Parametric WCET as a function of procedure arguments: analysis and applications

Sandro Grebant, Clément Ballabriga, Julien Forget, Giuseppe Lipari

Parametric WCET as a function of procedure arguments: analysis and applications*

Sandro Grebant, Clément Ballabriga, Julien Forget†, Giuseppe Lipari
firstname.lastname@univ-lille.fr
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL
F-59000, Lille
France

Abstract

Traditional Worst-Case Execution Time analysis derives an upper-bound to the execution time of a program for any possible combination of its software and hardware parameters. In comparison, Parametric Worst-Case Execution Time analysis derives a WCET formula that depends on the parameters. The formula can then be instantiated for some given parameter values, to produce a WCET that is specific to those values, and thus usually tighter.

In this work, we present a technique that, by static analysis of binary code, automatically produces a formula that represents the WCET of a procedure as a function of its arguments. The formula captures how the control-flow, and thus the WCET, depends on the arguments that appear in branch conditions (loop conditions and if-then-else conditions).

We detail two applications of this technique. In our first and main application, we show that WCET formulas can be instantiated during the parametric analysis itself, to make it modular. The code of a procedure is analysed only once, and the WCET of a call to that procedure is obtained by instantiating the corresponding formula with the parameter values passed at the call site.

Second, we show that WCET formulas can be instantiated at runtime, to implement adaptive real-time systems. We discuss how this can be leveraged to: 1) implement real-time systems that follow the recently proposed semi-clairvoyant mixed-criticality scheduling approach; 2) implement adaptive control-command laws.

Keywords – Worst-Case Execution Time analysis, real-time systems, abstract interpretation

1 Introduction

In real-time safety critical systems, it is of paramount importance to guarantee that computation is performed within certain time bounds. Avionics, aerospace, or autonomous car systems, are all examples of real-time safety critical systems. To guarantee that real-time constraints are satisfied, the developer needs first to compute bounds on the execution time of each task of the system, and then to guarantee that all tasks are scheduled in such a way that they will always meet their deadlines.

The execution time of a task often exhibits a large variability related to software parameters (e.g. program inputs) or hardware parameters (e.g. cache state). Static Worst-Case Execution

*This work is partially funded by the French National Research Agency, Sywext project (ANR-19-CE25-0002).
†Corresponding author

Published journal article: https://doi.org/10.1016/j.sysarc.2024.103086

Time (WCET) analysis aims at providing a *safe* upper-bound to the execution time of a task for any possible combination of the software and hardware parameters. Ideally, the estimated WCET must also be *tight* (close to the actual WCET) to keep resource over-provisioning to a minimum.

Traditional WCET analysis produces a constant numeric upper-bound to the WCET, that bounds the execution of the task for any possible combination of the hardware and software parameters. Instead, *parametric WCET analysis* produces a *formula* that represents the WCET as a function of the parameters. The formula can later be *instantiated* with actual parameter values to provide an upper-bound to the execution time for those parameter values. The instantiated WCET is usually tighter than in the traditional approach, as it considers a lower number of possible execution scenarios[1].

Formula instantiation can be performed either off-line (before system execution) or on-line (during system execution). We propose to use formula instantiation during the WCET analysis itself to make it modular. For each procedure, the analysis produces a formula that represents the WCET as a function of the procedure arguments. The WCET for a procedure call is then computed by instantiating the formula with the parameter values at the call site. This reduces the complexity of the analysis, thus enabling to analyse more complex programs.

On-line formula instantiation can be used to implement an *adaptive* real-time system. A real-time task typically releases periodically new jobs to execute. The task formula can be instantiated at job release to determine the job WCET considering the current parameter values. The system behaviour can then be adapted depending on the instantiated WCET. This can be leveraged to implement systems that follow the model considered in semi-clairvoyant scheduling for mixed-criticality systems [1, 15, 11], or to implement adaptive control-command laws.

1.1 Motivating example

We motivate our work with the example of Figure 1, previously presented in [27]. This procedure is part of an implementation of the G.723 speech encoding standard, taken verbatim from TACLeBench [24].

The G.723 codec is based on Adaptive Differential Pulse Code Modulation (ADPCM). During the signal encoding, each sample `sl` of the input signal is compared against a value `se` predicted based on previous samples. The difference `d=sl-se` is quantized to a logarithmic factor represented by argument `dqln`. The procedure reconstructs the difference signal based on that value (it also takes the `sign` of the value and the adaptive quantization step `y` as arguments). If the difference `dqln` is low[2] compared to the quantization step `y` (line 6), the reconstructed difference is set to 0 (line 7[3]). Otherwise (`else` branch), the procedure computes the antilog of `dql`, assuming a fixed-point signed representation of the real value `dqln`.

Our analysis, applied to the corresponding assembly code, detects that the branching instruction corresponding to source line 6 depends on two procedure arguments (arg_2 a.k.a. `dqln`, and arg_3 a.k.a. `y`), and infers the branch conditions $4 \times arg_2 + arg_3 \leq -1$ for the `then` case and $4 \times arg_2 + arg_3 \geq 0$ for the `else` case. Then, it produces a WCET formula that depends on those branch conditions.

Let us emphasize that the WCET variations are neither due to aberrant values, nor predictable before runtime, as they depend on the shape of the input signal.

[1] Reasoning with parametric values may render some auxiliary analyses more pessimistic (e.g. infeasible paths analysis or cache analysis, as discussed in [5]), but experiments show that the pros usually outweigh the cons.

[2] Addition on logarithmic values (`dqln` and `y`) amounts to multiplication.

[3] `dql` is signed, in two's complement, which explains the test at line 7.

```c
int reconstruct(int sign, int dqln, int y)
{
  short dql, dex, dqt, dq;

  dql = dqln + ( y >> 2 );
  if ( dql < 0 )
    return ( ( sign ) ? -0x8000 : 0 );
  else {
    dex = ( dql >> 7 ) & 15;
    dqt = 128 + ( dql & 127 );
    dq = ( dqt << 7 ) >> ( 14 - dex );
    return ( ( sign ) ? ( dq - 0x8000 ) : dq );
  }
}
```

Figure 1: Speech encoding, reconstructing the difference signal

This example has been chosen for illustrative purposes thanks to its simplicity. It shows that we can characterize the impact of argument values on the WCET of a procedure. The variation of WCET for such small function is a few tens of processor cycles, hence it is not useful to instantiate its WCET formula on-line: the evaluation function takes almost as much time as the potential maximum variability (see line *g723_enc_reconstruct* in Table 4 in Section 8.1.5). However, other more complex functions show a much larger variability and computing the formula on-line makes sense for those functions (see Section 8.1 for a complete set of experiments).

We underline the fact that, although procedure `reconstruct` is only a part of the complete encoder program, it is representative of many signal processing algorithms, which are pervasive in real-time systems, and whose computations and WCET vary depending on the input signal.

1.2 Contributions

In this paper we first present a parametric WCET analysis, which analyses the binary code of a procedure to produce a formula that represents the WCET of the procedure as a function of its arguments. Then, we detail how formula instantiation can be used during the WCET analysis itself to make it modular. We also illustrate how on-line formula instantiation can be leveraged to implement adaptive real-time systems.

Our approach is based on two of our previous works, on symbolic WCET computation [10], and on abstract interpretation of binary code [9]. In a nutshell, *symbolic WCET computation* starts from the Control-Flow Graph of the program (CFG), translates it into a Control-Flow Tree (CFT), transforms the CFT into a WCET formula, and finally simplifies the formula to reduce its size.

Although our analysis relies on foundations presented in the two papers mentioned above, many novel contributions and extensions were necessary to make it work in a coherent and automatic way. These extensions are detailed in this paper.

In the previous work, the programmer needed to manually specify which elements of the program were parameters, and it was not possible to express conditional expression. In this paper, we start by automatically identifying function arguments as parameters (Section 5.1). Then, we devise an analysis that infers *input conditions*, that is to say predicates on procedure arguments that serve as branch conditions, either in conditional statements or in loops. This analysis extends the relational abstract interpretation of binary code proposed in [9] and is presented in Section 5.2. Also, we introduce a new type of node in the CFT to represent branches subject to input conditions. This is presented in Section 6.1. Second, in Section 6.2 we extend the symbolic computation to support formulae where the input conditions appear as parameters. Furthermore, in Section 6.3 we propose extensive simplification procedures to reduce the size of the formulae. We also provide a compiler that generates C code, which is optimized to have low WCET, to evaluate the formula on-line (Section 6.4).

We detail a modular extension of the symbolic analysis in Section 7. This extension concerns both the abstract interpretation and the symbolic WCET computation steps.

Our evaluation consists of two parts. In Section 8.2 we illustrate how on-line formula instantiation can be leveraged to implement adaptive real-time systems. Based on experiments on TACLeBench, we demonstrate in Section 8.1 that our approach is **adaptive**, **embeddable**, and also **automated**:

- *Adaptivity*: the instantiated WCET can vary significantly when we take into account the value of the procedure arguments. Our approach detects *dynamically* infeasible paths, that is to say paths that are infeasible because of the current procedure argument values.
- *Embeddability*: the size of the WCET formula and the instantiation time are kept to a minimum, so as to enable on-line execution.
- *Automation*: our approach takes the binary code of a procedure as input and produces a WCET formula dependent on the procedure arguments as output, without requiring assistance from the programmer.

This paper is an extended version of [27]. The extensions include:

- The modular WCET analysis extension and its evaluation;
- The application of our method to the implementation of adaptive real-time systems;
- A more in-depth presentation of the background ([10], [9]), so as to make the paper more self-contained;
- A more detailed presentation of the inference of input conditions.

2 Related work

The most widely used WCET analysis technique is the Implicit Path Enumeration Technique (IPET) [32]. It takes as input a representation of the compiled program in the form of a graph (the Control-Flow Graph – CFG), and explores it to build an Integer Linear Programming (ILP) problem. The graph structure and the hardware features (pipeline, cache, etc.) are encoded by linear constraints, and the solution of the problem is a numerical upper bound to the execution time of the program. An extensive survey on WCET and IPET is available in [48].

Symbolic techniques have been considered in WCET analysis for different purposes. In [12, 13, 18], authors use symbolic techniques to speed up the analysis. In [34], symbolic analysis is used to trade off analysis time against tightness. Wilhelm et al. [49] model the effect of pipelines on the WCET using symbolic states. Reineke et al. [38] demonstrate how to represent various architectural effects, e.g. processor frequency, memory latencies or memory sizes, using symbolic WCET analysis. However, even though these approaches are symbolic, their *results* are not parametric.

The problem of computing WCET formulae that depend on various parameters has been studied before. Approaches that rely on source code analysis have been proposed. In [46], authors proposed a technique that produces a parametric formula with loop bounds as parameters. Coffman et al. [19] extended this approach so that it can compute the maximum between several parametric paths at runtime. The technique has then been used in [35, 36] for energy-aware scheduling. Lisper proposed a technique based on symbolic ILP in [33], but the symbolic ILP solver makes the approach computationally inefficient. One limitation of source code analysis is the need to account for compiler optimizations that may change the structure of the Control-Flow Graph, making the resulting WCET pessimistic.

