Segment Abstraction for Worst-Case Execution Time Analysis*

Pavol Černý[1], Thomas A. Henzinger[2], Laura Kovács[3], Arjun Radhakrishna[4], and Jakob Zwirchmayr[5]

[1]University of Colorado Boulder [2]IST Austria
[3]Chalmers University of Technology [4]University of Pennsylvania
[5]Institut de Recherche en Informatique de Toulouse

Abstract. In the standard framework for worst-case execution time (WCET) analysis of programs, the main data structure is a single instance of integer linear programming (ILP) that represents the whole program. The instance of this NP-hard problem must be solved to find an estimate for WCET, and it must be refined if the estimate is not tight. We propose a new framework for WCET analysis, based on abstract segment trees (ASTs) as the main data structure. The ASTs have two advantages. First, they allow computing WCET by solving a number of independent small ILP instances. Second, ASTs store more expressive constraints, thus enabling a more efficient and precise refinement procedure. In order to realize our framework algorithmically, we develop an algorithm for WCET estimation on ASTs, and we develop an interpolation-based counterexample-guided refinement scheme for ASTs. Furthermore, we extend our framework to obtain parametric estimates of WCET. We experimentally evaluate our approach on a set of examples from WCET benchmark suites and linear-algebra packages. We show that our analysis, with comparable effort, provides WCET estimates that in many cases significantly improve those computed by existing tools.

1 Introduction

Worst-case execution time (WCET) analysis [18] is important in many classes of applications. For instance, real-time embedded systems have to react within a fixed amount of time. For another example, consider computer algebra libraries that provide different implementations for the most heavily-used methods. Users have to choose the most suitable method for their particular system architecture. In both cases, a tool that soundly and tightly approximates the WCET of a program on a given architecture would thus be very helpful.

State of the art. Most state of the art WCET estimation tools proceed in three phases (see for instance the survey [18]):

* This research was supported in part by the European Research Council (ERC) under grant agreement 267989 (QUAREM), the Austrian National Research Network RiSE (FWF grants S11402-N23), the Swedish VR grant D0497701, the WWTF PROSEED grant ICT C-050, the French National Research Agency grant W-SEPT (ANR-12-INSE-0001), NSF Expeditions award CCF 1138996, DARPA under agreement FA8750-14-2-0263, and a gift from the Intel Corporation.

– *First phase: Architecture-independent flow analysis,* which computes invariants, loop bounds, and finds correlations between the number of times different basic blocks in the program are executed. Such facts are called flow facts in the WCET literature, and the analysis is called flow analysis.
– *Second phase: Architecture-dependent low-level analysis*, which finds WCET for each basic block, using a model of a particular architecture, and abstract interpretation over domains that model for instance caches and pipelines.
– *Third phase: Path analysis,* which combines the results of the previous two phases. The commonly-used algorithm is called Implicit Path Enumeration Technique (IPET) [15]. It constructs an Integer Linear Programming (ILP) problem using WCET estimates for each basic block and constraints arising from flow facts to rule out some infeasible paths.

Recently, several approaches to refining WCET estimates were proposed. These works [14, 3] use an approach named "WCET squeezing" in [14], which adds constraints to the ILP problem arising from IPET.

The main data structure used is a single ILP problem for the whole program. We make two observations about the standard approach: first, flow facts (gathered in the first phase, or obtained by refinement) lead to *global* constraints in the ILP constructed in the third phase. Hence, the ILP problem (an instance of an NP-hard problem) cannot be decomposed into smaller problems. Second, the current approaches to refinement add constraints to the ILP; and in this way eliminate only one path at a time.

Our thesis. The main thesis of this paper is that hierarchical segment abstraction [8] is the right framework for WCET analysis. Segments are sequences of program instructions. Segment abstractions are those where an abstract state represents a set of segments, rather than set of concrete states. We represent hierarchical segment abstraction in a data structure called abstract segment trees (ASTs). The concept of segment abstraction [8] and its quantitative version [6] were introduced only very recently. We believe that WCET analysis is a prime application for segment abstraction.

We give two main arguments in support of the thesis. First, hierarchical segment abstraction allows us to compute the WCET by solving a number of independent ILP problems, instead of one large global ILP problem. This is because ASTs allow storing constraints *locally*. Second, hierarchical segment abstraction enables us to develop a more precise and efficient refinement procedure. This is because ASTs store more *expressive* constraints than ILP.

Algorithm. In order to substantiate our thesis, we develop an algorithm for producing increasingly tight WCET estimates. There are three key ingredients to the algorithm. First, we define an abstraction of programs that contains quantitative information necessary to estimate WCET. Second, we develop an algorithm for WCET estimation on ASTs. Third, we develop counterexample-guided refinement for segment abstraction based on interpolation.

Abstractions for WCET: abstract segment trees. The main idea of hierarchical segment abstraction is that an abstract state corresponds to a set of concrete segments, rather than to a set of concrete states, as in state-based ab-

stractions. Reasoning about segments is suitable for WCET estimation, where the analysis needs, for example, to distinguish execution times for different paths through a loop body, and store the relative number of times these paths are taken. Consider a simple example of a loop through which there are two paths, P_1 and P_2. Let us assume that the path P_1 takes a long time to execute, but is taken only once every 4 iterations of the loop; otherwise, a cheaper path P_2 is taken. To obtain a precise estimate of WCET, we thus need to store two types of numerical facts: the first one is the current estimates of execution time of P_1 and P_2, and the second fact is that an iteration taking path P_1 is followed by 3 iterations taking path P_2. We show that both these facts about paths can be stored *locally* if the basic object in the representation is a set of segments. To contrast with the standard approach to WCET analysis, note that the second quantitative fact is stored as a global constraint in the ILP.

The abstraction is hierarchical in order to capture the hierarchical nature of traces of structured programs with loops and procedures. For example, we split the set of traces through a nested loop into repeated iterations of the outer loop, and each outer loop iteration is split into repeated inner loop iterations. The hierarchical abstraction is represented by an *abstract segment tree*.

Each node of the abstract segment tree represents a set of segments. The nodes in our abstract segment trees contain quantitative information so that the WCET of the program can be estimated using only the abstraction.

Evaluation of WCET on abstractions. After constructing the abstraction of the program (an AST), the next step is to evaluate the WCET on the AST. The AST is a hierarchical structure. Each node of the AST gives rise to a problem that can be solved using an ILP encoding. The key difference to the standard IPET approach is that IPET constructs one global ILP problem for the whole program, and the ILP cannot be decomposed due to global constraints. segment abstractions and the hierarchical nature of ASTs enable us to decompose the solving into smaller problems, with one such problem for each node.

We propose a new encoding of the constraint-solving problem that arises at each node into ILP. The problem at each node could be reduced to ILP by a technique presented in [16], but this would lead to a possibly exponential number of constraints, whereas our encoding produces linear number of constraints.

Abstraction-refinement for WCET. After evaluating the WCET on ASTs, we obtain a *witness* trace, that is, a trace through the AST that achieves the WCET. It might be that the trace is feasible in the current abstraction but is infeasible in the program. Hence, we need a refinement as the next phase, in order to obtain a more precise WCET estimate. The algorithm refines the abstraction based on the current WCET estimate and the corresponding worst-case path.

We use the classical abstraction-refinement loop approach, adapted to ASTs. If the witness trace is feasible in the original program, the current WCET estimate is tight and we report it. Otherwise, we refine the abstraction using a novel interpolation-based approach for refinement of segment predicates. The refinements are monotonic w.r.t. WCET estimates, i.e., they are monotonically decreasing. Having expressive constraints (relational predicates) stored in ASTs

allows us to perform more efficient and more precise refinement than state-of-the-art WCET refinement techniques that add constraints to the ILP problem. Our refinement is more efficient, as we can potentially eliminate many traces at a time, and it is more precise, as the constraints that determine how the set represented by abstract nodes can be combined are more expressive than the constraints in ILP.

Parameters. Furthermore, we extend our framework to provide parametric estimates of WCET, following [4, 1]. In many cases, a single number as a WCET estimate is a pessimistic over-estimate. For instance, for a program that transposes a $n \times n$ matrix, a single numeric WCET is the WCET for the largest possible value of n. We adapt our evaluation algorithm to compute parametric WCET estimates as disjunctive linear-arithmetic expressions.

Experimental Evaluation. Our goal was to evaluate the idea of using ASTs as the basic data structure for WCET estimation. We built a tool IBART for computing (parametric) WCET estimates for C programs. For obtaining WCET estimates for basic blocks of programs, we used two low-level analyzers from existing frameworks, r-TuBound [13] and OTAWA [2]. The low-level analyzers CalcWCET167 and owcet included in r-TuBound and OTAWA respectively provide basic block WCET estimates for the Infineon C167 processor and the LPC2138 ARM7 processor. We re-used the low-level analyzers of these frameworks, and show that our high-level analyzer provides more precise constraints that our solver uses to compute tighter WCET estimates than these two frameworks.

We evaluated IBART on challenging examples from WCET benchmark suites and open-source linear algebra packages. These examples were parametric (with parameters such as array sizes and loop bounds), and our tool provided parametric estimates. All the examples we considered were solved under 20 seconds. To compare our estimates with the non-parametric results provided by r-TuBound and OTAWA, we instantiated the parameters to a number of sample values.

The results show that IBART provides better WCET estimates, though based on the same low-level analyzers. This demonstrates that our segment algorithm improves WCET estimates independently of the low-level analyzer. We thus expect that ASTs and our framework could be used by other WCET tools.

2 Illustrative Examples

This section illustrates our approach to WCET computation. We use Example 1 to demonstrate the main differences between our approach and the standard approach to WCET analysis. We then use Example 2 to present the main steps of our method: segment abstraction, estimation of WCET on ASTs, and counterexample-guided abstraction refinement with interpolation.

Example 1. The program in Figure 1 performs operation `work()` (of execution cost 3 time units) within a loop. Every 3 loop iterations, it logs some values into a file, by using operation `logValue` whose execution takes 50 time units.

```
for (i=0;i<1000;i++)
    if ((i mod 4) == 0)
        logValues()  cost=50
        work1()  cost=3
    else
        work2()  cost=3
```

Fig. 1. Example 1.

