.
By definition of function eval, we have:

$$\begin{aligned}\mathsf{eval}(\eta', e, n) &= \sum_{i=0}^{n-1} (\eta' \otimes e)[i] = \sum_{i=0}^{n-1} ((\eta_1 \oplus \eta_2) \otimes e)[i] = \sum_{i=0}^{n-1} (\eta_1 \otimes e)[i] + \sum_{i=0}^{n-1} (\eta_2 \otimes e)[i] \\ &= \mathsf{eval}(\eta_1, e, n) + \mathsf{eval}(\eta_2, e, n).\end{aligned}$$

By definition of predicate V:

$$\forall p_1 \in \mathsf{prep}(t_1, e, n),\ \mathsf{time}(p_1) \leq \mathsf{eval}(t_1, e, n)$$

$$\forall p_2 \in \mathsf{prep}(t_2, e, n),\ \mathsf{time}(p_2) \leq \mathsf{eval}(t_2, e, n)$$

Any path $p \in \mathsf{prep}(t, e, n)$ is a permutation of some $p_1 @ p_2$, hence

$$\mathsf{time}(p) \leq \mathsf{eval}(\eta_1, e, n) + \mathsf{eval}(\eta_2, e, n) = \mathsf{eval}(\eta', e, n)$$

and this proves that $V(t', \eta')$ holds. From Lemma B.3, it follows that $V(t, \eta)$ also holds. □

Lemma B.5. *Let $t = Alt(t_1, t_2)$, $(l, \eta) = \gamma(t)$, and $(l_1, \eta_1) = \gamma(t_1)$, and $(l_2, \eta_2) = \gamma(t_2)$. Then,*

$$\forall e, n \in \mathbb{N}, \quad V(t_1, \eta_1) \wedge V(t_2, \eta_2) \implies V(t, \eta)$$

Proof. Let t' be the same tree as t but without the annotation on the root node and let $(l', \eta') = \omega(t')$. By definition of functions ω and γ, $\eta' = \eta_1 \uplus \eta_2$. It follows that

$$\eta' \otimes e = (\eta_1 \uplus \eta_2) \otimes e = (\eta_1 \otimes e) \uplus (\eta_2 \otimes e).$$

From the n greatest elements of $\eta' \otimes e$, we have x elements coming from $\eta_1 \otimes e$, and y elements coming from $\eta_2 \otimes e$. We note that we can have several valid values of x and y if there are shared time values between η_1 and η_2.

By definition, we have:

$$\text{eval}(\eta', e, n) = \sum_{i=0}^{n-1} (\eta' \otimes e)[i] = \sum_{i=0}^{n-1} ((\eta_1 \otimes e) \uplus (\eta_2 \otimes e))[i] \geq \sum_{i=0}^{x-1} (\eta_1 \otimes e)[i] + \sum_{i=0}^{y-1} (\eta_2 \otimes e)[i]$$

The last inequality is true for any choice of x and y such that $x + y = n$, because we pick the x greatest elements from $\eta_1 \otimes e$ and the y greatest elements from $\eta_2 \otimes e$. Moreover, the sum of the n greatest elements of $\eta' \otimes e$ is never inferior to the sum of the x greatest elements of $\eta_1 \otimes e$ and the y greatest elements of $\eta_2 \otimes e$.

Now, let p_{max} be the worst-case path of $\text{prep}(t', e, n)$. Because of the definition of prep on Alt nodes, we can find x and y such that $p_1 \in \text{prep}(t_1, e, x)$ and $p_2 \in \text{prep}(t_2, e, y)$, and such that p_{max} is a permutation of nodes from p_1 and p_2. Obviously, we have $\text{time}(p_{max}) = \text{time}(p_1) + \text{time}(p_2)$. Because of the induction hypothesis, we have $\text{time}(p_1) \leq \text{eval}(\eta_1, e, x)$ and $\text{time}(p_2) \leq \text{eval}(\eta_2, e, y)$. Therefore $\text{time}(p_{max}) \leq \text{eval}(\eta', e, n)$, and this proves that $V(t', \eta')$ holds. From Lemma B.3, it follows that $V(t, \eta)$ also holds. □

Lemma B.6. *Let $t = \mathit{Loop}(h, t_b, x_h, t_e)$, $(l, \eta) = \gamma(t)$, $(l_b, \eta_b) = \gamma(t_b)$, $(l_e, \eta_e) = \gamma(t_e)$. Then:*

$$\forall e, n \in \mathbb{N}, \quad V(t_b, \eta_b) \wedge V(t_e, \eta_e) \implies V(t, \eta)$$

Proof. Let t' be the same tree as t but without the annotation on the root node and let $(l', \eta') = \omega(t')$.