Regarding binary-level analyses, in [5] Altmeyer et al. rely on symbolic ILP [25] to adapt IPET analysis to the parametric case. In [16], Bygde et al. propose a different non-IPET approach: the minimal propagation algorithm, which is more efficient but also less tight. Althaus et al. [3, 4] try to improve on both efficiency and tightness with a parametric path analysis.

Tree-based WCET analysis has first been considered in [37]. Later, tree-based parametric WCET analyses have been considered as a parametric alternative to IPET. Colin et al. [21] introduced a tree-based program representation dedicated to parametric WCET analysis. This representation associates two expressions to each node of the tree: a parametric WCET expression, and a frequency expression that represents the number of executions of the tree. However, authors did not consider the problem of producing such a model from a program. Ballabriga et al. [10] also proposed a tree-based parametric WCET computation approach and detailed how to produce the tree model from a CFG. Their approach can represent a wider range of hardware and software timing effects than previous tree-based WCET analyses. It supports parametric loop bounds and parametric execution blocks (blocks of code whose WCET is a parameter). However, the programmer needs to manually specify which elements of the program are parameters. One could consider using parametric execution blocks to represent parametric conditional statements, by replacing each conditional statement by a parametric execution block, where the parameter represents the WCET of the different alternatives of the conditional statement. This would, however, cause space explosion for nested conditional statements.

Our work is indirectly related to infeasible paths analysis, for which several approaches have been proposed in the WCET analysis community: using abstract interpretation [30, 44, 17], symbolic execution [28, 29], or SMT solvers [14, 40, 41]. A survey about infeasible paths analysis can be found in [23]. These works focus on detecting (and exploiting) *statically* infeasible paths, i.e. program paths that can never be executed because of some exclusive branch conditions and assignments. In comparison, our approach detects *dynamically* infeasible paths, that is to say paths that are infeasible because of the current procedure argument values.

We conclude this section with a summary of how our work compares with existing works. First, existing works mostly consider parametric loop bounds only, none considers conditional statements with parametric conditions. Our experiments show that programs containing loop bounds that depend on procedure arguments are rarer than programs containing conditional statements that depend on procedure arguments. Second, the kind of WCET formulas supported by existing works is simpler than in our work: a single parameter, or additions between a single parameter and a constant (except [33], although that approach has other limitations). This is insufficient to represent many input conditions, such as for instance that of the motivating example of Figure 1. In comparison, we support conjunctions on linear inequalities on parameters. Finally, no existing work is simultaneously adaptive, automated and embeddable.

3 Overview

We illustrate the workflow of our approach on the program of Figure 2. Starting from the binary code of function f, the analysis consists of the following steps.

```
f:                      @  int f(int n) {
  @ ...                 @    /* A */
  str r0, [fp, #-32]    @    /* A */
  @ ...                 @    /* A */
  ldr r3, [fp, #-32]    @    /* A */
  cmp r3, #10           @    if (n <= 10) /* A */
  bgt .L2               @    { /* still A */
  @ ...                 @      /* C */
  b   .L3               @    } /* C */
.L2:                    @    else {
  @ ...                 @      /* B */
.L3:                    @    }
  @ ...                 @    /* D */
  ldr r3, [fp, #-32]    @    /* D */
  cmp r3, #-1           @    if (n <= -1)  /* D */
  bgt .L4               @    { /* still D */
  @ ...                 @      /* F */
  b .L5                 @    } /* F */
.L4:                    @    else {
  @ ...                 @      /* E */
.L5:                    @    }
  @ ...                 @    /* G */
  bx lr                 @    return; /* still G */
.global  main           @  }
main:                   @  int main() {
  @...                  @    /* ... */
  ldr r0, [fp, #-8]     @    /* Setting parameters */
  bl f                  @    f(i); /* function call */
  @ ...                 @  }
```

Figure 2: Running example

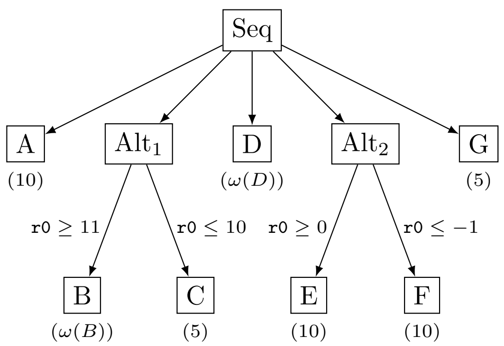


Figure 3: Control-Flow Tree for function f of Figure 2

CFG extraction the binary code is translated into a Control-Flow Graph, where nodes are basic blocks[4] and edges represent the program control-flow. We obtain a CFG with basic blocks A to G. We rely on OTAWA [8] for this step.

Hardware analysis the hardware analysis infers the WCET of each basic block. Let us assume that the resulting WCET obtained for A, E, F is 10, for C, G is 5, and that the WCET of B and D are symbolic (denoted $\omega(B)$, $\omega(D)$). We also rely on OTAWA for this step.

Inferring input conditions the abstract interpreter identifies the value stored in `r0` as an argument (a.k.a. n) of procedure f at line 1 (as per function call conventions). At line 7, it infers $\texttt{r0} \geq 11$ as the *input condition* for branching to label *L2* (a.k.a. block B) and $\texttt{r0} \leq 10$ if the program does not branch. Similarly, the input conditions $\texttt{r0} \geq 0$ and $\texttt{r0} \leq -1$ are inferred at line 16. We extend the abstract interpretation analysis of [9] to infer predicates on conditional branches and loops which depend on function arguments (see Section 5).

CFT with symbolic input conditions The CFG is translated into the Control-Flow Tree (CFT) depicted in Figure 3. It consists of a sequence (the root node *Seq*) of basic blocks (A, D, G) and of alternatives (Alt_1, Alt_2) between two subtrees (B or C, resp. E or F). Output edges of alternative nodes are annotated with the input conditions inferred by the abstract interpreter. We extend the CFT of [10] with a new type of alternative node to model conditional branches (see Section 6.1).

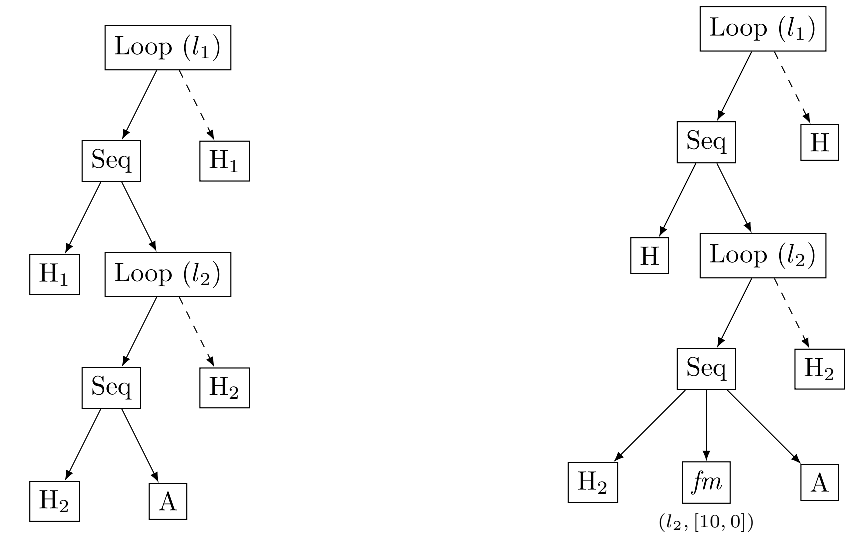
WCET formula The CFT is translated into a WCET formula. In order to support input conditions, a new $\circledast$ operator is introduced in Section 6.2. In the example, the formula contains symbolic values, therefore it cannot be reduced to a numeric value. Instead, we reduce its size using special simplification rules described in Section 6.3. After simplification, the WCET is as follows (formulas will be presented more formally in Section 4.1):

$$\begin{cases} 25 + \omega(B) + \omega(D) & \text{if } \texttt{r0} \geq 11 \\ 30 + \omega(D) & \text{otherwise} \end{cases}$$

Formula instantiation The formula is instantiated when symbolic values become known. For instance, assuming $n = 0$ (i.e. $\texttt{r0} = 0$), $\omega(B) = \omega(D) = 8$, we obtain a WCET of 38. Note that a non-parametric analysis would produce a higher WCET in this case, namely 41. The instantiated WCET reflects the fact that execution paths that include B are infeasible when $n = 0$. In Section 6.4, we present a simple compiler that, starting from a (previously simplified) formula, produces C code whose WCET is low and can be easily bounded. It can be embedded in the program to enable adaptive scheduling.

4 Background

In this section we recall the theoretical background from previous papers on which our work relies.



(a) Before transformation

(b) After transformation

Figure 4: Instruction cache transformation

4.1 Symbolic WCET computation

We first recall the main concepts of symbolic WCET computation [10]. It starts from a CFG representation of the binary program under analysis. First, it translates the CFG into a *Control-Flow Tree* (CFT). A Control-Flow Tree is similar to a Control-Flow Graph, in the sense that it also represents the possible execution paths of a program, albeit with a tree structure. Being a tree structure, the CFT is prone to recursive WCET analysis. The WCET of a CFT is expressed as a formula that follows the tree structure and in which we can easily introduce symbolic values.

The construction of the CFT from a CFG only works for CFGs that contain no irreducible loops (i.e. loops with multiples entries). However, in the general case it is possible to transform CFGs with irreducible loops by using node splitting algorithms, with relatively low overhead [31, 47]. More generally, structure-breaking instructions (such as `goto`, `break`, `continue`, or multiple `return`), cause basic block aliasing in the CFG to CFT construction. However, experiments show that the increase in the size of the CFT is relatively small compared to the size of the CFG [10].

4.1.1 Control-Flow Tree

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

Definition 4.1 (CFT)**.** *Let* $n, m \in \mathbb{N}^*$, t_1, ..., t_n, $\in \mathcal{T}^n$*. A control-flow tree* $t \in \mathcal{T}$ *is one of:*

- *Leaf*(b)*, which represents the execution of a portion* b *of the program (typically a basic block);*
- *Alt*($t_1, \ldots, t_n$)*, which represents an alternative between the execution of trees* t_1, ..., t_n*;*

[4] A *basic block* is a sequence of instructions such that if the first instruction of this sequence is executed, then the remaining instructions of that sequence are executed as well.