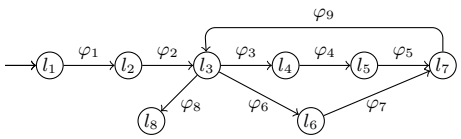

Fig. 2. CFG of Example 1.

Consider the CFG of Example 1 in Figure 2 where edges have been marked by instructions. For instance, the edge labels φ_4, φ_5, and φ_7 correspond to instructions `logValues()`, `work1()`, and `work2()`, respectively. The standard IPET algorithm would construct an ILP as follows. For each instruction φ_i, the variable X_i represents the number of times the instruction is executed in the worst-case path. The objective function for the ILP is to maximize $\sum_i X_i \cdot cost(\varphi_i)$, where $cost(\varphi_i)$ is the time taken to execute φ_i. The ILP constraints correspond to either conservation of flow, for instance, $X_7 + X_5 = X_9$ and $X_2 + X_9 = X_3 + X_6 + X_8$, or loop bounds, for instance $1000X_2 = X_3 + X_6$. Solving the ILP gives a way to estimate the WCET. However, the estimate would be imprecise, as it would find a solution that takes the expensive branch every time.

We could add a constraint specifying that the expensive branch is taken once, every 4 iterations: $3X_3 = X_6$. However, two important points are to be noted:

- First, this type of constraint that relates edges in different branches of the program is non-local. In general, the two branches that need to be related, can be far apart in the CFG. This makes decomposing the single large ILP (representing the whole program) into smaller problems hard, and to the best of our knowledge, no existing tool attempts this.
- Second, consider a version where the if-condition is replaced by `((i mod 30) == 0)`. We cannot use the constraint $29X_3 = X_6$, as this would not have a solution: the number of iterations (1000) is not divisible by 30. Alternatively, we could use less precise constraints like $29X_3 \geq X_6 \geq 29(X_3 - 1)$. Most current WCET tools would not handle the example precisely.

Consider in contrast how we obtain *local* bounds by reasoning about hierarchies of sets of segments. Let B_2 be the set of segments representing a single iteration of the loop. For example, B_2 contains segments that start at l_3, and go through l_4, l_5, l_7 (see Figure 2). This represents an iteration that goes through the expensive branch. The set of segments B_2 is represented by the regular expression $l_3(l_4l_5 \vee l_6)l_7$. Let B_1 denote a set of segments through the loop - the set of segments that start at l_3 and exit the loop. The set B_1 can be over-approximated by B_2^*. We also store the loop bound with B_1, and thus the over-approximation effectively becomes B_2^{1000}.

The counterexample-guided refinement will then refine the segment set of B_2, by splitting B_2 into two sets: the set (B_2^t) where the formula $((i \; mod \; 4) = 0)$ holds (the expensive iteration), and the set (B_2^f) where the formula $((i \; mod \; 4) = 0)$ does not hold (the cheap iteration). The set of segments B_2^t is represented by

```
if (a<b)
   for (i=0;i<n;i++)
      if (i<⌊n/2⌋)
         op1();  cost=10
      else
         op2();  cost=1
else
   op3();  cost=50
```

```
l1: if (*)
        assume a<b;  (φ1)
l2:     i:=0;  (φ2)
l3:     while (*)
            assume (i<n);  (φ3)
l4:         if (*)
                assume i<⌊n/2⌋;  (φ4)
l5:             op1();  (φ5), cost=10
            else
                assume i≥⌊n/2⌋;  (φ6)
l6:             op2();  (φ7), cost=1
l7:         i:=i+1;  (φ8)
        assume (i≥n);  (φ9)
    else
        assume a≥ b;  (φ10)
l8:     op3();  (φ11), cost=50
```

Fig. 3. Example 2 (above); written in a while-language (right).

the regular expression $l_3l_4l_5l_7$, and the set of segments B_2^t is represented by the regular expression $l_3l_6l_7$. The over-approximation B_1 will therefore become $((B_2^t)^1(B_2^f)^3)^*$. Note that this keeps the information locally: it says that one expensive iteration is followed by 3 cheap iterations. The node B_2 is hence refined into 1 iteration of B_2^t, followed by 3 iterations of B_2^f. The loop bound of 1000 would still be stored locally with B_1, requiring the total number of calls to B_2^t and B_2^f be 1000. This information is enough to obtain a precise WCET estimate. The same approach would work for the variant considered above.

Example 2 (Running example). We now explain our approach in detail using the program in Figure 3, which will be our running example through the paper. Program blocks `op1()`, `op2()`, and `op3()` are operations whose executions take 10, 1, and 50 time units, respectively (these costs are derived from a low-level timing analysis tool). In this example, we assume that program conditionals and simple assignments take 1 time unit.

It is not hard to see that for small values of the *loop bound* n, the WCET path of this program visits the outermost `else` branch containing `op3()` – when `n` is small, the execution cost of `op3()` dominates the cost of the loop. However, for larger values of `n`, the WCET path visits the `then` branch of the outermost `if` and the `for`-loop. The WCET of this example thus depends on n. Our approach discovers this fact, and infers the WCET of the program as a function of n as follows: *if* $n \leq 5$; 51 *else* $3+4n+9\lfloor n/2 \rfloor$. The computation proceeds as follows.

Control-flow graph. We construct the control-flow graph (CFG) of the program in Figure 3. First, for clarity of presentation, we transform the program in a `while`-loop language with `assume` statements — see Figure 3 (right column). We have labeled the assumptions and the transition relations (i.e. transition predicates) of instructions. For example, (φ_1) denotes the assumption $a < b$; and (φ_8) represents the transition predicate $i' = i + 1$ of the assignment $i := i + 1$.

Hierarchical segment-base abstraction. We next apply segment abstraction on the CFG of Figure 4. The initial abstraction is given by the abstract segment

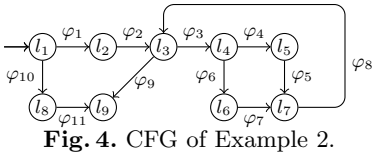

Fig. 4. CFG of Example 2.

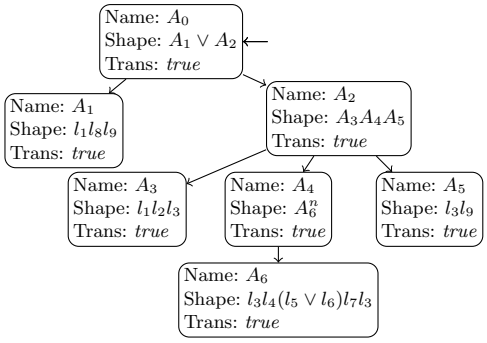

Fig. 5. Initial abstraction for Example 2.

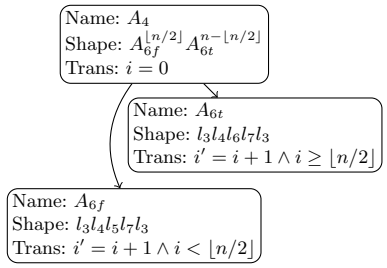

Fig. 6. Partial structure of the refined tree of Figure 5.

tree (AST) (Figure 5). The tree structure arises from the hierarchical nature of the CFG. Nodes of the tree (denoted by A_k) represent a set of execution segments, i.e., parts of program executions. Each node stores a *shape predicate* (denoted Shape) describing the paths of the segments through the CFG, and a *transition predicate* (denoted Trans) characterizing the transition relation of the segments. A shape predicate is an extended regular expression over either the children of the node, or over the CFG nodes. It is an extended regular expression, as it may contain symbolic exponents obtained, for example, from loop bounds. The transition predicate is a formula over the values of program variables at the beginning and end of segments. Note that in the formal definitions, the shape is a transition system rather than a regular expression and the nodes store more detailed information. Here, we use a regular expression for better readability.

We describe node A_2 in more detail—the other nodes are constructed similarly. The construction of A_2 in the initial abstraction is done syntactically. Node A_2 represents all segments corresponding to the then-branch of the outermost `if`. It is split into three sets of segments: (a) node A_3 denoting the set of segments before the loop, i.e., segments through the CFG nodes $l_1 l_2 l_3$; (b) node A_4 denoting the set of segments given by the loop of the CFG; and (c) node A_5 representing the set of segments after the loop of the CFG. Take n as the bound on the number of loop iterations in the CFG. For building A_4 we use node A_6 describing all segments in one iteration of the loop in the CFG. The segments in A_6 can be concatenated to cover all segments in A_4. For computing loop bounds, we use [13]. The loop bound n is noted in the shape predicate of A_4.

WCET estimation on ASTs. For each node in Figure 5, we next calculate the cost of the segments represented by it, i.e., its WCET. As each node is defined in terms of its children, we traverse the tree bottom-up. The root contains then a WCET estimate of the complete set of segments, and hence of the program.

In order to evaluate WCET on an AST, we need to supply an ILP at each node of the AST. Here, instead of presenting each ILP and solving it, we give only

a simple explanation tailored to the example under consideration. To estimate the WCET of a node, we consider the graph represented by the node. The vertices in this graph correspond to children of the node. We use the shape predicate of the node to construct the graph, and use the WCET of the children nodes to estimate the cost of the node. For example, for node A_2, we construct a graph with three nodes, with directed edges from A_3 to A_4 and from A_4 to A_5. For node A_4, we obtain a graph with one node (A_6) that can repeat at most n times. The costs of the AST nodes are calculated as:

- $cost(A_6) = cost(\varphi_3) + \max(cost(\varphi_4) + cost(\varphi_5), cost(\varphi_6 + cost(\varphi_7)) + cost(\varphi_8) = 13$
- $cost(A_4) = n \cdot A_6 = 13n$
- $cost(A_3) = cost(\varphi_1) + cost(\varphi_2) = 2$
- $cost(A_5) = cost(\varphi_9) = 1$
- $cost(A_2) = cost(A_3) + cost(A_4) + cost(A_5) = 3 + 13n$
- $cost(A_1) = cost(\varphi_{10}) + cost(\varphi_{11}) = 51$
- $cost(A_0) = \max(cost(A_1), cost(A_2)) = \max(51, 3{+}13n) = \mathit{if}\ n \leq 3\ ;\ 51\ \mathit{else}\ 3{+}13n$

The WCET estimate of our running example is given by $cost(A_0)$, and depends on the value of n, i.e., when $0 \leq n \leq 3$ the WCET is different than in the case when $n > 3$. To ensure that the computed WCET estimate is precise, we need to ensure that our abstraction did not use an infeasible program path to derive the current WCET estimate. We therefore pick a concrete value of n for each part of the WCET estimate, and check whether the corresponding witness worst-case path is feasible. If it is, the derived WCET estimate is actually reached by the program and we are done. Otherwise, we need to refine our AST. In our example, we thus have the following two cases:

- *Case 1:* $n \leq 3$. We pick $n = 1$. The WCET estimate of A_0 is then 51. Here, the witness trace is $l_1 l_8 l_9$. This trace is a feasible trace of Figure 3.
- *Case 2:* $n > 3$. We pick $n = 4$ and the witness trace is $l_1 l_2 \, (l_3 l_4 l_5 l_7)^4 l_3 l_9$, which is infeasible, and we proceed to the refinement step.