If $l_b = l_h$, from the definition of ω, it follows that the estimated time for one full execution of loop l is constant. Let us name this constant $c = \text{eval}(\eta_b, 1, x_h) = \sum_{i=0}^{x_h - 1} \eta_b[i]$. By definition of γ and ω:

$$\text{eval}(\eta', e, n) = cn + \text{eval}(\eta_e, e, n).$$

For all $p \in \text{prep}(t', e, n)$, $\exists p_1, \ldots, p_n \in \text{prep}(t_b, 1, x_h)$ and $\exists p_e \in \text{prep}(t_e, e, n)$, such that p is a permutation of $p_1@\ldots@p_n@p_e$. We have $\text{time}(p) = \text{time}(p_1) + \ldots + \text{time}(p_n) + \text{time}(p_e)$. Also, $\forall k, \text{time}(p_k) \leq \text{eval}(\eta_b, 1, x_h)$. Therefore,

$$\text{time}(p) \leq n \cdot \text{eval}(\eta_b, 1, x_h) + \text{eval}(\eta_e, e, n) = \text{eval}(\eta', e, n).$$

Notice that we can rule out case $l_e = l_h$, by definition of context annotations.

If $l_b \neq l_h$, then by definition of γ and ω functions, we have

$$\eta'[i] = \sum_{j=i \cdot x_h}^{i \cdot x_h + x_h - 1} \eta_1[j].$$

We know that n is a multiple of e. Let $n = k \cdot e$. We have:

$$\begin{aligned}
\text{eval}(\eta', e, n) &= \text{eval}(\eta', 1, k) \cdot e \\
&= e \cdot \sum_{i=0}^{k-1} \sum_{j=i \cdot x_h}^{i \cdot x_h + x_h - 1} (\eta_b)[j] + \text{eval}(\eta_e, e, n) = e \cdot \sum_{i=0}^{k \cdot x_h - 1} \eta_b[i] + \text{eval}(\eta_e, e, n) \\
&= e \cdot \text{eval}(\eta_b, 1, k \cdot x_h) + \text{eval}(\eta_e, e, n) = \text{eval}(\eta_b, e, n \cdot x_h) + \text{eval}(\eta_e, e, n).
\end{aligned}$$

Since $l_b \neq l_h$, and from Lemma B.2, we know that no annotation in t refers to the current loop. Therefore, for all $p \in \text{prep}(t', e, n)$, p can be expressed as the permutation of $p_b@p_e$, where paths $p_b \in \text{prep}(t_b, e, n \cdot x_h)$ and

$p_e \in \mathsf{prep}(t_e, e, n)$. Then:

$$\mathsf{time}(p) = \mathsf{time}(p_b) + \mathsf{time}(p_e) \leq \mathsf{eval}(\eta_b, e, n \cdot x_h) + \mathsf{eval}(\eta_e, e, n) = \mathsf{eval}(\eta', e, n).$$

This proves that $V(t', \eta')$ holds. From Lemma B.3, it follows that $V(t, \eta)$ also holds. □

We can now conclude on the validity of our complete WCET evaluation method.

Theorem B.7. *Let G a CFG. Let $\mathcal{D} = DAG(G, \top)$ and let $t = \mathsf{MakeCFT}(\mathcal{D}, \mathcal{D}_s, \mathcal{D}_e)$. Let $(l, \eta) = \gamma(t)$. We have:* $\forall p \in \mathsf{gpaths}(G, G_e), \mathsf{time}(p) \leq \mathsf{eval}(\eta, 1, 1)$

Proof. Consequence of Theorem A.2 and Theorem B.1. □