- *Loop*(l, t_b, n, t_e)*, which represents a loop, identified uniquely by* l*, that repeats the execution of tree* t_b*, with a maximum number of iterations* n*, and exits from the loop executing the tree* t_e*;*
- *Seq*$(t_1, \ldots, t_n)$*, which represents a sequential execution of trees* $t_1, \ldots, t_n$*.*

Example 4.1. *Figure 4a shows a CFT with a loop nested into another loop, repeating several times the code in the basic block A. Nodes* H_1 *and* H_2 *are the loop tests, repeated at the beginning of each iteration of the loop and also when exiting the loop (the dashed edge indicates the exit node).*

4.1.2 Abstract WCET

When located inside a loop, successive iterations of a CFT node can yield different WCETs. The WCET of a CFT is represented as an *abstract WCET*.

Definition 4.2 (Abstract WCET)**.** *The abstract WCET of a CFT is a pair* $a = (l, w)$*, where* l *is a loop identifier and* w *is a list of integers sorted in non-increasing order. The presence of a value* n *in* w *means that the CFT may have an execution time* n*, but only once each time* l *is entered. The smallest value of* w *is implicitly repeated indefinitely.*

Example 4.2. $(l, [10, 10, 5, 3])$ *represents the WCET of a node inside some loop* l*. The WCET of the node is at most twice 10, once 5, and 3 for all other iterations of loop* l*.*

Example 4.3. *Let us illustrate how we can represent the effect of the instruction cache. Consider the CFT of Figure 4a. Assume that a cache categorization technique [2] determines that A contains a* first-miss *cache access for loop* l_2*, i.e. the instruction is in the cache for all iterations of* l_2 *except the first one. Assume also that the cache miss penalty is 10 cycles. This is modeled in Figure 4b by a leaf fm (a virtual block) with WCET* $(l_2, [10, 0])$*.*

Representing a WCET as a list of integers instead of a single integer is a key difference compared to other existing tree-based WCET analysis techniques. This allows to account for execution times that are sensitive to the execution context and thus to significantly improve the tightness of the WCET.

The following definitions on the CFT topology are required to define operations on abstract WCET.

Definition 4.3. *Let* t *be a CFT,* t_1 *and* t_2 *be two sub-trees of* t*, where* t_1 *is a loop with identifier* l_1 *and* t_2 *is a loop with identifier* l_2*. Then:*

- *We say that loop* l_1 contains *loop* l_2*, denoted* $l_2 \sqsubseteq l_1$*, if* t_2 *is a sub-tree of* t_1*. Also, by definition, for any loop* l *we have* $l \sqsubseteq l$*;*
- $\top$ *is a fictive loop that contains any loop of* t*;*
- $\bot$ *is a fictive loop that contains no loop of* t*;*
- *Let* L *denote the set of loop identifiers of* t*. Then,* $(L \cup \{\top, \bot\}, \sqsubseteq)$ *is a lattice*[5]*;*
- $l_1 \sqcap l_2$ *denotes the greatest lower bound of* l_1 *and* l_2*, that is to say the greatest element of* $\{l : l \sqsubseteq l_1 \wedge l \sqsubseteq l_2\}$

[5] For every pair of loops l_1, l_2, their supremum is their closest common ancestor loop in the CFT (or $\top$ if no such ancestor exists). Their infinimum is their closest common descendant loop (or $\bot$ if no such descendant exists).

We now recall operations on abstract WCETs. Let $|w|$ denote the number of elements of w.

Definition 4.4. *Let $a = (l, w)$ and $a' = (l', w')$ be abstract WCETs. Then:*

- $\theta = (\top, [0])$ *is the null abstract WCET;*
- $w[n]$ *denotes the* $(n+1)^{th}$ *element of list* w *if* $n \leq |w|$*, or* $w[|w|]$ *otherwise;*
- $a \oplus a' = (l", w")$ *is such that:* $\forall i \in \mathbb{N}$*,* $w"[i] = w[i] + w'[i]$ *and* $l" = l \sqcap l'$ *(a pointwise sum on the size of the longest of the two lists* w *and* w'*);*
- $a \uplus a' = (l \sqcap l', (w \cup w') \setminus \{k : k < max(m, m'))$*, where* m*,* m' *denote respectively the smallest value of* w *and* w' *(an order-preserving list union, except that elements smaller than infinitely repeated ones are dropped);*
- $(l, w)^{n,l'}$ *represents an iteration over* (l, w)*. There are two cases (see Example 4.5 for illustration):*

$$(l, w)^{n,l'} = \begin{cases} (\top, [\sum_{i=0}^{n-1} w[i]]) & \text{if } l = l' \\ (l, \bigcup_{i \in \mathbb{N}} [\sum_{j=0}^{n-1} w[i \times n + j]]) & \text{otherwise} \end{cases}$$

Example 4.4. *We illustrate operations on abstract WCET below:*

- *Let* $w = (l, [10, 10, 5, 3])$*. Then* $w[2] = 5$*, and* $w[5] = 3$ *since* 3 *is repeated infinitely;*
- $(l, [4, 3, 2]) \oplus (l', [3, 1]) = (l \sqcap l', [4 + 3, 3 + 1, 2 + 1]) = (l \sqcap l', [7, 4, 3])$*;*
- $(l, [4, 3, 2]) \uplus (l', [3, 2, 1]) = (l \sqcap l', [4, 3, 3, 2])$*. Value* 1 *is dropped because it is smaller than the minimum WCET of the left operand;*
- $(l, [5, 4])^{4,l} = (\top, [5 + 4 + 4 + 4]) = (\top, [17])$*;*
- *Assuming* $l \neq l'$*, we have* $(l, [5, 4])^{4,l'} = (l, [5 + 4 \times 3, 4 \times 4]) = (l, [17, 16])$.

4.1.3 Computing the WCET of a control-flow tree

Using the abstract WCET representation above, the abstract WCET $\omega(t)$ of a CFT t is computed inductively on the CFT structure as follows:

$$\begin{aligned} \omega(\text{Leaf}(b)) &= \omega(b) \\ \omega(\text{Seq}(t_1, \ldots, t_n)) &= \omega(t_1) \oplus \ldots \oplus \omega(t_n) \\ \omega(\text{Alt}(t_1, \ldots, t_n)) &= \omega(t_1) \uplus \ldots \uplus \omega(t_n) \\ \omega(\text{Loop}(l, t_b, n, t_e)) &= \omega(t_b)^{n,l} \oplus \omega(t_e) \end{aligned}$$

Example 4.5. *In Figure 4b, there are two nested loops:* l_1 *and* l_2*. The first-miss leaf fm has WCET* $(l, [10, 0])$*. When* $l = l_1$ *(resp.* $l = l_2$*) a cache miss occurs each time we enter* l_1 *(resp.* l_2*). In the first case, for a complete execution of the program, the miss penalty applies only once, whereas in the second case it applies for every iteration of* l_1*, since* l_2 *is* entered *at each iteration of* l_1*. Assuming* $\omega(A) = (\top, [15])$*,* $\omega(H_1) = (\top, [5])$*,* $\omega(H_2) = (\top, [5])$*, assuming 3 iterations for each loop* l_1*,* l_2*, and denoting* t *the CFT of Figure 4b, we have:*

$$\begin{aligned} \omega(t) &= ((\top, [5]) \oplus ((\top, [5]) \oplus (l, [10, 0]) \oplus (\top[15]))^{3,l_2} \oplus (\top, [5]))^{3,l_1} \oplus (\top, [5]) \\ &= ((\top, [5]) \oplus (l, [30, 20])^{3,l_2} \oplus (\top, [5]))^{3,l_1} \oplus (\top, [5]) \end{aligned}$$

If $l = l_1$ (single miss):

$$\begin{aligned}\omega(t) &= ((\top, [5]) \oplus (l_1, [70, 60]) \oplus (\top, [5]))^{3,l_1} \oplus (\top, [5]) \\ &= (l_1, [80, 70])^{3,l_1} \oplus (\top, [5]) \\ &= (\top, [220]) \oplus (\top, [5]) \\ &= (\top, [225])\end{aligned}$$

If $l = l_2$ (three misses):

$$\begin{aligned}\omega(t) &= ((\top, [5]) \oplus (\top, [70]) \oplus (\top, [5]))^{3,l_1} \oplus (\top, [5]) \\ &= (\top, [80])^{3,l_1} \oplus (\top, [5]) \\ &= (\top, [240]) \oplus (\top, [5]) \\ &= (\top, [245])\end{aligned}$$

When some parameters of the CFT are unknown, $\omega(t)$ produces a formula containing symbolic values. The set of WCET formulae produced by $\omega(t)$ is defined by the following grammar:

$$\begin{array}{lcl} w & ::= & \mathit{constw} \mid \mathit{symb} \mid w \oplus w \mid w \uplus w \mid w^{l,it} \\ it & ::= & \mathit{int} \mid \mathit{symb} \end{array}$$

Term *constw* represents the set of constant abstract WCET values (e.g. $(l_2, [10, 0])$). Term *symb* corresponds to a set of symbols used to denote unknown values. Term *int* corresponds to the set of integer literals. Term l corresponds to the set of loop identifiers. For now, symbols can be of two kinds (this will be extended in the following sections):

- A *symbolic WCET*. For instance, $X \uplus (l, \{4\})$, where X is an unknown WCET;
- A *symbolic loop bound*. For instance, $(l, \{5, 3\})^{N,l'}$, where N is an unknown integer loop bound.

$\omega(t)$ produces a formula that is linear in the size of t. When the formula contains symbolic values, it cannot be reduced to a single operand. However, in order to decrease its size and evaluation time, the formula is reduced using simplification rules based on mathematical properties of the abstract WCET operations. For instance, $((l, \{5\}) \oplus X) \uplus ((l, \{4\}) \oplus X)$ reduces to $(l, \{5\}) \oplus X$ (see Section 6.3 for more details).

4.2 Abstract interpretation of binary code

We will now recall the main concepts of the abstract interpretation procedure of [9]. *Abstract interpretation* [22] is a general static analysis method that infers program invariants. It propagates an *abstract state* of the program, which overapproximates the set of all possible *concrete states*, until a fixpoint is reached. It is *sound*, in the sense that the invariants it infers hold for any possible concrete program state.

While abstract interpretation usually targets source code, we rely on the abstract interpretation procedure for *binary* code proposed in [9] because we want to inject the inferred invariants into our WCET analysis, which takes as input a CFG reconstructed from a binary program. We summarize the main features of this interpretation procedure below.

Table 1: Abstract states at several program points in Figure 2

line	Polyhedron	Registers	Memory
2	$p_2 = \langle\rangle$	$\mathcal{R}^\sharp_2 = \{\texttt{r0} : x_0, \texttt{fp} : x_1\}$	
4	$p_4 = \langle x_2 = x_0, x_3 = x_1 - 32\rangle$	$\mathcal{R}^\sharp_2$	$*^\sharp_4 = \{x_3 : x_2\}$
6	p_4	$\mathcal{R}^\sharp_6 = \{\texttt{r0} : x_4, \texttt{r3} : x_2, \texttt{fp} : x_1\}$	$*^\sharp_4$
8	$p_8 = p_4 \sqcap_\diamond \langle x_2 \leq 10\rangle$	$\mathcal{R}^\sharp_6$	$*^\sharp_4$
11	$p_{11} = p_4 \sqcap_\diamond \langle x_2 > 10\rangle$	$\mathcal{R}^\sharp_6$	$*^\sharp_4$

4.2.1 Polyhedra

A Polyhedron is defined as follows.