Counterexample-guided refinement using interpolation. We refine the AST of Figure 5 using the infeasible trace. We traverse the tree top-down to refine each node of the counterexample. We refine the children of the node corresponding to a node in the counterexample with new context information obtained from the counterexample, via interpolation. We detail our refinement approach only for A_4, the rest of the nodes are refined in a similar way. By analyzing the predecessor segments of A_4 in the counterexample, we derive $i = 0$ as a useful property for our refinement. This property is obtained using the same refinement process that we now describe for A_4.

To refine A_4, we analyze its children, that is n repetitions (i.e., iterations) of A_6. In what follows, we denote by i_k the value of the variable i after the k-th iteration of A_6, for $0 \leq k \leq n$. Let i_0 denote the value of i before A_4. We compute the property $i_1 = i_0 + 1$ summarizing the first iteration of A_6, where the summarization process includes interpolation-based refinement. Similarly, from the second iteration of A_6 we compute $i_2 = i_1 + 1$. Hence, at the second iteration of A_6 the formula $i_0 = 0 \wedge i_1 = i_0 + 1 \wedge i_2 = i_1 + 1 \wedge n = 4$ is a valid property of the witness trace; let us denote this formula by A (recall that we fixed $n = 4$ above). However, after the second iteration of A_6 we have $(i_2 < n) \wedge (i_2 < \lfloor n/2 \rfloor)$

as a valid property of the witness trace; we denote this formula by B. Observe that $A \wedge B$ is unsatisfiable, providing hence a counterexample to the feasibility of the current witness trace. From the proof of unsatisfiability of $A \wedge B$, we then compute an interpolant I such that $A \implies I$, $I \wedge B$ is unsatisfiable, and I uses only symbols common to both A and B. We derive $i_2 \geq \lfloor n/2 \rfloor$ as the interpolant of A and B.

We now use the interpolant $i_2 \geq \lfloor n/2 \rfloor$ to refine the segment abstraction of A_6, as follows. The interpolant $i_2 \geq \lfloor n/2 \rfloor$ is mapped to the predicate $i \geq \lfloor n/2 \rfloor$ over the program variables. We then split A_6 into two nodes: node A_{6f} denoting segments where $i \geq \lfloor n/2 \rfloor$ does not hold, and node A_{6t} describing segments where $i \geq \lfloor n/2 \rfloor$ holds. The interpolants $i_1 = i_0 + 1$ and $i_2 = i_1 + 1$ computed from the first and second iteration of A_6 yield the transition predicate $i' = i + 1$; this formula holds for every segment in A_6, and hence also in A_{6f} and A_{6t}. The transition predicates of A_{6t} and A_{6f} are then used to compute the new shape predicate $A_{6f}^{\lfloor n/2 \rfloor} A_{6t}^{n - \lfloor n/2 \rfloor}$ for A_4. The resulting (partial) refined AST is given in Figure 6. This refined AST yields the WCET estimate *if* $n \leq 5$; 51 *else* $3 + 4n + 9\lfloor n/2 \rfloor$, which is a precise WCET estimate for the program in Figure 3.

3 Problem Statement

Instruction and predicate language. We express program instructions, predicates, and assertions using standard first-order logic. Let $\mathcal{F}(X)$ represent the set of linear integer arithmetic *formulae* over integer variables X. We represent an instruction of a program as a formula from $\mathcal{F}(V \cup V')$. Intuitively, a variable $v \in V$ and its primed version $v' \in V'$ represent the values of the program variable `v` before and after the execution of the instruction, respectively. For example, an instruction `i := i + j` in a C-like language would be represented as $i' = i + j$.

Program model. We model programs with assignments, conditionals, and loops, over a finite set of scalar integer variables V. While we do not handle procedure calls, our techniques can be generalized to non-recursive procedure calls. We represent programs by their control-flow graphs. A *control-flow graph* (CFG) is a graph $G = \langle \mathcal{C}, E, V, \Delta, \iota_0, \mathit{init}, F \rangle$, where (a) $\mathcal{C}$ is a set of nodes (representing control-flow locations); (b) $E \subseteq \mathcal{C} \times \mathcal{C}$ is a set of edges; (c) V is the set of program variables; (d) $\iota_0 \in \mathcal{C}$ is an initial control-flow location; (e) $\mathit{init} \in \mathcal{F}(V)$ is an initial condition on variables; (f) $F \subseteq \mathcal{C}$ is a set of final program locations; and (g) $\Delta : E \to \mathcal{F}(V \cup V')$ maps edges to the instruction that is executed when the edge is taken. We denote *program states* by pairs of the form (l, σ) where $l \in \mathcal{C}$ and σ is a valuation of program variables V.

Semantics. The semantics $[\![G]\!]$ of a CFG G is the set of *finite* sequences of program states (called *traces*) $(l_0, \sigma_0) \ldots (l_k, \sigma_k)$ such that: (a) $l_0 = \iota_0$ and $\sigma_0 \models \mathit{init}$, (b) $l_k \in F$, and (c) $\forall 0 \leq i < k . (l_i, l_{i+1}) \in E \wedge (\sigma_i, \sigma_{i+1}) \models \Delta((l_i, l_{i+1}))$. Note that we assume that the program represented by G is terminating.

Cost model. We assume a simple cost model for instructions given by a function $\mathit{cost} : E \to \mathbb{N}$ where $\mathit{cost}((l_1, l_2))$ is the maximum execution time of the instruction from l_1 to l_2. We also refer to costs as *weights*. The weight $\mathit{cost}(\pi)$ of

a trace $\pi = (l_0, \sigma_0) \dots (l_k, \sigma_k)$ is $\Sigma_{i=0}^{k-1} cost((l_i, l_{i+1}))$. In practice, costs of edges are obtained from a low-level architecture dependent analyzer. Note that even this simple cost model can already capture some information about the context of an instruction's execution such as some cache hit/miss information. For example, if the low-level analysis determines that an instruction will always be a cache hit, it can provide a lower cost accordingly.

Problem statement. The *worst-case execution time* $WCET(G)$ of a CFG G is defined by $WCET(G) = \max_{\pi \in [\![G]\!]} cost(\pi)$. The task of the *WCET estimation problem* is: Given a CFG G, compute a number e such that $e \geq WCET(G)$. The additional aim is to compute an estimate e that is tight, i.e., close to $WCET(G)$.

The rest of this paper describes the main steps of our approach to solving this problem: segment abstraction (Section 4), WCET estimation for segment abstractions (Section 5), and counterexample-guided refinement (Section 6). We summarize our algorithm in Section 7, describe the parametric extension in Section 8, and present our tool and experimental results in Section 9.

4 Segment Abstraction for Flow Analysis

Our abstraction technique for flow analysis is based on the hierarchical segment abstraction of [8, 6]. We adapt the definitions of hierarchical segment abstraction from [6] to the setting of worst-case execution time analysis.

Let us fix a CFG $G = \langle \mathcal{C}, E, V, \Delta, \iota_0, init, F \rangle$. A *segment* is a finite sequence of program states (i.e., pairs of control-flow locations and variable valuations).

Abstract Segment Trees (ASTs). An *abstract segment tree* T is a rooted tree, where each node represents a set of segments. Intuitively, the segments of each node are composed from the segments of its children. Each node is a tuple $(segPred, children, shape, Init, Exit, slMin, slMax, gMax)$ where:

- $segPred \in \mathcal{F}(V \cup V')$ is a relational predicate satisfied by the initial and final variable valuations of all the segments represented by the current node;
- For internal nodes, the set *children* is the set of its children in T, and for leaf nodes, *children* is a subset of the control-flow edges E of the CFG G;
- $shape \subseteq children \times children$ is a transition relation on *children* — for leaf nodes, where $children \subseteq E$, we have that $((l_0, l_1), (l_2, l_3)) \in shape$ if $l_1 = l_2$;
- $Init, Exit \subseteq children$ are a set of initial child nodes and exit child nodes;
- $gMax \in \mathbb{N} \cup \{\infty\}$ is a bound on the maximal number of segments of child nodes in a segment of the current node; and
- $slMin, slMax : children \to \mathbb{N} \cup \{\infty\}$ are functions that map each child to the minimum and maximum possible consecutive repetitions of segments represented by the child in a segment represented by the current node.

We use the functions *gMax*, *slMin*, and *slMax* to store information about bounds on the number of times certain iterations of a loop can be repeated. In practice, these are computed using standard loop bound computation techniques.

Remark 1. Note that the quantitative information stored in the AST, i.e., *slMin*, *slMax*, and *gMax*, is different from the quantitative information in [6]. In [6],

the interest was in limit-average estimation where storing bounds on segment length is useful; while here, bounds on the number of segments is more useful.

Example 3. We clarify the definition of ASTs using Figures 5 and 6. In node A_0 from Figure 5 and node A_4 from Figure 6 (with parameter $n = 4$), the AST representation is in Table 1. In Figure 5 and Figure 6, the components *shape*, *Init*, *Exit*, *slMin*, and *slMax* have been combined into one regular expression.