Definition 4.5 (Polyhedron)**.** *Let $\mathcal{V}$ be a set of variables and $\mathcal{C}$ be a set of linear constraints (equalities and/or inequalities) on the variables in $\mathcal{V}$. Then, $\langle c_1, \ldots, c_m\rangle$ is the polyhedron consisting in all the vectors in $\mathbb{Z}^n$ that satisfy the constraints $c_1, \ldots, c_m$, where $c_i \in \mathcal{C}$ for $1 \leq i \leq m$.*

Less formally, a polyhedron p can be viewed as the multi-dimensional geometrical shape that represents the set of possible values of the variables of $\mathcal{V}$ for which all the equalities and inequalities in $\mathcal{C}$ are satisfied. The variables of a polyhedron are also called its *dimensions* in the literature. We denote:

- $p" = p \sqcap_\diamond p'$ the polyhedron consisting of the union of the constraints of p and p';
- $vars(p)$ the set of variables of p;
- $proj(p, \{x_0, \ldots, x_n\})$ the *projection* of a polyhedron p on a subset $\{x_0, \ldots, x_n\}$ of its variables. The result is a polyhedron p' with less variables, such that every possible value $\{v_0, \ldots, v_n\}$ that satisfies the constraints of p also satisfies the constraints of p' and vice versa.

To better understand the meaning of the projection operation, it may be useful to think of geometric shapes in a 3D space: a projection on the variables (x, y) of a cube in (x, y, z) is simply the geometric projection of the cube on the plane (x, y). We will use projections in Section 5 to explain how we treat input conditions when building a symbolic formula.

4.2.2 Abstract state

Let $\mathcal{R}$ denote the set of hardware registers, $\mathcal{V}$ denote the set of polyhedra variables, $\mathcal{P}$ denote the set of polyhedra on $\mathcal{V}$. The set of abstract states is defined as $\mathcal{A} = \mathcal{P} \times (\mathcal{R} \to \mathcal{V}) \times (\mathcal{V} \to \mathcal{V})$.

Definition 4.6 (Abstract state)**.** *An abstract state $a \in \mathcal{A}$ is a tuple $(p, \mathcal{R}^\sharp, *^\sharp)$, which consists of a polyhedron p, a* register mapping *$\mathcal{R}^\sharp$ and an* address mapping *$*^\sharp$. We have $\mathcal{R}^\sharp(r) = v$ iff the variable v represents the value of the register r in p. Also, we have $*^\sharp(x_1) = x_2$ iff x_2 represents the value at the memory address represented by the variable x_1.*

Example 4.6. *In the following abstract state, register r_0 contains a positive value and address* 7872 *contains a value greater than that of r_0:*

$$(\langle x_1 \geq 0, x_2 = 7872, x_3 \geq x_1\rangle, \{r_0 : x_1\}, \{x_2 : x_3\})$$

In the rest of the paper, we use the term *data location* to refer indistinctly to registers or memory addresses. We denote $m' = m[x : y]$ the mapping such that $m'(x) = y$ and $\forall x' \neq x : m'(x') = m(x')$.

4.2.3 Interpretation procedure

A program procedure F is represented as a sequence of labeled instructions $l_0 : I_0, l_1 : I_1, \ldots, l_n : RET$, where I_k is the instruction at label l_k $(0 \leq k \leq n)$ and RET returns the control-flow to the caller[6]. The result of the interpretation $M = \mathit{interpret}(F)$ maps labels of program procedure F to their corresponding abstract states: $M[l_k]$ denotes the abstract state immediately *before* the execution of instruction I_k. For more details on procedure *interpret*, the reader is referred to [9]. An important specificity of the interpretation procedure is that the mapping between variables and data-locations can change as the interpretation progresses.

Example 4.7. *Table 1 details the abstract states at several points of the program of Figure 2. We assume that the value of n is not modified in the program. Until line 4, the register* `r0` *contains the value n, represented by variable* x_0*. Assume that* `r0` *is used to store the result of some arithmetic operation at line 4. As a result, at line 6 the value of* `r0` *does not correspond to argument n anymore, instead it is mapped to a new variable* x_4 *that corresponds to the value computed at line 4. However, note that variable* x_0 *still represents the value of the argument n in the abstract state at line 6.*

5 Inferring input conditions

In this section, we extend the abstract interpretation analysis from [9] to infer the input conditions of a binary program. We consider 32-bit ARM programs, but the analysis can easily be extended to other architectures with similar procedure call conventions.

5.1 Identifying procedure arguments

By convention [6], 32-bit ARM programs pass the first four arguments of a procedure call through registers `r0`, `r1`, `r2` and `r3`. Additional arguments are passed through the stack. In our experiments, we found that few procedures use more than four arguments. Therefore, in the following we only consider arguments passed through these registers, which we call *input registers*.

We modify the abstract interpreter so that it identifies the polyhedra variables that are associated to input registers. As the variable-to-data-location mapping evolves during the interpreter progression, a variable represents a procedure argument if and only if it is mapped to one of the input registers in the abstract state at the starting location of the procedure.

More formally, let F denote the procedure under analysis. Let $\mathcal{A}_0^\sharp$ denote the *argument mapping* of F, that associates a polyhedron variable to each argument of F. For ARM programs, we let $\mathcal{A}_0^\sharp = \{r_0 : x_0, r_1 : x_1, r_2 : x_2, r_3 : x_3\}$, where $x_0, \ldots, x_3$ are fresh variables. We slightly modify the procedure *interpret* so that it takes $\mathcal{A}_0^\sharp$ as the initial register mapping. As a consequence, for all location l_k of F, the procedure arguments are always represented by the variables of $\mathcal{A}_0^\sharp$.

Example 5.1. *In Figure 2, at line 1,* r_0 *is mapped to* x_0*. Now assume that line 4 changes the value of* `r0`*. Thus,* `r0` *is mapped to another variable:* x_4*. Nevertheless, in the subsequent abstract states (e.g. the branch at line 7) the analysis correctly identifies that* x_0 *corresponds to a procedure argument and that* x_4 *does not.*

[6] For procedures with several exit edges, the CFG reconstruction step adds a single node to which all exit edges point.

5.2 From polyhedra to input conditions

In this section, we explain how we extract input conditions from the abstract states of the program.

5.2.1 Conditional statements

When the interpreter analyses a conditional branching instruction, it adds the corresponding condition to the abstract state of the branch target; this is called *filtering*. We modify the analysis so that, whenever a filtering occurs, we project the resulting polyhedron over the variables corresponding to procedure arguments. As a result, we obtain a polyhedron corresponding to the constraints that the input registers must satisfy in order to branch to the corresponding location. These constraints consist in a conjunction of inequalities on input registers, which we call *input conditions*.

More formally, let F denote the procedure under analysis. Let $\mathcal{A}_0^\sharp$ denote the argument mapping of F. Let $M = \mathit{interpret}(F)$. Let l_k be a label of F such that $F[l_k]$ is a conditional branching instruction, and l_t be a direct successor of l_k (either l_{k+1} or the branch target). Then, the input condition for the edge (l_k, l_t) is computed as $c = \mathit{proj}(p_t, \mathit{Img}(\mathcal{A}_0^\sharp))$, where p_t is the polyhedron of $M[l_t]$ and Img denotes the image of a mapping.

Example 5.2. *In Figure 2, in the abstract state at line 8 of Table 1, the register* `r3` *is associated to the variable x_2, which is equal to x_0 (i.e. the procedure argument). Since line 8 is in the* then *block of the conditional statement, it contains the filtering condition $x_2 \leq 10$. To obtain the input condition, we project the polyhedron over the variable x_0:*

$$\mathit{proj}(\langle x_2 = x_0, x_3 = x_1 - 32, x_2 \leq 10\rangle, \{x_0\}) = \langle x_0 \leq 10\rangle$$

In the general case, the input conditions are passed unchanged to the CFT builder. There are however two particular cases:

- If the projected polyhedron has no constraints ($c = \langle\rangle$, universe polyhedron), this either means that the branch condition is not related to procedure argument, or that it can not be represented by the polyhedra formalism (e.g. it contains a disjunction). In that case, we set the input condition to *true*. This may introduce execution paths that are feasible in the CFT, while they are infeasible in the actual program. From a WCET point-of-view, this is safe, as it can only cause over-approximation (no under-approximation);
- If the projected polyhedron has unsatisfiable constraints (empty polyhedron), the branch target is dead code, then the input condition is set to *false*.

5.2.2 Loop bounds

If the branch instruction is located in a loop header, we compute a loop bound instead of a condition. This is done using a "ghost" register, that does not correspond to an actual data-location used by the program but represents the induction variable of the loop[7]. The register is set to 0 at the entry of the loop and is incremented at each loop iteration.

More formally, let F denote the procedure under analysis. Let $\mathcal{A}_0^\sharp$ denote the argument mapping of F. Let $M = \mathit{interpret}(F)$. Let l_k be a loop header (identified as such by the CFG builder), $M[l_k] = (p_k, \mathcal{R}_k^\sharp, *_k^\sharp)$, and r_g denote the ghost register of l_k. Function $\mathit{lbound}(p_k, \mathcal{A}_0^\sharp, \mathcal{R}_k^\sharp[r_g])$ computes the loop bound as follows. First, it computes $p_{k'} = \mathit{proj}(p_k, \mathcal{A}_0^\sharp \cup \{\mathcal{R}_k^\sharp[r_g]\})$. From there, two cases can occur:

[7] In a way, it represents information similar to execution counters in the IPET method

```
1  f:                        @  int f(int x){
2    @ ...                   @    // r0 contains x
3    str  r0, [fp, #-16]     @    // (fp-16) contains x
4    mov  r3, #1             @    int res = 1;
5    str  r3, [fp, #-8]      @
6    mov  r3, #0             @    int i = 0;
7    str  r3, [fp, #-12]     @
8  .L2:                      @    do{ // mov gr, #0
9    ldr  r3, [fp, #-8]      @
10   lsl  r3, r3, #1         @      res += res;
11   str  r3, [fp, #-8]      @
12   ldr  r3, [fp, #-12]     @
13   add  r3, r3, #1         @      i++;
14   str  r3, [fp, #-12]     @
15   ldr  r2, [fp, #-12]     @
16   ldr  r3, [fp, #-16]     @      // add gr, gr, #1
17   cmp  r2, r3             @    }
18   blt  .L2                @    while(i < x);
19   ldr  r3, [fp, #-8]      @
20   mov  r0, r3             @    // r0 contains res
21   @ ...                   @    return res;
22   bx   lr                 @  }
```

Figure 5: Assembly and C code of a loop

- We have $p_{k'} = \langle 0 \leq \mathcal{R}^\sharp_k[r_g], \mathcal{R}^\sharp_k[r_g] \leq e \rangle$, where e is a linear expression on variables in $\mathcal{A}^\sharp_0$. Then e is the loop bound;
- Otherwise, we are not able to compute a loop bound and it must be provided by the user.

Example 5.3. *The code of a simple f consisting of a simple loop is detailed in Figure 5. When entering the loop, the ghost register* `gr` *is initialized to 0 inside the abstract state of the analysis, similarly to a* `mov gr, #0`*, as shown in comment at line 8. Then, at the end of each iteration* `gr` *is incremented. At the end of the loop interpretation, the state of the loop contains the bound to the value of the ghost register. Thus, assuming that x_0 is the variable that corresponds to the argument x, we have: $\mathcal{R}^\sharp(gr) \leq x_0$. We simply replace $\mathcal{R}^\sharp(gr)$ with lb such that we have $lb \leq x_0$.*

6 Symbolic WCET with input conditions

In this section, we detail how we extend the symbolic WCET computation approach from [10] to support input conditions.

6.1 Control-flow Tree with input conditions

We extend the previous definition of alternative nodes so that an input condition is associated to each alternative.

Definition 6.1. *Let $(t_1, \ldots, t_n)$ be a tuple of CFTs, $(e_1, \ldots, e_n)$ be a tuple of input conditions and $1 \leq k \leq n$. The* deterministic alternative *node $Alt(e_1 \to t_1, \ldots, e_n \to t_n)$ represents an* *alternative between the execution of one tree among* $(t_1, \ldots, t_n)$*, such that the tree* t_k *can be executed only if* e_k *is true.*

We use the term *deterministic alternative* because values of parameters that appear in the input conditions determine which sub-tree of the alternative is executed. In comparison, in non-deterministic alternatives, any sub-tree of the alternative can be executed, whatever the parameter values.

Example 6.1. *Figure 3 depicts the CFT obtained for the program of Figure 2. For instance, we can see that the input condition* $\texttt{r0} \geq 11$*, whose inference was detailed in Example 5.2, appears as an input condition to execute* B *in the deterministic alternative node* Alt_1*.*

Concerning loop nodes, their definition remains unchanged, except that the loop bound n can now be a linear expression on procedure arguments.

Example 6.2. *The node* $Loop(l, t_1, 4 \times \texttt{r0} + \texttt{r1}, t_2)$ *represents a loop identified by* l*, that executes* $4 \times \texttt{r0} + \texttt{r1}$ *times the tree* t_1 *and exits by executing the tree* t_2*.*

6.2 WCET formulas with input conditions

We define a new operator $\circledast$ that multiplies a WCET by an input condition. It has higher priority than operators $\oplus$ and $\uplus$, but lower priority than the other operators. It is used to compute the WCET of an Alt node:

$$\omega(\text{Alt}(e_1 \to t_1, \ldots, e_n \to t_n)) = e_1 \circledast \omega(t_1) \uplus \ldots \uplus e_n \circledast \omega(t_n)$$

Definition 6.2. *Let* e *be an input condition and* w *be an abstract WCET.*

$$e \circledast w = \begin{cases} w & \text{if } e \text{ is true} \\ \theta & \text{otherwise} \end{cases}$$

Example 6.3. *The subtree* Alt_1 *of Figure 3 is translated into the formula* $(\texttt{r0} \geq 11) \circledast \omega(B) \uplus (\texttt{r0} \leq 10) \circledast (\top, \{5\})$*. This corresponds to* $\omega(B)$ *if* $\texttt{r0} \geq 11$*, or to* $(\top, \{5\})$ *otherwise.*

The set of WCET formulae produced by $\omega(t)$ is now defined by the following grammar:

$$\begin{array}{lll} w & ::= & constw \mid symb \mid w \oplus w \mid w \uplus w \mid ipred \circledast w \mid (w)^{l,it} \\ it & ::= & symb \mid linexp \\ linexp & ::= & int \mid reg \mid int * reg \mid linexp + linexp \\ ipred & ::= & linineq \mid linineq \wedge ipred \\ linineq & ::= & linexp \leq int \mid linexp \geq int \end{array}$$

As previously, *constw* represents constant abstract WCET values, *symb* unknown values, and *int* integer literals. The novelty compared to the previous grammar of Section 4.1.3 is the introduction of terms *linexp* and *ipred*. Term *linexp* represents integer linear expressions on register names (*reg*). Term *ipred* represents conjunctions of linear inequations on register names.

6.3 Simplifying WCET formulas

The size of the formula $\omega(t)$ is linear in the number of nodes of t. In this section, we detail simplification rules to reduce the size of WCET formulae. The simplification procedure applies simplification rules in an order that follows the classic integer arithmetic simplification strategy described in [20].

Commutativity

$$(e_k \wedge e_l) \circledast w_1 \mapsto (e_l \wedge e_k) \circledast w_1 \qquad \text{if } e_l \lhd e_k \tag{1}$$

$$e_k \circledast w_1 \oplus e_l \circledast w_2 \mapsto e_l \circledast w_2 \oplus e_k \circledast w_1 \qquad \text{if } e_l \lhd e_k \tag{2}$$

$$e_k \circledast w_1 \uplus e_l \circledast w_2 \mapsto e_l \circledast w_2 \uplus e_k \circledast w_2 \qquad \text{if } e_l \lhd e_k \tag{3}$$

Factorization

$$e_k \circledast w_1 \oplus e_l \circledast w_1 \mapsto w_1 \qquad \text{if } e_l \Leftrightarrow \neg e_k \tag{4}$$

$$e_k \circledast w_1 \uplus e_l \circledast w_1 \mapsto w_1 \qquad \text{if } e_l \Leftrightarrow \neg e_k \tag{5}$$

$$e_k \circledast w_1 \oplus e_l \circledast w_2 \mapsto e_k \circledast (w_1 \oplus w_2) \qquad \text{if } e_k \Leftrightarrow e_l \tag{6}$$

$$e_k \circledast w_1 \uplus e_l \circledast w_2 \mapsto e_k \circledast (w_1 \uplus w_2) \qquad \text{if } e_k \Leftrightarrow e_l \tag{7}$$

$$e_k \circledast w_1 \oplus (e_k \wedge e_l) \circledast w_2 \mapsto e_k \circledast (w_1 \oplus e_l \circledast w_2) \tag{8}$$

$$e_k \circledast w_1 \uplus (e_k \wedge e_l) \circledast w_2 \mapsto e_k \circledast (w_1 \uplus e_l \circledast w_2) \tag{9}$$

Multiplication

$$e_k \circledast \theta \mapsto \theta \tag{10}$$

$$e_k \circledast w_1 \mapsto \theta \qquad \text{if } e_k \Leftrightarrow \mathit{false} \tag{11}$$

$$e_k \circledast w_1 \mapsto w_1 \qquad \text{if } e_k \Leftrightarrow \mathit{true} \tag{12}$$

$$e_k \circledast (e_l \circledast w_1) \mapsto e_k \circledast w_1 \qquad \text{if } e_k \Leftrightarrow e_l \tag{13}$$

Loops

$$(e_k \circledast w_1)^{it,l} \mapsto e_k \circledast (w_1)^{it,l} \tag{14}$$

Figure 6: Rewriting rules with input conditions

6.3.1 Simplification rules

The new simplification rules for WCET formulae that contain input conditions are detailed in Figure 6. e_k and e_l are input conditions, w_1 and w_2 are abstract WCETs, l is a loop identifier and it is a loop bound. These rules are added to the rules of [10]. For each rule of the form $l \mapsto r$ we must prove that $l = r$. We illustrate the general proof principle for rule (8) below. The equivalence proofs of l and r for all these rules can be found in A.

Property 6.1. $e_k \circledast w_1 \oplus (e_k \wedge e_l) \circledast w_2 = e_k \circledast (w_1 \oplus e_l \circledast w_2)$

Proof. Case by case on the possible values of e_k and e_l. We write 0 (resp. 1) as a shorthand for *false* (resp. *true*).

1. Case: $e_k = 0$

$$0 \circledast w_1 \oplus (0 \wedge e_l) \circledast w_2 = \theta \oplus 0 \circledast w_2 = \theta$$
$$0 \circledast (w_1 \oplus e_l \circledast w_2) = \theta$$

2. Case: $e_l = 0$

$$e_k \circledast w_1 \oplus (e_k \wedge 0) \circledast w_2 = e_k \circledast w_1 \oplus 0 \circledast w_2 = e_k \circledast w_1$$
$$e_k \circledast (w_1 \oplus 0 \circledast w_2) = e_k \circledast (w_1 \oplus \theta) = e_k \circledast w_1$$

3. Case: $e_k = e_l = 1$

$$1 \circledast w_1 \oplus (1 \wedge 1) \circledast w_2 = w_1 \oplus 1 \circledast w_2 = w_1 \oplus w_2$$
$$1 \circledast (w_1 \oplus 1 \circledast w_2) = 1 \circledast (w_1 \oplus w_2) = w_1 \oplus w_2$$

□

Factorization rules require to test the equivalence of input conditions. The equivalence test is detailed in Section 6.3.2. For distributivity rules, we rely on an order relation $\lhd$ on input conditions (see Section 6.3.3 below) so that they can only be applied in one direction, to ensure termination of the simplification. Multiplication rules are direct consequences of the definition of the operator $\circledast$.

6.3.2 Testing input conditions equivalence

Checking the equivalence of an input condition to either *true* or *false* is straightforward. No simplification rule can create a new predicate that is equivalent to *true* or *false*. Therefore, we can simply check syntactically that the input condition is the predicate *true* or the predicate *false*.

In other cases, to test the equivalence of two input conditions, we first put them in *canonical form*. Then, we test the syntactic equality of the canonical forms. This equivalence test is exact if and only if the polyhedra domain used by the abstract interpretation procedure always represents equivalent constraints by syntactically identical terms. This is the case for the Parma Polyhedra Library [7], which we use in our implementation. An input condition is in canonical form iff:

1. The left-hand side of comparison operators is 0;
2. Comparison operators are either $\leq$ or $=$;
3. Terms are ordered by increasing symbol identifiers;
4. The last term is a constant.

Example 6.4. *The canonical form of input condition* $10 \geq 15 + r1 + r0$ *is* $0 \leq -r0 - r1 - 5$.