Component	Value in A_1 (Fig. 5)	Value in A_4 (Fig. 6)
segPred	*true*	$i = 0$
children	$\{A_1, A_2\}$	$\{A_{6t}, A_{6f}\}$
shape	$\emptyset$	$\{(A_{6f}, A_{6f}), (A_{6f}, A_{6t}), (A_{6t}, A_{6t})\}$
Init	$\{A_1, A_2\}$	$\{A_{6f}\}$
Exit	$\{A_1, A_2\}$	$\{A_{6t}\}$
gMax	1	4
slMax	$slMax(A_1) = 1$ $slMax(A_2) = 1$	$slMax(A_{6f}) = 2$ $slMax(A_{6t}) = 2$
slMin	$slMin(A_1) = 1$ $slMin(A_2) = 1$	$slMin(A_{6f}) = 2$ $slMin(A_{6t}) = 2$

Table 1. Definition of ASTs

AST Semantics. We define $[\![A]\!]$ for a node A in terms of its children. The semantics $[\![T]\!]$ of the AST T is then the semantics of the root node. AST T is a *sound abstraction* for a CFG G if $[\![G]\!] \subseteq [\![T]\!]$. We need two notions to aid the definition. Let $s_1 = (l_0^1, \sigma_0^1) \dots (l_n^1, \sigma_n^1)$ and $s_2 = (l_0^2, \sigma_0^2) \dots (l_m^2, \sigma_m^2)$ be two segments.

- the function $form(s_1)$ represents the serial composition of formulas $\Delta((l_i, l_{i+1}))$ for $0 \leq i < k$, i.e., it is the relation on the initial and final program states of s_1 implied by the instructions of s_1; and
- the segment $s_1 \oplus s_2$ is the concatenation of s_1 and s_2 where the last state of s_1 is substituted for the first state of s_2, i.e., $s_1 \oplus s_2 = (l_0^1, \sigma_0^1) \dots (l_n^1, \sigma_n^1)(l_1^2, \sigma_1^2) \dots (l_m^2, \sigma_m^2)$.

Let $A = (children, shape, segPred, Init, Exit, slMin, slMax, gMax)$ be a node in T. Segment s is in $[\![A]\!]$ iff $s = s_0 \oplus \dots \oplus s_n$ and there exist $c_0, \dots c_n$ such that:

(a) for each i, $s_i \in [\![c_i]\!]$ where $c_i \in children$—for leaf nodes, c_i is a control-flow edge (say (l_i, l_j)) and we let $((l_i, \sigma_i), (l_j, \sigma_j)) \in [\![c_i]\!]$ if $(\sigma_i, \sigma_j) \models \Delta((l_i, l_j))$;
(b) for all $0 \leq i < n$, we have that $(c_i, c_{i+1}) \in shape$;
(c) initial and final variable valuations of s satisfy *segPred*: $form(s) \Rightarrow segPred$;
(d) $c_0 \in Init$ and $c_n \in Exit$, and for each maximal contiguous sequence $c_p c_{p+1} \dots c_q$ of the same child, $slMin(c_p) \leq q - p + 1 \leq slMax(c_p)$, and
(e) $n \leq gMax$.

Example 4. Consider the CFG from Figure 4, and its AST in Figure 5. A segment π passing through locations $l_1 l_2 l_3 l_4 l_5 l_7 l_3 l_9$ is in the semantics of the node A_2, as it can be split into three segments: (a) the prefix π_1 through $l_1 l_2 l_3$; (b) the middle π_2 through $l_3 l_4 l_5 l_7 l_3$; and (c) the suffix π_3 through $l_3 l_9$. As π_1 is in $[\![A_3]\!]$, π_2 is in $[\![A_4]\!]$, and π_3 is in $[\![A_5]\!]$, we have that $\pi \in [\![A_2]\!]$.

Reducibility of CFGs and the Initial Abstraction. We assume that CFGs are reducible, i.e., that every loop has a unique entry. This assumption holds for

programs in high-level programming languages. The function $InitAbs(G)$ takes a CFG an input, and constructs an AST T such that $[\![G]\!] \subseteq [\![T]\!]$. The construction is simple (see, for example, Figure 5). The main point to note is that each maximal strongly connected component (i.e., a loop) corresponds to a node with just one child. The child represents segments corresponding to individual iterations. The *segPred* predicate for each node is initially set to true.

Proposition 1. *Let G be a CFG. If $T = InitAbs(G)$, we have that $[\![G]\!] \subseteq [\![T]\!]$.*

5 Evaluating WCET on ASTs

In the previous section, we discussed segment abstractions (ASTs). Here, we present a method to compute WCET estimates from ASTs. Given an AST T, let $WCET(T) = \sup_{\pi \in [\![T]\!]} cost(\pi)$. If T is a sound abstraction of G, $[\![T]\!] \supseteq [\![G]\!]$ and hence, $WCET(T) \geq WCET(G)$. Therefore, if an AST T is a sound abstraction of a CFG G, $WCET(T)$ is an over-approximation of $WCET(G)$.

5.1 Maximum-Weight Length-Constrained Paths

We take a recursive approach to computing $WCET(T)$ for an AST T. The WCET of each node is computed using the WCET values of its children by reducing the problem to the the *length-constrained maximum-weight path* problem.

Let $\langle V, E \rangle$ be a graph with vertices V and edges E. Given initial and final vertices v_{in} and v_{out}, cost function $cost : V \to \mathbb{N}$, global length bound $g_{max} \in (\mathbb{N} \cup \{\infty\})$, and local bounds $l_{min}, l_{max} : V \to \mathbb{N} \cup \{\infty\}$, the length-constrained maximum-weight path problem asks for the maximum weight path: (a) starting at v_{in} and ending at v_{out}; (b) of length at most g_{max}; and (c) with every maximal contiguous repetition of a vertex v in the path having length at least $l_{min}(v)$ and at most $l_{max}(v)$. Without loss of generality, we assume that $v_{in} \neq v_{out}$ and that v_{in} and v_{out} have only outgoing and incoming edges, respectively.

Reduction. Given a node $A = (children, shape, segPred, Init, Exit, slMin, slMax, gMax)$, we define a graph with vertices being $children \cup \{v_{in}, v_{out}\}$, edges being $shape \cup \{(v_{in}, v) \mid v \in Init\} \cup \{(v, v_{out}) \mid v \in Exit\}$, starting and ending vertices being v_{in} and v_{out}, and the global bound being $gMax$, respectively. For a child c, we have $cost(c) = WCET(c)$, $l_{min}(c) = slMin(c)$, and $l_{max}(c) = slMax(c)$. We call this graph with the corresponding functions the *semantic structure graph* for A and denote it by $Gr(A)$.

Theorem 1. *For each node A in an AST, $WCET(A)$ is equal to the weight of the length-constrained maximum-weight path in $Gr(A)$.*

Hardness. The length-constrained maximum-weight path problem is at least as hard as Unambiguous-SAT. Hence, a Ptime algorithm implies that Np = Rp, i.e., non-deterministic and randomized polynomial time are the same. However, considering g_{max} as a parameter, the problem is fixed parameter tractable Fpt.

5.2 Encoding Optimal Paths

We first discuss the standard technique used in WCET tools to find optimal paths in graphs—the implicit path enumeration technique (IPET) [15, 16]. We emphasize that the graph in standard techniques for WCET estimation is the control-flow graph (i.e., a syntactic object), while in our technique it is the semantic structure graph. However, similar principles apply for the graph problem in both cases and we briefly recall the IPET approach as a starting point.

Implicit path-enumeration technique. IPET encodes paths in a graph as an integer linear program (ILP). The encoding uses variables X_v and $X_{(u,v)}$ to represent the number of times vertex v and edge (u, v) occur in the path.

Objective function. The weight of a path is given by $\sum_{v \in V} cost(v) \cdot X_v$. Hence, the objective of the ILP is to maximize $\sum_{v \in V} cost(v) \cdot X_v$.

Kirchhoff's law. To ensure that X_v and $X_{(u,v)}$ values correspond to a real path, we have: for each $v \in V$, we have $X_v = \sum_{(u,v)} X_{u,v} + start_v = \sum_{(v,w)} X_{v,w} + end_v$ where $start_v = 1$ (resp. $end_v = 1$) for $v = v_{in}$ (resp. $v = v_{out}$); otherwise, $start_v = 0$ (resp. $end_v = 0$). Intuitively, for each vertex, the number of incoming edges is equal to the number of outgoing edges, except for v_{in} and v_{out}.

Connectivity. However, Kirchhoff's laws are not sufficient to ensure that the values for X_v and $X_{(u,v)}$ form a feasible path. This is because the disconnected components problem, i.e., the values may correspond to a feasible path along with additional cycles that are disconnected from the path.

Example 5. Consider a graph with vertices $\{v_{in}, v_{out}, v_1, v_2\}$ and edges $\{(v_{in}, v_{out}), (v_{in}, v_1), (v_1, v_2), (v_2, v_1)\}$. The values $X_{v_{in}} = X_{v_{out}} = 1$, $X_{v_1} = X_{v_2} = 10$, $X_{(v_{in},v_{out})} = 1$, $X_{(v_{in},v_1)} = 0$, and $X_{(v_1,v_2)} = X_{(v_2,v_1)} = 10$ satisfy the Kirchhoff's law. However, these values do not correspond to a path as the cycle $(v_1 \to v_2 \to v_1)$ is disconnected from the rest of the path.

Standard IPET formulations overcome this problem through loop bounds—constraints are added to ensure that a loop is executed at most a constant multiple of times an edge to enter the loop is taken. In Example 5, we would add $X_{(v_1,v_2)} \leq c \cdot X_{(v_{in},v_1)}$ where c is the loop bound for the cycle $v_1 \to v_2 \to v_1$. For structured (reducible) graphs, this approach works very well as each cycle has a unique entry. However, for irreducible graphs, a cycle may not have a unique entry—instead, we need to write such constraints for each subset of vertices which may form a cycle, and each entry to such a cycle, adding an exponential number of constraints just to ensure connectivity (see, for example, [16]).

Semantic structure of loops. While the IPET approach works well in the standard WCET analysis framework even for irreducible graphs, the simple loop bound approach to handling connectivity does not work directly as the vertices in the semantic structure graph may represent not only instructions, but also more complex segments (such as different iterations of a loop).