6.3.3 Termination of the simplification procedure

The orientation of each rule is such that either of the following holds: 1) r has less operands than l; 2) r has less parentheses than l; 3) input conditions in l are "smaller" than those in r according to relation $\lhd$ (defined below). Based on these properties, we can define a strict order relation $\prec$ such that we have $l \prec r$ for each rule. This ensures that the simplification procedure terminates. The ordering relation on input conditions is defined as follows:

$$\begin{aligned} e_k \lhd e_l \Leftrightarrow & (\text{lid}(e_k) < \text{lid}(e_l)) \lor \\ & (\text{lid}(e_k) = \text{lid}(e_l) \land \text{size}(e_k) < \text{size}(e_l)) \lor \\ & ((\text{conj}(e_k) = \mathit{false} \land \text{conj}(e_l) = \mathit{false}) \land \\ & (\text{lid}(e_k) = \text{lid}(e_l)) \land (\text{size}(e_k) = \text{size}(e_l)) \land \\ & (\text{linconst}(e_k) < \text{linconst}(e_l))) \end{aligned} \tag{15}$$

Where *lid* returns the lowest symbol identifier (or -1 if there is no parameter), *size* returns the number of terms in an input condition, *linconst* returns the constant (-1 for a conjunction) of the input condition and *conj* is true iff the input condition is a conjunction of input conditions.

Example 6.5. *Consider the input conditions* $0 \le \mathtt{r0} + \mathtt{r1} + 10 \land 0 \le \mathtt{r2}$. *We have:*

$$\begin{aligned} \mathit{lid}(0 \le \mathtt{r0} + \mathtt{r1} + 10 \land 0 \le \mathtt{r2}) &= 0 \\ \mathit{size}(0 \le \mathtt{r0} + \mathtt{r1} + 10 \land 0 \le \mathtt{r2}) &= 6 \\ \mathit{linconst}(0 \le \mathtt{r0} + \mathtt{r1} + 10) &= 10 \\ \mathit{conj}(0 \le \mathtt{r0} + \mathtt{r1} + 10 \land 0 \le \mathtt{r2}) &= \mathit{true} \end{aligned}$$

6.4 Formula instantiation

We compile the simplified formula into a C procedure, whose arguments correspond to the arguments of the procedure under analysis. This procedure can be executed off-line, e.g. for sensitivity analysis, or on-line, e.g. to implement an adaptive real-time system.

In order to enable aggressive optimizations by the C compiler, we produce code that respects the following rules: 1) the program is standalone, i.e. it has no library dependencies; 2) each value of a WCET list is declared as a separate variable; 3) there are as few loops and function calls as possible, and no pointers. Experiments show that the resulting executable program has very low execution time (see Section 8.1.5).

Note that since the WCET of a procedure is the worst-case for any possible execution scenario, executing the instantiation code before executing the procedure *cannot* increase the WCET of the procedure.

7 Modular WCET analysis of pure functions

In this section, we present an extension of our approach, a modular analysis that analyzes each procedure independently. This extension is currently limited to pure functions, that is to say functions without side-effects. Supporting side-effects, that is to say pointers, is significantly more challenging due to possible pointer aliasing in the procedure arguments.

7.1 Modular abstract interpretation

In our previous abstract interpretation analysis [9], each procedure call caused an analysis of the callee within the context of the caller (virtual inlining), possibly leading to several analyses of

Algorithm 1 Summary construction

function CONSTRUCTSUMMARY(P)
- $\mathcal{A}^\sharp \leftarrow \{r_0 : x_0, r_1 : x_1, r_2 : x_2, r_3 : x_3\}$
- $\mathcal{A}_1^\sharp \leftarrow \mathcal{A}^\sharp$
- $s \leftarrow (\top, \mathcal{A}_1^\sharp, \emptyset)$
- $(p_P, \mathcal{R}_P^\sharp, *_P^\sharp) \leftarrow \mathit{interpret}(s, P)$
- $p_s \leftarrow \mathit{proj}(p_P, \mathit{Img}(\mathcal{A}^\sharp) \cup \{\mathcal{R}_P^\sharp(r_0)\})$
- **return** $(p_s, \mathcal{A}^\sharp, \mathcal{R}_P^\sharp)$

the same procedure. Virtual inlining has two negative impacts on the analysis complexity. First, when a procedure is called several times in the same program, it must be analysed each time. Second, the number of variables used to analyze a procedure has an exponential impact on the complexity of the analysis of the procedure. Virtual inlining adds variables of the sub-procedure to those of the calling procedure, thus exponentially impacting the complexity.

In this section we detail a *modular* abstract interpretation analysis, which relies on the extensions previously presented in this paper. Each procedure is analyzed only once per program analysis, and in isolation from other procedures. This significantly reduces the complexity of the analysis and improves its scalability. The approach is similar to the *functional* approach to inter-procedural data-flow analysis [42], although we consider a significantly more complex abstract domain. Modular abstract interpretation has been considered before (e.g. in [43]), but only for analysis of source code.

The modular analysis consists of two parts: 1) inferring a *summary* for each procedure, representing how a call to the procedure impacts the state of the caller; 2) deriving *call predicates* for each procedure call, which represent inferred constraints on the values of the procedure arguments at the call site. Call predicates are not required for the modular abstract interpretation of the program, they are only used during the symbolic WCET computation step.

In the following, a program is represented as a set of procedures $\mathcal{P}$, one of which is the main procedure, i.e. the entry point of the program. As previously, a procedure $P \in \mathcal{P}$ is defined as a sequence of labeled instructions $l_0 : I_0, \dots, l_n : RET$.

7.1.1 Procedure summary

In the 32-bit ARM convention [6], the value returned by a procedure is stored in register r_0. The summary of a procedure is defined as a tuple $(p, \mathcal{A}_0^\sharp, \mathcal{R}^\sharp)$, where:

- p is a polyhedron that represents the abstract state of the analysis at the end of the procedure interpretation;
- $\mathcal{A}^\sharp$ is the argument mapping of the procedure;
- $\mathcal{R}^\sharp$ is a register mapping;
- There is no memory mapping since we only consider procedures without side effects.

Let F denote the procedure under analysis. Let $\mathcal{A}_0^\sharp$ denote the argument mapping of F. Let $M = \mathit{interpret}(F)$. Let l_e be the end label of F, i.e. $F[l_e] = RET$. Let $M[l_e] = (p_e, \mathcal{R}_e^\sharp, *_e^\sharp)$. Then, the summary of F is $(p_{e'}, \mathcal{A}_0^\sharp, \mathcal{R}_e^\sharp))$, where $p_{e'} = \mathit{proj}(p_e, \mathit{Img}(\mathcal{A}^\sharp) \cup \{\mathcal{R}_e^\sharp(r_0)\})$.

```
add_nozero:                 @  int add_nozero(int a , int b){
  add  r2, r0, r1           @    int res = a+b;
  cmp  r2, #0               @
  bne  .L2                  @    if(res == 0){
  add  r2, r2, #1           @      res++;
.L2:                        @    }
  mov  r0, r2               @    return res;
  bx   lr                   @  }
```

(a) Arm32 assembly code

(b) Abstract interpretation of the procedure

Label	Polyhedron	Registers
1	p_1	$\mathcal{R}^\sharp_1 = \mathcal{A}^\sharp = \{r_0 : x_0, r_1 : x_1\}$
3	$p_3 = \langle x_2 = x_0 + x_1 \rangle$	$\mathcal{R}^\sharp_3 = \mathcal{R}^\sharp_1[r_2 : x_2]$
5	$p_5 = p_3 \sqcap_\diamond \langle x_2 = 0 \rangle$	$\mathcal{R}^\sharp_5 = \mathcal{R}^\sharp_3$
6	$p_6 = p_5 \sqcap_\diamond \langle x_3 = x_2 + 1 \rangle$	$\mathcal{R}^\sharp_6 = \mathcal{R}^\sharp_3[r_2 : x_3]$
7	$p_7 = \langle x_0 + x_1 \leq x_2 \leq x_0 + x_1 + 1 \rangle$	$\mathcal{R}^\sharp_7 = \mathcal{R}^\sharp_3$
8	$p_8 = p_7 \sqcap_\diamond \langle x_4 = x_2 \rangle$	$\mathcal{R}^\sharp_8 = \mathcal{R}^\sharp_7[r_0 : x_4]$

Figure 7: A simplified pure function that sums its inputs and never returns 0

```
caller:                     @  int caller(int x, int y, int z){
  add  r3, r0, r1           @    int f = x + y;
  mov  r1, r2               @    // set z as second argument
  mov  r0, r3               @    // set f as first argument
  bl  add_nozero            @
  mov  r3, r0               @
  mov  r0, r3               @    return add_nozero(f, z);
  bx   lr                   @  }
```

(a) Arm32 assembly code

(b) Abstract interpretation of the procedure

Label	Polyhedron	Registers
1	$p_{1'}$	$\mathcal{R}^\sharp_{1'} = \{r_0 : x_5, r_1 : x_6, r_2 : x_7\}$
3	$p_{3'} = p_{1'} \sqcap_\diamond \langle x_8 = x_5 + x_6 \rangle$	$\mathcal{R}^\sharp_{3'} = \mathcal{R}^\sharp_{1'}[r_3 : x_8]$
5	$p_{5'} = p_{3'} \sqcap_\diamond \langle x_9 = x_7, x_{10} = x_8 \rangle$	$\mathcal{R}^\sharp_{5'} = \mathcal{R}^\sharp_{3'}[r_0 : x_{10}, r_1 : x_9]$

Figure 8: A procedure that calls *add_nozero*

Algorithm 2 Summary instantiation

1: **function** InstantiateSummary$((p_s, \mathcal{A}^\sharp_s, \mathcal{R}^\sharp_s), (p, \mathcal{R}^\sharp, *^\sharp))$
2: $\quad (p_t, \mathcal{A}^\sharp_t, \mathcal{R}^\sharp_t) \leftarrow \mathit{fresh}((p_s, \mathcal{A}^\sharp_s, \mathcal{R}^\sharp_s))$
3: $\quad p'_t \leftarrow p_t$
4: $\quad$ **for all** $a \in \mathit{Dom}(\mathcal{A}^\sharp)$ **do**
5: $\quad\quad p'_t \leftarrow p'_t[\mathcal{R}^\sharp(a)/\mathcal{A}^\sharp_t(a)]$
6: $\quad p' \leftarrow p \sqcap_\diamond p'_t$
7: $\quad \mathcal{R}^\sharp_1 \leftarrow \mathcal{R}^\sharp$
8: $\quad$ **for all** $r \in \mathit{Dom}(\mathcal{R}^\sharp_t)$ **do**
9: $\quad\quad$ **if** $\mathcal{R}^\sharp_t(r) \in p'$ **then**
10: $\quad\quad\quad \mathcal{R}^\sharp_1 \leftarrow \mathcal{R}^\sharp_1[r : \mathcal{R}^\sharp_t(r)]$
11: $\quad$ **return** $(p', \mathcal{R}^\sharp_1, *^\sharp)$

Example 7.1. *The procedure* add_nozero *in Figure 7 is a pure function. Its return value depends on its two input arguments. To ease understanding, the assembly code is slightly simplified compared to what a compiler would actually produce. The procedure is summarized as:*

$$(\mathit{proj}(p_8, \mathit{Img}(\mathcal{A}^\sharp) \cup \{\mathcal{R}^\sharp_8(r_0)\}), \mathcal{A}^\sharp, \mathcal{R}^\sharp_8) = (\langle x_0 + x_1 \le x_4, x_4 \le x_0 + x_1 + 1\rangle, \mathcal{A}^\sharp, \mathcal{R}^\sharp_8)$$

In other words, $\mathit{arg}_1 + \mathit{arg}_2 \le \mathit{return\_value} \le \mathit{arg}_1 + \mathit{arg}_2 + 1$.

7.1.2 Summary instantiation

Let $p[x_i/x_j]$ denote the polyhedron resulting from the substitution of variable x_j by x_i in p. The instantiation of a procedure summary is detailed in Algorithm 2. It takes as arguments the procedure summary $(p_s, \mathcal{A}^\sharp_s, \mathcal{R}^\sharp_s)$ and the abstract state at the procedure call $(p, \mathcal{R}^\sharp, *^\sharp)$. At line 2, it creates a fresh copy of the summary, where all the variables of the summary are substituted by fresh variables. From line 3 to line 5, it substitutes the variables mapped to procedure arguments in the summary by the actual argument variables at the call site. Line 6 intersects the (modified) polyhedron of the summary with the polyhedron at the call site. From line 7 to line 10, it updates the register mapping of the caller to account for the register modifications performed by the callee. Line 11 returns the abstract state obtained after the call.

Example 7.2. *The procedure* caller *of Figure 8 calls the procedure* add_nozero *at label 5. By instantiating the summary obtained in Example 7.1, we obtain the abstract state* $(p_{6'}, \mathcal{R}^\sharp_{6'}, *^\sharp_{6'})$ *at label 6 of* caller*, with:*

$$\begin{aligned} p_{6'} &= p_{5'} \sqcap_\diamond (\langle x'_0 + x'_1 \le x'_4 \le x'_0 + x'_1 + 1\rangle[x_{10}/x'_0, x_9/x'_1]) \\ \mathcal{R}^\sharp_{6'} &= \mathcal{R}^\sharp_8[r_0 : x'_4, r_1 : x_9] \\ *^\sharp_{6'} &= \{\} \end{aligned}$$

where x'_n *denotes the fresh variable substituted for* x_n *in the summary.*

7.1.3 Call predicates

We derive call predicates at each call site. Each call predicate relates one argument of the callee to the arguments of the caller. In other words, it provides information on how this argument passed to the callee depends on the arguments of the caller.

Definition 7.1 (Call predicate)**.** *Let f be a procedure with an instruction that calls a procedure g at label l_i. Let $M = interpret(f)$, $(p, \mathcal{R}^\sharp, *^\sharp) = M[l_i]$. Let A_f denote the set of variables mapped to the arguments of f. Let A_{g_i} be such that $A_{g_i}(j)$ denotes the variables of the $(j+1)^{th}$ argument passed to g at call site l_i*[8]*. The* call predicate *$cpred_{g_i}(j)$ is defined as:*

$$cpred_{g_i}(j) = export(proj(p, Img(A_f) \cup \{A_{g_i}(j)\}))$$

where $export(p')$ exports p' as a set of constraints, after substituting $A_f(k)$ by the identifier `f_k`*, and $A_{g_i}(j)$ by the identifier* `g_i_j`*.*

Example 7.3. *Consider the procedure* caller *in Figure 8. For the call to* caller *at label 5, we have:*

$$\begin{aligned} cpred_{add\_nozero}(0) &= export(proj(p_{5'}, \{x_5, x_6, x_7, \} \cup \{x_{10}\}) \\ &= export(\langle x_{10} = x_5 + x_6 \rangle) \\ &= \{add\_nozero_0 = caller_0 + caller_1\} \end{aligned}$$

Similarly, we obtain $cpred_{add\_nozero}(1) = \{add\_nozero_1 = caller_2\}$

7.2 Modular WCET analysis

In this section, we detail the modular WCET analysis, which relies on the input conditions and call predicates inferred by the abstract interpretation.

7.2.1 Procedure calls and control-flow trees

In our previous work on symbolic WCET computation [10], for each procedure call, the CFT of the callee is inlined in the CFT of the caller. Instead, for our modular WCET analysis we introduce a new kind of tree to represent a procedure call.

Definition 7.2 (Call control-flow tree)**.** *Let f be a procedure and $(m_1, \ldots, m_n)$ be a set of call predicates. The tree $Call(f, (m_1, \ldots, m_n))$ represents a call to the procedure f, where $m_k = cpred_f(k)$ for $1 \le k \le n$.*

The abstract WCET of a call is defined as:

$$\omega(\text{Call}(f, (m_1, \ldots, m_n))) = f(m_1, \ldots, m_n)$$

where f identifies the WCET formula of f.

Example 7.4. *Let us consider the example of Figure 8. Let w_1 denote the WCET of instructions at lines 2-4, and w_2 denote the WCET of instructions at lines 6-8. Then, based on the call predicates obtained in Example 7.3, the WCET of procedure* caller *is:*

$$w_1 \oplus add\_nozero(add\_nozero_1 = f_1 + f_2, add\_nozero_2 = f_3) \oplus w_2$$

[8] In the following we omit subscript i when clear from context.

7.2.2 Simplification

We instantiate sub-formulas of procedure calls during formula simplification. To do so, we update input conditions so that they depend on arguments of the caller rather than on arguments of the callee. More formally, we introduce the following simplification rule:

$$f(m_1, \dots, m_n) \mapsto \mathit{inst}(f, p, \mathit{Dom}(p))$$

where[9]:

$$\begin{aligned}
p &= \langle m_1, \dots, m_n \rangle \\
\mathit{inst}((l, w), p, \mathit{vs}) &= (l, w) \\
\mathit{inst}(w \oplus w', p, \mathit{vs}) &= \mathit{inst}(w, p, \mathit{vs}) \oplus \mathit{inst}(w', p, \mathit{vs}) \\
\mathit{inst}(w \uplus w', p, \mathit{vs}) &= \mathit{inst}(w, p, \mathit{vs}) \uplus \mathit{inst}(w', p, \mathit{vs}) \\
\mathit{inst}(e \circledast w, p, \mathit{vs}) &= \mathit{proj}(p \sqcap_\diamond \langle e \rangle, \mathit{vs}) \circledast \mathit{inst}(w, p \sqcap_\diamond \langle e \rangle, \mathit{vs}) \\
\mathit{inst}(w^{n,l}, p, \mathit{vs}) &= \begin{cases} \mathit{inst}(w, p, \mathit{vs})^{n,l} & \text{if } n \text{ is constant} \\ \mathit{inst}(w, p, \mathit{vs})^{\mathit{lbound}(p \sqcap_\diamond \langle lb \leq n \rangle \}, \mathit{vs}, lb), l} & \text{otherwise} \end{cases}
\end{aligned}$$

Example 7.5 (Sub-formula instantiation)**.** *Consider the two procedures* caller *and* add_nozero *of Figure 7 and Figure 8. Assume the WCET of* add_nozero *to be:* $w_3 \oplus ((\mathit{add\_nozero}_1 + \mathit{add\_nozero}_2 = 0) \circledast w_4) \oplus w_5$*.*

After simplification, we obtain the following WCET for procedure caller*:*

$$w_1 \oplus (w_3 \oplus ((\mathit{caller}_1 + \mathit{caller}_2 + \mathit{caller}_3 = 0) \circledast w_4) \oplus w_5) \oplus w_2$$

8 Evaluation

In this section we present the evaluation of our approach. First, we detail experiments on TACLeBench. Second, we illustrate how on-line formula instantiation can be leveraged to implement adaptive real-time systems.

8.1 Experiments

We first present our experimental setup, to enable the reproduction of our experiments. Then, we detail our benchmarks selection criteria. Finally, we provide metrics obtained by running our tool on the selected benchmarks.

8.1.1 Experimental setup

We implemented our approach[10] as an extension to OTAWA, an open-source WCET analysis tool [8]. We used the following setup:

- Auxiliary analyses: loop bounds that are not symbolic are computed by Polymalys as detailed in [9];
- Infeasible paths analysis: none;

[9] Recall that function *lbound* was defined in Section 5.2.2

[10] See `WCET-procedure-arguments-as-parameters.md` at https://gitlab.cristal.univ-lille.fr/otawa-plugins/WSymb.

- Hardware analyses: the pipeline analysis is adapted from the Exegraph method [39] as detailed in [27]. The cache analysis is based on the classic cache categorization approach [26], combined with CFT transformations as previously illustrated in Example 4.3;
- Processor model: 1 ALU, 1 FPU, 1 MU. Integer addition costs 1 cycle, floating point addition 3 cycles, multiplication 6 cycles, division 15 cycles. Processor has a 4 stages pipeline (fetch, decode, execute, commit), a fetch queue of size 3, fetches 2 instructions per cycle, and can execute up to 4 instructions in parallel;
- L1 instruction cache: 64KB, LRU replacement policy, 1-way. The miss penalty is 10 cycles;
- Compilation: each benchmark is compiled as a standalone binary file using GCC version 10.3.1 for ARM, with flags -O0 -g -nostdinc -nostdlib -mtune=cortex-a8 -mfpu=neon -mfloat-abi=hard. *cjpeg_wrbmp* uses a custom memcpy implementation in order to compile with gcc, which does not compile without standard library otherwise;
- Analyses execution times: measured on an Intel® Core™ i7-8550U CPU @ 1.80GHz × 8 with 16 GB of RAM.

8.1.2 Benchmark selection

We run our experiments on the TACLeBench benchmarks suite [24]. We did not analyze all the procedures of the benchmarks:

- 11 programs are not supported by OTAWA (out of the 54 of TACLeBench): 2 because of recursions (*fac* and *recursion*), 9 because of the incomplete support for division instructions (*adpcm_dec, adpcm_enc, ammunition, cjpeg_transupp, epic, h264_dec, huff_enc, quicksort* and *susan*);
- 181 procedures have arguments, out of the 1032 procedures of the supported programs;
- Procedures that contain switch-cases are excluded from the experiments, due to incomplete support by Otawa;
- The only data-type supported by the polyhedra analysis is integers. Thus it derives incorrect results for 4 procedures (*gsm_enc_norm, isqrt_usqrt, st_calc_Var_Stddev* and *st_sqrtf*);
- The polyhedra analysis is intractable for 31 procedures: it either executes for more than an hour, or runs out-of-memory. This happens for procedures with complex memory access patterns, which lead to an explosion of the number of variables in the polyhedron.

Among the remaining procedures, we present only the procedures for which the polyhedra analysis derived at least one input condition. Each procedure name is prefixed with the program it is part of (e.g. *fft_modff* is from the *fft* program). Only *gsm_dec_Long_Term_Synthesis_Filtering* and *mpeg2_dist2* have more than 4 arguments; we simply ignore the additional arguments.