- While CFGs and graphs arising from real programs may be irreducible in the IPET approach, the "degree of irreducibility" is usually low, i.e., only a few additional constraints are necessary to ensure connectivity. On the other hand, since the graphs arising from AST correspond to the semantic

structure of loops, they may be highly irreducible (for example, a clique) and an exponential number of additional constraints may be necessary.
- A further reason why an IPET-like approach is not possible for semantic graphs is that there may not exist bounds on cycles in the semantic graphs.

Example 6. Consider the logging example from Section 2 with the modification of the if condition from `i mod 4 == 0` to `i mod 4 == 0` $\wedge$ `started_logging`. The boolean variable `started_logging` is set to false initially, and is non-deterministically set to true at some point during the execution of the loop. Now, consider the three segments sets corresponding to the iterations where the following hold: (a) $\neg$*started_logging* (say I_1), (b) *started_logging* $\wedge\, i\%4 == 0$ (say I_2), and (c) *started_logging* $\wedge\, i\%4 \neq 0$ (say I_3). In the semantic structure graph, there is a cycle containing vertices corresponding to I_2 and I_3; and an entry to this cycle from the vertex corresponding to I_1. However, there is no bound on the number of times this cycle can be executed in terms of the number of times the entry is taken.

The LC-IPET encoding. We now present our ILP encoding for the length-constrained maximum-weight path problem. This encoding works for (a) firstly, irreducible graphs with only linearly many constraints; and (b) secondly, no bounds on the execution of cycles are required. Hence, this encoding is of interest for WCET analysis independent of the rest of our framework. Given a graph G, we denote the encoding into ILP by LC-IPET(G).

Objective and Kirchhoff's laws. The objective function and Kirchhoff's law constraints are as in the classical IPET approach.

Global and local bounds. The global bound and the local bounds can be ensured using $\sum_{v \in V} X_v \leq g_{max}$ and $l_{min}(v) \cdot \sum_{u|(u,v)\wedge v \neq u} X_{(u,v)} \leq X_v \leq l_{max}(v) \cdot \sum_{u|(u,v)\wedge v \neq u} X_{(u,v)}$ for each vertex v in the graph.

Connectivity flow. We ensure connectivity of the path generated by ILP using an auxiliary flow that goes through only the edges in the path. Intuitively, we ensure that some flow is lost at each visited vertex (i.e., it is a partial sink) except the start vertex which may generate flow (i.e., only the start vertex can be a source). Hence, flow in a component of the path is feasible if and only if it is connected to the start vertex. We use the variables $F_{(u,v)}$ to represent the auxiliary flow through an edge. We have the following:
- Flow is non-negative only in visited edges: for all edges, $F_{(u,v)} \geq 0$ and for all edges, $|V|X_{(u,v)} \geq F_{(u,v)}$. Hence, if $X_{(u,v)}$ is zero, we have $F_{(u,v)} = 0$.
- Every visited vertex other than the start vertex loses some flow: for all vertices $v \neq v_{in}$, $\sum_{(u,v)} F_{(u,v)} - \sum_{(v,w)} F_{(v,w)} \geq X_v$. If X_v is positive (i.e., v is visited), $(\sum_{(u,v)} F_{(u,v)} - \sum_{(v,w)} F_{(v,w)})$ is positive, i.e., the incoming flow to v is greater than the outgoing flow from v.

Theorem 2. *Given graph G, initial and final vertices, and local and global length bounds, the optimal value of LC-IPET(G) is the cost of the length-constrained maximum weight path in G.*

Proof. Clearly, the objective function of LC-IPET(G) corresponds exactly to the weight of a set of nodes in the graph. Hence, it is sufficient to show that every feasible solution of LC-IPET(G) corresponds to a feasible path in G, and vice versa. That every feasible solution of LC-IPET(G) encodes at least one path that follows the local and global bounds is easy to check.

Given a solution to LC-IPET(G), we show that the X_v and $X_{(u,v)}$ values form a path. By Kirchhoff's laws, the value of X_v and $X_{(u,v)}$ consist of a path along with some possibly disconnected components of visited vertices. We show that there cannot be any disconnected components using the auxiliary flow. Suppose $\overline{V}$ and $\overline{E}$ are the subset of vertices and subset of edges which form a disconnected component. Now, we have that for each vertex $v \in \overline{V}$, $\sum_{(u,v)} F_{(u,v)} - \sum_{(v,w)} F_{(v,w)} > 0$. Hence, we have $\sum_{v \in \overline{V}} (\sum_{(u,v)} F_{(u,v)} - \sum_{(v,w)} F_{(v,w)}) > 0$ or equivalently, $\sum_{v \in \overline{V}} \sum_{(u,v)} F_{(u,v)} - \sum_{v \in \overline{V}} \sum_{(v,w)} F_{(v,w)} > 0$ However, note that for edges (u, v) or (v, w) that enter or leave the component we have $F_{(u,v)} = 0$ as the flow is positive if and only if (u, v) is visited. Therefore, we have that both $\sum_{v \in \overline{V}} \sum_{(u,v)} F_{(u,v)}$ and $\sum_{v \in \overline{V}} \sum_{(v,w)} F_{(v,w)}$ are equal to the sum of flow through all edges in $\overline{E}$. This leads to a contradiction as we need $\sum_{v \in \overline{V}} \sum_{(u,v)} F_{(u,v)} - \sum_{v \in \overline{V}} \sum_{(v,w)} F_{(v,w)} > 0$.

Now, given a length-constrained path π in G, we provide a satisfying assignment to the variables in LC-IPET(G). Clearly, the X_v and $X_{(u,v)}$ variables are assigned to the number of times v and (u, v) are visited in π, respectively. We need to find satisfying assignments for $F_{(u,v)}$. For this, we construct separate simple paths π_v from v_{in} to v for each visited vertex v through edges in π. We let $F_{(x,y)} = \sum_v X_v \cdot \pi_v[(x, y)]$ where $\pi_v[(x, y)]$ is equal to 1 if (x, y) occurs is π_v and 0 otherwise. It is easy to see that these $F_{(u,v)}$ values satisfy the auxiliary flow constraints. Intuitively, the auxiliary flow is composed of separate flows of magnitude X_v going from v_{in} to v; call each such flow the flow to v. Now, for every $v' \neq v$, the flow to v' enters and leaves v in the same magnitude. However, the flow to v stops at v, ensuring that the incoming flow to each visited vertex is greater than the outgoing flow by 1.

Given an optimal solution to LC-IPET($Gr(A)$) the worst-case path can be computed using an algorithm for finding Eulerian paths in multi-graphs. Summarizing the approach, given an AST T, we compute the WCET of each node (using its children's WCET values) by reducing to the length-constrained maximum-weight problem. This problem is then solved through the LC-IPET encoding, and the worst-case path can be computed from the solution to the ILP.

Optimizations and practicality. The semantic structure graphs that arise in practice often allow us to avoid the flow variables in the LC-IPET encoding.

The first case where we can avoid the flow variables is the case where we have no global bound. In our methodology, global bounds arise due to loop bounds in the input program and hence, in each AST node that does not deal with loops, there is no global bound ($g_{max} = \infty$). In this case, either no cycle is reachable in the graph $Gr(A)$, or the worst-case path has weight ∞. Hence, in this case, one can avoid solving the ILP and instead use simpler polynomial time algorithms.

If we have a global bound, we are analyzing different kinds of iterations of a loop, i.e., branches through a loop. While the semantic structure of the loops may be arbitrarily complicated, most of the programs generate semantic structures that fall into several common easily analyzable patterns:

- Progressive phases: These are cases where the execution of the loop is divided into phases, i.e., in each phase only one particular branch through the loop is taken, and this branch is never taken after the completion of the phase. For example, the loop from Example 2 instantiated with $n = 10$ is divided into two phases, one for $i < 5$ and one for $i \geq 5$. The evaluation is easy in such cases as the semantic structure graph is a directed acyclic graph and there cannot be any disconnected cycles. A large number of loops in practice fall into this class (see [17] for an empirical study).
- Cyclic phases: These are cases where the execution of the loop is divided into phases, which repeat in a cycle. For example, the loop in the program from Example 1 is divided into two phases, one for (i % 4 == 0) and one for (i % 4 $\neq$ 0). Again, in such cases, auxiliary flow variables are unnecessary as there is exactly one cycle in the semantic structure graph.

6 Interpolation for AST Refinement

Once the WCET path is computed from an AST, we check if the computed worst-case path is feasible in the CFG. If such a path is infeasible, we call it an infeasible witness trace. Formally, a *witness trace* (*wit*) is a sequence of CFG nodes that witnesses the current WCET estimate. It is obtained from the techniques presented above. We now describe our interpolation-based AST abstraction refinement algorithm for an infeasible witness trace.

AST refinement algorithm. The main idea of Algorithm 1 is to trace the *infeasible* witness trace (*wit*) through the abstract segment tree (AST), and refine the AST nodes touched by *wit*. For each node N, we discover the segment predicates that are important at the interface of the subtree rooted at N and the rest of the AST. When processing an AST node, we split each child (visited by the *wit*) with some new "context" information, obtained via interpolation. Algorithm 1 takes four inputs: (a) an AST T, (b) a node N in T, (c) an infeasible witness trace *wit* that is a segment in N, and (d) a formula SumAbove that summarizes the part of the original witness trace *wit* outside of the subtree rooted at N. Initially, the algorithm is called with N being the root of the AST, and the formula SumAbove is set to *true*.

Refinement procedure. We now detail the REFINE procedure of Algorithm 1 and illustrate it on our running example. For a node N, the procedure REFINE obtains a sequence $s = s_0 s_1 \dots s_k$ of children of N that the witness trace *wit* passes through (line 1 of Alg. 1). Note that a child can be repeated in s. The *wit* can be split into segments, where the i-th segment wit_i of *wit* belongs to the i-th child s_i. Recall that the infeasible *wit* of Example 2 was $wit = l_1 l_2 \ (l_3 l_4 l_5 l_7)^4 l_3 l_9$ for $n = 4$. The CFG of Example 2 is Figure 4, and its AST is in Figure 5.

Algorithm 1 Procedure Refine

Input: AST T, node N in T, witness trace *wit* and formula SumAbove
Output: Refined AST T

```
1: s ← TraceWit(N,wit)
2: for all i ∈ {0,...,|s|} do
3:     context ← SumAbove ∧ SumLR(s,wit,N) ∧ segPred(N)
4:     child ← form(projection(wit,s_i))
5:     I ← Interpolate(child,context)                    ▷ context ∧ child unsat
6:     r_t ← addToTree(s_i,I); r_f ← addToTree(s_i,¬I)
7:     Refine(T, r_t, I∧ segPred(s_i), projection(wit,s_i))   ▷ Recursively refine.
8:     StrengthenDown(r_f)
9:     RemoveFromTree(T,s_i)
10: StrengthenUp(N)
```

Consider the node A_4 in Figure 5 as the node N. The node A_4 represents a loop and the node A_6 a single iteration. The sequence s is then A_6^4.

Next, each child s_i is refined using wit_i (loop at line 2). The variable *context* stores a formula that summarizes what we know about *wit* outside of s_i (line 3). It is obtained as a conjunction of the formula SumAbove, the segment predicate of N, and the information computed by the function SumLR(). The function SumLR() computes information about the trace *wit* as it passes through the children of N other than s_i. When refining s_i, SumLR() returns a formula $\wedge_{k<|s|\wedge(k\neq i)} J_k$, where J_k is $form(wit_k)$ and wit_k is the part of the *wit* going through the node s_k. (*form* was defined in Section 4.) The variable *child* stores a formula that summarizes what we know about the *wit* inside of s_i (line 4). It is computed as $form(wit_i)$, where wit_i was obtained by the projection of *wit* to the node s_i. For our running example, consider the third iteration of A_6. In this case, the value of *context* is $i_0 = 0 \wedge \bigwedge_{k=0}^{1} i_{k+1} = i_k + 1$ (we show only the relevant part of the formula) and the value of *child* is $i_2 < 4 \wedge i_2 < 2$ (4 and 2 are n and $n/2$, respectively).

Note that *child* $\wedge$ *context* is unsatisfiable, as (a) the original *wit* is infeasible, and (b) *context* and *child* summarize the *wit*. We hence can use interpolation to infer a predicate explaining the infeasibility at the boundary of the subtree of child s_i and the rest of the AST. We compute an interpolant I from the proof of unsatisfiability *context* $\wedge$ *child* (line 5) such that *context* $\implies I$ and *child* $\wedge I \implies \bot$, where I is over only those variables that are common to both *context* and *child*. In our running example, we obtain the interpolant $i_2 \geq 2$.

Using the computed interpolant I, we next replace the node s_i by two nodes r_t and r_f (line 6). The node r_t is like s_i (in terms of its children in the AST), but has a transition predicate equal to $segPred(s_i) \wedge I$. Similarly, for r_f we take its transition predicate $segPred(r_f)$ as $segPred(s_i) \wedge \neg I$. In this way, each of these nodes has more information about its context than s_i had. We can further refine these two nodes and use them in the AST T instead of s_i.

Observe that for r_t we added the predicate I to its transition predicate. As *child* $\wedge I$ is unsatisfiable, the trace *wit* is not represented in r_t. The node r_t can thus be refined by a recursive call to the Refine procedure (line 7). As

Algorithm 2 Precision Refinement for WCET

```
1: Input: Program P;                    Output: WCET of P
2: Build the CFG G of P;
3: Construct the AST T corresponding to G;    // Abstraction
4: for each node A in T (post-order traversal) do    // Evaluation
5:     construct the LC-IPET(Gr(A));
6:     WCET(A) ← optimum of LC-IPET(Gr(A));
7: w_i ← witness trace corresponding to WCET(T)
8: if w_i is infeasible then
9:     Refine(T,w_i,root(T),true) (Algorithm 1)    // Refinement
10:    go to line 4;
11: return WCET(T).
```

child $\wedge \neg I$ is satisfiable, for r_f there is nothing more to learn from the *wit*. We simply strengthen the node, that is, propagate the new predicate, $\neg I$, to the children of r_f. This is done by calling the *StrengthenDown*() function (line 8), which propagates the new information $\neg I$ to the children t of the node r_f. To this end, it checks whether it finds a segment in the node t which is excluded from the node by $\neg I$, and then calls the REFINE procedure to perform refinement with the discovered segment used as a witness trace. In our running example r_f corresponds to A_{6f} and r_t is A_{6t}. Finally, the function *StrengthenUp*() uses the information discovered during the refinement process for the children of a node N, and strengthens the *segPred* predicate of N (line 10).

7 WCET Computation Algorithm

Algorithm 2 describes our approach to computing precise WCET estimates. Given a program $\mathcal{P}$, we first construct its CFG (line 2), and build the corresponding initial AST T (line 3). For the AST T, we compute the $WCET(T)$ (lines 4-6), using the LC-IPET approach detailed in Section 5. The WCET is precise if a feasible program path exhibits the WCET. We therefore check if the witness trace exhibiting the WCET is indeed feasible (line 8). If not, we refine our current AST using Algorithm 1 (line 9).

8 Parametric WCET Computation

We now extend our techniques to handle parametric programs and return a parametric WCET estimate, i.e., one that may depend on the parameter values.
Parameters. *Program parameters* $P \subseteq V$ are program variables whose values do not change in any execution. Given a valuation $val(P) : P \to \mathbb{N}$ of P, the CFG $G_{val(P)}$ is obtained by replacing variables in P by their values given by $val(P)$.
Solution language. Let $\mathcal{A}(P)$ be the set of arithmetic expressions over P. The language of *disjunctive expressions* $\mathcal{E}(P)$ consists of sets of pairs $W = \{D_0 \mapsto N_0, \ldots, D_k \mapsto N_k\}_i$ where D_i and N_i are boolean and arithmetic expressions

over P, respectively; and we have $\bigvee_i D_i = true$ and $\forall i \neq j . D_i \wedge D_j \implies \bot$. Intuitively, the value of W is N_i when D_i holds. Given a valuation $val(P)$, we write $W[val(P)]$ for the explicit integer value of $N_i[P]$ where $D_i(val(P))$ holds.

It is easy to define standard arithmetic, comparison, and max operators over $\mathcal{E}(V)$. For example, if $W^1 = \{D_i \mapsto N_i\}_i$ and $W^2 = \{D_j \mapsto N_j\}_j$, then $W^1 + W^2 = \bigcup_{i,j}\{(D_i \wedge D_j \mapsto N_i + N_j)\}$, and $\max(W^1, W^2) = \{D_i \wedge D_j \wedge N_i > N_j \mapsto N_i\} \cup \{D_i \wedge D_j \wedge N_i \leq N_j \mapsto N_j\}$.

Problem statement. A *parametric WCET estimate* $WCET_p(G, P)$ of a CFG G is an expression in $\mathcal{E}(P)$, such that for all valuations $val(P)$ of parameters, $WCET_p(G, P)[val(P)] \geq WCET(G_{val(P)})$. The parametric WCET estimate, $WCET_p(G, P)$, is an over-approximation $WCET(G_{val(P)})$ for each valuation $val(P)$. The task of our *parametric WCET estimation problem* is: Given a CFG G and a set of parameters P, compute $WCET_p(G, P)$, the parametric WCET.

The Parametric Framework. We describe the changes necessary to adapt our WCET estimation framework to the parametric case.

Abstraction. A *parametric AS T* is similar to an AST, except that that for each node, *gMax* has type $\mathcal{A}(P)$, and *slMax* and *slMin* have type *children* $\rightarrow \mathcal{A}(P)$.

Evaluation. The evaluation of $WCET(T)$ for a parametric AST is more involved than for a standard AST. This procedure is detailed in Section 8.1.

Refinement. The refinement procedure from Section 6 can be used directly in the parametric framework. However, an important aspect is that the procedure works best if the interpolants generated are independent of the parameter valuations. In our implementation, the theorem prover was tuned to produce such interpolants.

The parametric WCET estimation algorithm. The parametric WCET estimation algorithm follows Algorithm 2 with the major difference being the feasibility checking of worst-case paths. As parametric WCET estimates are disjunctive, we generate worst-case paths for each disjunct by choosing appropriate parameter valuations and use them for feasibility analysis and refinement as in Algorithm 2.

8.1 Parametric Maximum-Weight Length-Constrained Paths

For evaluating the WCET of parametric ASTs, we proceed recursively as in the non-parametric case. At each level, we reduce the problem to the parametric version of the length-constrained maximum-weight paths in a graph. Let $\langle V, E\rangle$, v_{in}, and v_{out} be as in the non-parametric case. Given a cost function $cost : V \rightarrow \mathcal{E}(P)$, a global bound expression $g_{max} \in \mathcal{A}(P)$, and local bound functions $l_{min}, l_{max} : V \rightarrow \mathcal{A}(P)$, the *parametric length-constrained maximum-weight path problem* asks for an expression $W \in \mathcal{E}(P)$ such that for every valuation of parameters $val(P)$, we have that $W[val(P)]$ is equal to the cost of the length-constrained maximum-weight path in the graph where $cost$, g_{max}, l_{min}, and l_{max} have been instantiated with $val(P)$.

Restrictions. The problem is hard even in the case where l_{min}, l_{max} and g_{max} range over polynomial expressions. Hence, we place restrictions on the expressions and assume that g_{max}, l_{min}, and l_{max} are all linear expressions over a single parameter. Further, we present our techniques for the case where $cost(v)$ is a numeric value instead of a disjunctive expression. The algorithm where $cost(\cdot)$

yields disjunctive expressions is similar with all max and + operations over integers being replaced by max and + operations over $\mathcal{E}(P)$. Note that restricting l_{min}, l_{max}, and g_{max} to expressions in one parameter does not restrict the CFG and the AST to one parameter—multiple parameters may appear in different nodes of the AST. Before we present our algorithm for the parametric length-constrained maximum-weight path problem, we need the following lemmata.

Lemma 1 (One non-extremal node). *For every parametric length-constrained maximum-weight problem and val(P), there is an optimal path* $\pi = v_0^{k_0} v_1^{k_1} \dots v_n^{k_n}$ *such that* $l_{min}(v_i) < k_i < l_{max}(v_i)$ *for at most one* i.

The lemma holds as for any path having two non-extremal nodes (say v_{i_1} and v_{i_2} with $cost(v_{i_1}) \geq cost(v_{i_2})$), we can build another path of equal or greater weight where v_{i_1} is taken more number of times and v_{i_2} fewer times.

While the previous lemma bounds the number of repetitions of self-loops we need to consider, the next one does the same for other cycles. The cycle decomposition of a path π is given by $\langle \sigma, (L_0, n_0), (L_1, n_1) \dots, (L_k, n_k) \rangle$ where: (a) σ is a simple path from v_{in} to v_{out}; (b) each L_i is a simple cycle; (c) together, the multi-set of visited nodes in σ and the cycles L_i's each taken n_i times is the same as the visited nodes in π. Note that L_i's are not self-loops and that the classification "simple" does not take into account self-loops. Every path has a cycle decomposition and further, for every cycle decomposition where the simple path and simple cycles are connected, there is a path for which it is a cycle decomposition. In any worst-case path, the *heaviest cycle* is taken most often.

Lemma 2 (One heavy loop). *For every parametric length-constrained maximum-weight problem and val(P), there exists an optimal path* π *with cycle decomposition* $\langle \sigma, (L_0, n_0), (L_1, n_1), \dots, (L_k, n_k) \rangle$ *such that for all* $i > 0$*: (a)* $cost(L_0)/|L_0| \geq cost(L_i)/|L_i|$*; and (b)* $n_i|L_i| < lcm(|L_0|, |L_i|)$.

The algorithm. We describe the algorithm for the restricted version of the parametric length-constrained maximum weight problem. Intuitively, the algorithm considers cycle decompositions $\langle \sigma, (L_0, n_0), (L_1, n_1), \dots (L_k, n_k) \rangle$ where n_0 is a linear expression in $\mathcal{A}(P)$, and each $n_i < lcm(|L_0|, |L_i|)/|L_i|$ is an integer for $i > 0$, and a non-extremal node v and builds the disjunctive expression $\{cond \mapsto wt, \neg cond \mapsto 0\}$ where *cond* and *wt* are explained below. The solution is the maximum of such disjunctive expressions. Note that σ and L_i's can be restricted to sequences where vertices only occur either l_{max} or l_{min} times; and further, it can be assumed that $|L_0|$ is not a parametric expression, but an integer. The expression *wt* is the expression over the parameters P obtained as the sum of weights in the guessed path, i.e., $cost(\sigma) + \sum_i n_i \cdot cost(L_i)$. The condition *cond* expresses that $\langle \sigma, (L_0, n_0), (L_1, n_1), \dots (L_k, n_k) \rangle$ is a valid cycle decomposition that respects Lemma 2, and that the total length is less than g_{max}. The correctness of the algorithm depends on the above lemmata and the fact that there are only a finite number of such parametric cycle decompositions.

Theorem 3. *The restricted parametric length-constrained maximum-weight problem can be solved in* EXPSPACE *in the size of the inputs on a computing model where operations on disjunctive expressions have constant cost.*

Practical cases. As in the non-parametric case, we provide efficient algorithms for the most commonly occurring practical cases.

- *No global bounds.* If the graph has $g_{max} = \infty$, we can use the standard dynamic programming longest-path algorithm for DAGs with the integer max and + operations being replaced by max and + operations over $\mathcal{E}(P)$.
- *Progressive phases.* In the progressive phases case, the same maximum-weight longest-path algorithm can be used with the modification of accumulating the length of the path along with the weight, and then constraining the final result with the condition that the length is at most the global bound.

9 Experimental Evaluation

We implemented our approach in a tool called IBART. It takes C programs (with no procedure calls) as input, and returns a parametric WCET estimate.

```
n:=0;
while(n < iters)
 if(health==round0)
  HighVoltageCurrent(health)
  UpdatePeriod(temp, 5)
  if(hit_trigger_flag==0)
   ResetPeakDetector()
 if(health==round1)
     ...
 if(health==round4)
   LowVoltageCurrent()
 ...
 if(health!=0)
   health--
 else
   health=9
 n++
```

Fig. 7. from `ex2` from Debie suite.

Low-level analysis. IBART analyzes WCET for Infineon C167 and LPC2138 ARM7 processors, using CalcWcet167 [12] and owcet [2] respectively, as low-level analyzers to compute basic block execution costs. These costs are then mapped from the binary to the source level and used in our analysis. Thus, IBART is platform-aware; it can be easily extended to other architectures by supplying the architecture-dependent basic block execution times on source level. However, we note that due to this approach, we cannot refine the WCET estimates in the case the infeasibility is due to caching or pipeline effects between basic blocks, and are beyond what is analyzable with the low-level tools. This is an orthogonal issue, as we focus on better estimates of WCET by having a better approximation of feasible paths. However, in the future, we plan to alleviate this issue by using our framework to automatically discovering predicates about caches and pipelines, and by performing the loop unrolling on-demand to aid the low-level analyzers.

Dependent loops. We implement a slight extension of the parametric algorithm presented in Section 8 to handle dependent loops. Consider two loops `for(i=0;i<n;i++) for(j=0;j<i;j++){...}`. The worst-case cost of an outer loop iteration is $n \cdot k$ (where k is the cost of an iteration of the inner loop). Using this worst-case cost, we get that the worst-case cost of the outer loop is $n^2 \cdot k$. However, the inner-loop costs only $k \cdot i$ in the i^{th} iteration. In this case, we incorporate the precise cost of the child node while computing the cost of the parent node, i.e., the more precise estimate for the outer loop is $\sum_{i=0}^{n-1} i \cdot k$. Intuitively, when the child node costs a polynomial (say $p(i)$) in the i^{th} repetition,

we can compute the more precise estimate as $\sum_{i=1}^{n} p(i)$. Note that this extension is equivalent to considering the loop counter of the outer loop a parameter while evaluating the inner loop.

Benchmarks. We evaluated IBART on 10 examples (examples 2 to 11 in Table 2) taken from WCET benchmark suites and open-source linear algebra packages. Of the 10 examples, 3 are small functions with less than 30 lines of code; the remaining 7 have between 34 and 109 lines of code. While small, the examples were chosen to be challenging for WCET analysis, due to two features: (a) branching statements within loops, leading to iterations with different costs, and (b) nested loops, whose inner loops linearly depend on the outer loops.

WCET benchmark suites. We used the Debie and the Mälardalen benchmark suite from the WCET community [18], which are commonly used for evaluating WCET tools. We analyzed one larger example (109 lines) from the Debie examples (`ex2` in Table 2) and 4 programs from the Mälardalen suite. The parametric timing behavior of these examples comes from the presence of symbolic loop bounds. An excerpt from the Debie example is shown in Figure 7.

Note that in Figure 7, different paths in the loop have different execution times. Moreover, every conditional branch is revisited at every tenth iteration of the loop. Computing the WCET of the program by taking the most expensive conditional branch at every loop iteration would thus yield a pessimistic overestimate of the actual WCET. Our approach derives a tight parametric WCET by identifying the set of feasible program paths at each loop iteration.

Linear algebra packages. We used 5 examples from the open-source Java linear algebra libraries JAMA and Jampack. These packages provide user-level classes for matrix operations including inverse calculation (`ex7`), SVD (`ex8`), triangularization (`ex9`), and eigenvalue decomposition (`ex10`, `ex11`) of matrices. We manually translated them to C. These benchmarks contain nested loops, often with conditionals, and with inner loops linearly depending on outer loops.

Results. We evaluated IBART for parametric WCET computation, and compared IBART with state-of-the-art WCET analyzers. All experiments were run on a 2.2 GHz Intel Core i7 CPU with 8 GB RAM and took less than 20 seconds.

IBART results. Our results are summarized in Table 2. Column 3 shows the parametric WCET (in the solution language of Section 8) calculated by IBART with basic block execution times provided by CalcWCET167. In all cases, the number of refinements needed was between 2 and 6.

Comparison with WCET tools. We compared the precision of IBART to r-TuBound [13] supporting the Infineon C167 processor and OTAWA, supporting the LPC2138 processor. Note that r-TuBound and OTAWA can only report a single numeric value as a WCET estimate. Therefore, to allow a fair comparison of the WCET results we use the basic block execution times of the respective low-level analyzer in IBART, and instantiate the symbolic parameters in the flow facts with concrete values when analyzing the WCET with r-TuBound, respectively OTAWA. To this end, parameters were supplied to OTAWA by means of (high-level) input annotations. r-TuBound does not support input annotations, therefore parameters were encoded directly in the ILP, if possible.

Ex	Source/File	Parametric WCET (C167)
ex1	Section 2, Figure 3	$n \leq 5 \mapsto 24940$, $n \geq 6 \mapsto 5040 + 2800\lfloor n/2 \rfloor + 1900n$
ex2	Debie/ health	$n \leq 0 \mapsto 2620$, $n > 0 \mapsto 2620 + \lfloor n/10 \rfloor 59100 + (n\%10) * 6800$
ex3	Mälardalen/ adpcm	$dlt \neq 0 \mapsto 4180 + 5060n$, $dlt = 0 \mapsto 4260 + 2500n$
ex4	Mälardalen/ crc	$jrev > 0 \mapsto 5560 + 3860len$, $jrev \leq 0 \mapsto 4320 + 3380len$
ex5	Mälardalen/ crc	$len \geq -1 \wedge init = 0 \mapsto 7800 + 3840len$, $init \neq 0 \mapsto 3060$
ex6	Mälardalen/ lcdnum	$n \geq 0 \mapsto 1740 + 2460n$, $n < 0 \mapsto 1740$
ex7	Jampack/ Inv	$2 > n \wedge n \geq 0 \mapsto 13540 + 6420n$, $0 > n \mapsto 13380$, $n > 2 \mapsto 13380 - 3100n + 9480n^2$
ex8	Jampack/ Zsvd	$nc \leq nr \wedge r \geq c \mapsto 3840$, $nc > nr \wedge c > r > b \mapsto 18260 + 18820(r - b)$, $nc \leq nr \wedge c < r \mapsto 3920$ ...
ex9	JAMA/ Cholesky-Decomposition	$1 = n \mapsto 44880$, $1 > n \mapsto 14260$, $n > 1 \mapsto 14260 + 15447n + 13419n^2 + 1754n^3$
ex10	JAMA/ Eigenvalue-Decomposition	$1 > n \mapsto 11780$, $n \geq 1 \mapsto -11784 + 17602n - 5146n^2 + 11108n^3$
ex11	Jampack / Eigenvalue-Decomposition	$n < 0 \mapsto 25460$, $n \geq 0 \mapsto 25460 + 28400n + 9500n^2 + 11220n^3$

Table 2. Parametric WCET computation for the C167 architecture

Our results, summarized in Table 3, show that IBART provides significantly better WCET estimates than the respective framework. For larger values of parameters, the difference increases rapidly. This is because r-Tubound and OTAWA over-approximate each iteration much more than IBART; so if the number of iterations increases, the difference grows. Column 2 lists the values of parameters. Columns 3 and 4 show the WCET computed by IBART and r-TuBound for the C167 architecture, while columns 5 and 6 show the WCET computed by IBART and OTAWA for the LPC2138 architecture. Note that Columns 3 and 4 are in nanoseconds, while Columns 5 and 6 are in cycles.

IBART reports a parametric formula instead of a single number. Instantiating with concrete parameter values (see Table 3), often gives a tighter WCET estimate. In cases when the WCET estimate of IBART overlaps with the estimate of r-TuBound or OTAWA, IBART usually allows to infer tighter estimates for specific parameter configurations. For example, for ex4, in both architectures the estimates are identical when $jrev < 0$. IBART automatically discovers the predicate $jrev \geq 0$ to specialize cases where a tighter estimate is possible. On the other hand, this information cannot be used in r-TuBound, while OTAWA fails to exploit the supplied input-annotations leading to over-estimation.

10 Related Work

We briefly summarize the large body of related work here.

Segment abstraction. Segment abstraction was introduced in [8] and was shown to subsume a large class of program analysis techniques. In [6], it was extended

Ex	Parameter assignments	IBART	r-TuBound	IBART	OTAWA
		C167 (ns)		LPC2138 (cycles)	
ex1	$n = 5$	**22,300**	26,060	393	393
	$n = 100$	**388,040**	48,0160	**6,888**	8,338
ex2	$n = 10$	**62,020**	124,920	**2,115**	2,258
	$n = 50$	**298,420**	612,920	**10,435**	11,098
	$n = 200$	**1184,920**	2442,920	**41,635**	44,248
ex3	$n = 6, dlt = 0$	**19,260**	345,040	**246**	295
	$n = 0, dlt \neq 0$	**4,180**	4,260	56	56
	$n = 0, dlt = 0$	4,260	4,260	**55**	56
ex4	$len = 5, jrev < 0$	24,860	24,860	520	520
	$len = 5, jrev \geq 0$	**21,220**	24,860	**447**	520
	$len = 0, jrev \geq 0$	**4,320**	5,560	**112**	140
ex5	$init = 0, len = 255$	987,000	987,000	18,534	18,534
	$init = 1, len = 255$	**2,920**	987,000	**102**	18,534
ex6	$n = 10$	**21,660**	26,340	**349**	404
	$n = 5$	13,960	13,960	214	214
ex7	$n = 5$	**243,880**	519,280	**2,123**	2,538
	$n = 1$	**19,760**	19,960	**263**	346
ex8	$r = 4, c = 5,$ $nr < nc, b = 0$	**93,540**	100,480	**837**	880
ex9	$n = 5$	**646,220**	1545,760	**2,902**	6,993
	$n = 1$	44,880	44,880	**317**	357
ex10	$n = 5$	**1335,920**	2606,180	**6,059**	23,089
	$n = 0$	11,620	11,620	**209**	445
ex11	$n = 5$	1799,620	1799,620	**6,371**	22,946
	$n = 1$	74,580	74,580	582	582

Table 3. WCET comparisons for the C167 and the LPC2138.

to quantitative properties. This paper brings a key contribution: a systematic way for computing WCET using computation of local ILPs at each node of AST, instead of one large global ILP. Furthermore, (a) we adapt segment abstraction for the timing analysis by using global and local bound functions; (b) we refine using a novel interpolation based technique (c) we propose the LC-IPET encoding, and (d) the parametric bounds are novel.

Asymptotic analysis. Computing bounds automatically was explored (e.g., [9]). Our work differs both conceptually and methodologically from these as we compute the worst-case execution time rather than asymptotic complexity, and we infer predicates using interpolation rather than using template-based methods.

Static WCET analysis. Most state-of-the-art static WCET tools, see e.g. [13, 2], compute a constant WCET, requiring numeric upper bounds for all loops. Our parametric WCET is computed only once and replacing parameters with their values yields the precise WCET for each set of concrete values, without rerunning the WCET analysis as in [13, 2]. These and other WCET tools use ILP as the basic data structure. Our basic data structure is ASTs, which leads to more efficient, and more precise, algorithms. All these approaches to WCET estimation (including ours) are dependent on low-level analysis developed for modelling timing related features of architectures. For a survey of techniques in this area, see [18].

Parametric WCET estimates. Parametric WCET calculation is also described in [5, 1, 10], where polyhedra-based abstract interpretation is used to derive integer constraints on program executions. These constraints are solved as parametric integer linear programming problem, and a parametric WCET is obtained.

In [10], various heuristics are applied in order to approximate program paths by small linear expressions over execution frequencies of program blocks. In [4], the authors describe an efficient, but approximate method of solving parametric constraints—in contrast, our approach solves parametric constraints exactly. Compared to [5, 10], our segment abstraction lets us to reason about the WCET as a property of a sequence of instructions rather than a state property.

Refinement for WCET. We are aware of two recent works for refining WCET estimates [14, 3]. WCET squeezing [14] is based on learning ILP constraints from infeasible paths—the constraints learned are based purely on syntactic methods. The recent work [3] also automatically discovers additional ILP constraints using minimal unsatisfiable cores of infeasible paths. These approaches have the disadvantage of being susceptible to requiring many refinements to eliminate related infeasible paths that can be eliminated with one segment predicate.

Symbolic Execution based WCET analysis. The most relevant work to our approach is [7]. In [7], the authors use symbolic execution to explore the program as a tree, and find and merge segments that have similar timing properties (for example, different loop iterations). More precisely, by merging segments generated by symbolic execution, one can obtain trees equivalent to ASTs (generated in our approach by splitting nodes in the initial tree). However, the cost of generating these objects can be significantly different. The key difference is that parts of the tree that do not occur in the worst-case path can be analyzed quickly in our case (we do not need to split nodes when even the over-approximation shows that it is not part of WCET path). On the other hand, in the symbolic execution case, even the parts of the tree not occurring in the worst-case path have to be explicitly explored and merged. However, if all paths have similar execution times, symbolic execution has an advantage as in our case the tree will eventually be split up completely. However, many of the advantages such as having locality of constraints also apply in the case of [7].

11 Conclusion

Our approach to WCET analysis is based on the hypothesis that segment abstraction is the ideal framework for WCET estimation. This is intuitively clear, as WCET is a property that accumulates over segments, i.e., sequences of instructions, and is not a state property, and therefor not being fully amenable to standard state-based abstractions. Our approach based on abstract segment trees provides two clear advantages. First, ASTs allow us to decompose the problem into multiple smaller ones. In particular, it allows us to decompose the integer linear program (ILP) for path analysis to multiple smaller integer linear programs. Second, it allows us to compute more precise refinements compared to existing techniques. This is because ASTs can encode more expressive constraints than ILPs.

A possible direction for future work is to explore is to extend our techniques to additional cost models—including aspects such as cache-persistence [18] (where only the first access to a location is a cache-miss). In fact, the segment abstraction

is rich enough to incorporate hard constraints such as scoped persistence [11]. Further, segment abstraction and refinement can be used to refine the low-level timing analysis—segment abstractions can be used to drive CFG transformations to enhance the precision of low-level timing analysis. Extending our method with interpolation in first-order theories, e.g., the theory of arrays, could yield transition predicates over data structures. Finally, estimating WCET could be used in synthesis of optimal programs.

References

1. S. Altmeyer, C. Humbert, B. Lisper, and R. Wilhelm. Parametric timing analysis for complex architectures. In *RTCSA '08*, pages 367–376, 2008.
2. C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: an Open Toolbox for Adaptive WCET Analysis. In *Proc. of SEUS*, 2010.
3. B. Blackham, M. Liffiton, and G. Heiser. Trickle: automated infeasible path detection using all minimal unsatisfiable subsets. In *RTAS*, pages 169–178, 2014.
4. S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric WCET calculation. *Journal of Systems Architecture*, 57(6):614 – 624, 2011.
5. S. Bygde and B. Lisper. Towards an Automatic Parametric WCET Analysis. In *Proc. of WCET*, 2008.
6. P. Černý, T. Henzinger, and A. Radhakrishna. Quantitative Abstraction Refinement. In *Proc. of POPL*, pages 115–128, 2013.
7. D. Chu and J. Jaffar. Symbolic simulation on complicated loops for WCET path analysis. In *EMSOFT*, pages 319–328, 2011.
8. P. Cousot and R. Cousot. An abstract interpretation framework for termination. In *Proc. of POPL*, pages 245–258, 2012.
9. S. Gulwani and F. Zuleger. The reachability-bound problem. In *Proc. of PLDI*, pages 292–304, 2010.
10. B. Huber, D. Prokesch, and P. Puschner. A Formal Framework for Precise Parametric WCET Formulas. In *Proc. of WCET*, pages 91–102, 2012.
11. B.K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *RTAS 2011*, pages 203–212. IEEE, 2011.
12. R. Kirner. The WCET Analysis Tool CalcWcet167. In *Proc. of ISoLA*, pages 158–172, 2012.
13. J. Knoop, L. Kovács, and J. Zwirchmayr. r-TuBound: Loop Bounds for WCET Analysis. In *Proc. of LPAR*, pages 435–444, 2012.
14. J. Knoop, L. Kovács, and J. Zwirchmayr. WCET squeezing: on-demand feasibility refinement for proven precise WCET-bounds. In *RTNS*, pages 161–170, 2013.
15. Y. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, pages 456–461, 1995.
16. P. Puschner and A. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. *Real-Time Systems*, 13(1):67–91, 1997.
17. R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, pages 703–719, 2011.
18. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.