Four procedures only have symbolic loop bounds: *audiobeam_adjust_delays*, *audiobeam_calculate_energy*, *audiobeam_find_max_in_array* and *audiobeam_find_min_in_arr*. Five procedures have both symbolic loops bounds and symbolic conditional statements: *audiobeam_calc_distances*, *g723_enc_quan*, *ludcmp_test*, *minver_minver* and *minver_mmul*. The remaining procedures only have symbolic conditional statements.

Table 2: WCET variability (in cycles)

Procedure	IPET	CFT		
		Lowest	Highest	Diff (%)
audiobeam_adjust_delays	9,261	1,718	9,383	81.7
audiobeam_calc_distances	174,295	340	176,550	98.1
audiobeam_calculate_energy	303	303	303	0.0
audiobeam_find_max_in_arr	5,274	1,331	5,366	75.2
audiobeam_find_min_in_arr	5,327	1,384	5,429	74.5
audiomeam_wrapped_dec	525	490	525	6.7
audiobeam_wrapped_dec_offset	316	281	316	11.1
audiobeam_wrapped_inc	563	528	563	6.2
audiobeam_wrapped_inc_offset	344	309	344	10.2
cjpeg_wrbmp_write_colormap	1,266,466	1,188,091	1,288,709	7.8
fft_modff	319	319	319	0.0
g723_enc_quan	4,621	341	5,291	93.6
g723_enc_reconstruct	702	335	702	38.9
gsm_dec_APCM_inverse_-quantization	15,024	15,259	15,297	0.2
gsm_dec_APCM_quantization_-xmaxc_to_exp_mant	1,311	1235	1,353	8.7
gsm_dec_asl	855	268	855	68.7
gsm_dec_asr	420	290	420	31.0
gsm_dec_Long_Term_-Synthesis_Filtering	47,389	48,652	48,703	0.1
gsm_dec_sub	343	305	343	11.1
gsm_enc_asl	855	268	855	68.7
gsm_enc_asr	420	290	420	31.0
gsm_enc_div	5,072	3,287	5,092	35.4
gsm_enc_sub	343	305	343	11.1
lift_do_impulse	1,117	1,135	1,197	5.2
ludcmp_test	108,705	9,741	110,841	91.2
minver_minver	53,356	359	57,141	99.4
minver_mmul	12,300	380	12,492	97.0
mpeg2_dist2	134,023	134,305	134,368	0.0
ndes_getbit	383	349	383	8.9
rijndael_dec_fseek	470	380	470	19.1
rijndael_enc_fseek	449	381	449	15.1

8.1.3 WCET variability

Table 2 summarizes our results regarding WCET variability. As illustrated in Section 8.2, this variability can be leveraged to implement adaptive systems. The *Procedure* column contains the name of the analyzed procedure. We first report the WCET computed with *IPET*. The *CFT* sub-columns indicate the *Lowest* and the *Highest* WCET computed by our technique, as well as the difference between these two columns as a percentage (in the *Diff* column). We relied on manual code inspection to determine parameter values that yield the lowest and highest WCET values. Automating this process in non-trivial and will be the subject of future works.

For 26 out of 31 procedures, the variability, i.e. the difference between the highest and the lowest WCET, is more than 5%. Many examples exhibit from 30% to 70% variability, usually due to symbolic conditional statements. Although our tool supports loop bounds expressed as linear expressions on parameter values, benchmarks only contain loop bounds expressed either as a constant or as a single parameter.

The highest variability values (those over 90%) are exhibited when loop bounds are set to 0, which is not likely to happen commonly in actual applications. Similarly, for *minver_minver* the lowest WCET is obtained for an unlikely argument value: it occurs when the size of the matrix is lower than 2 or higher than 500, in which case the procedure returns immediately.

Only two procedures exhibit no variability even though their WCET formula contains symbols. The formula for *fft_modff* contains an *if-then-else* whose condition depends on the procedure arguments. However, the input condition inferred for one of the two *if-then-else* branches is approximated to *true*, because the corresponding condition contains a disjunction (the input condition inferred for the other branch is not approximated, that condition is a conjunction). Furthermore, the WCET for that branch is higher than that of the other branch. Thus the WCET of the whole *if-then-else* is always the same, it does not change depending on argument values. The formula for *audiobeam_calculate_energy* contains a symbolic loop bound, however its maximum value is 0 in TACLeBench.

The *Highest* WCET is slightly higher than the WCET inferred by IPET (1.4% on average, 0% minimum, 12.7% maximum). This is because: 1) the transformation from CFG to CFT can introduce execution paths that do not exist in the CFG (see [10] for details); 2) the hardware analyses are slightly more pessimistic in our approach (e.g. loops with multiple exits impair the pipeline analysis, loop headers duplicated by the transformation to CFT impair the cache analysis).

8.1.4 Analysis time

The analysis times of IPET and our technique are presented in Table 3. The *IPET* column indicates the analysis time for IPET. The *CFT* sub-columns indicate the analysis time for our technique: *Polyhedra* indicates the time spent for the abstract interpretation, while *Symbolic WCET* indicates the time spent for the WCET computation. The sum of the *Polyhedra* and the *Symbolic WCET* columns give the global execution time of our technique.

For small procedures, the analysis times are similar for the IPET analysis, the polyhedra analysis, and the symbolic WCET computation. This is because the execution time for the CFG reconstruction dominates the execution time of the actual analysis.

For bigger procedures, the analysis time increases, and the analysis times for IPET and for the Symbolic WCET computation (without considering polyhedra analysis times) are similar. This is because the cache analysis, performed by both, dominates the rest of the analysis. Its complexity is exponential in the depth of loop nests. In some cases, the polyhedra analysis has higher execution times. This corresponds to programs with many memory accesses, which cause the introduction of many variables and constraints. Furthermore, we also noticed that

Table 3: Analysis times (in seconds)

Procedure	**IPET**	**CFT**	
		Polyhedra	**Symbolic WCET**
audiobeam_adjust_delays	1.120	1.006	1.096
audiobeam_calc_distances	222.809	20.881	216.863
audiobeam_calculate_energy	0.242	0.099	0.246
audiobeam_find_max_in_arr	0.869	0.346	0.827
audiobeam_find_min_in_arr	0.852	0.471	0.820
audiomeam_wrapped_dec	0.303	0.034	0.297
audiobeam_wrapped_dec_offset	0.163	0.022	0.162
audiobeam_wrapped_inc	0.463	0.039	0.455
audiobeam_wrapped_inc_offset	0.241	0.015	0.238
cjpeg_wrbmp_write_colormap	7.234	113.109	7.383
fft_modff	0.140	0.007	0.141
g723_enc_quan	0.247	0.598	0.244
g723_enc_reconstruct	24.510	0.045	24.790
gsm_dec_APCM_inverse_-quantization	6.551	8.199	6.441
gsm_dec_APCM_quantization_-xmaxc_to_exp_mant	1.067	0.184	1.033
gsm_dec_asl	0.495	0.059	0.484
gsm_dec_asr	0.272	0.028	0.266
gsm_dec_Long_Term_Synthesis_-Filtering	2.175	2.844	2.095
gsm_dec_sub	0.226	0.022	0.220
gsm_enc_asl	0.498	0.057	0.483
gsm_enc_asr	0.274	0.025	0.266
gsm_enc_div	0.904	0.409	0.874
gsm_enc_sub	0.225	0.015	0.219
lift_do_impulse	0.391	0.058	0.385
ludcmp_test	4.702	21.641	4.636
minver_minver	72.026	645.606	71.018
minver_mmul	1.714	6.300	1.640
mpeg2_dist2	9.410	37.567	9.154
ndes_getbit	0.381	0.035	0.357
rijndael_dec_fseek	0.259	0.053	0.252
rijndael_enc_fseek	0.212	0.057	0.204

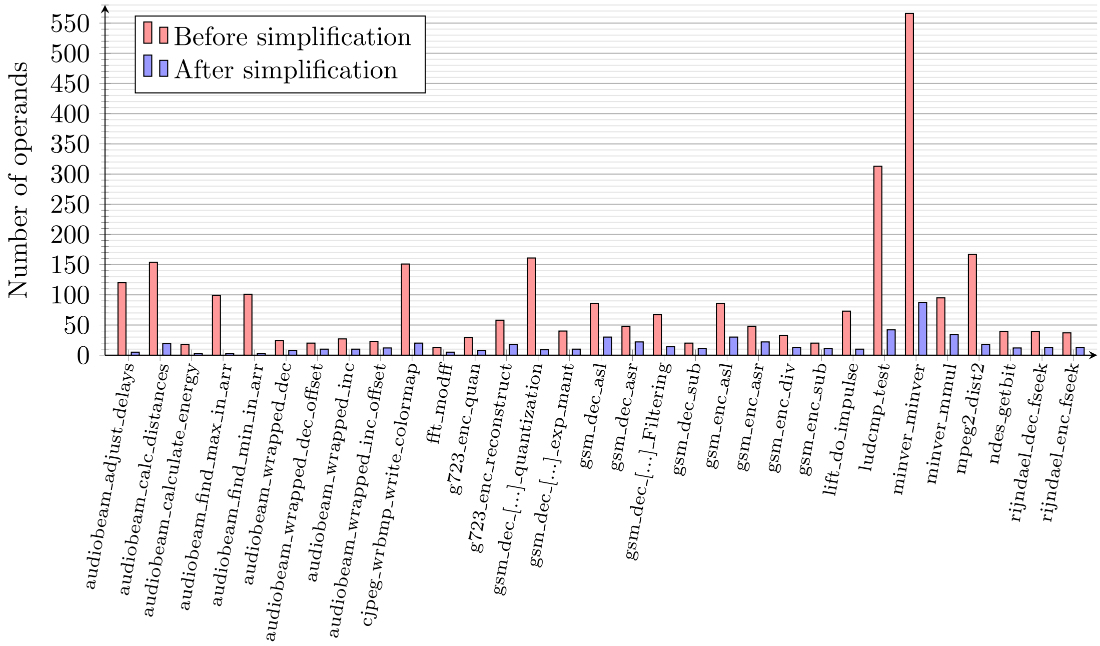


Figure 9: Symbolic WCET formula size before and after simplification

our extensions to support input conditions have very little to no impact on the symbolic WCET analysis time.

The major difference between our work and IPET concerning analysis time is the abstract interpretation part, which extracts input conditions. There are many ways to improve the scalability of this part of our approach, by adapting the rich set of optimization techniques developed by the community on abstract interpretation over the past decades. Nonetheless, our approach is already capable of producing WCET formulas for programs that are currently out of the scope of other tools in the literature.

8.1.5 Embeddability

The size of the initial and simplified formulae are reported in Figure 9. A simplified formula typically contains between 10 and 50 operands. Its size depends on the number of input conditions in the non-simplified formula. The largest formula (*minver_minver*) is reduced to 15% of its initial size by our simplification procedure.

Table 4 reports instantiation times (in cycles) for a selection of procedures with various characteristics, in terms of WCET, variability, and formula size. *Instantiation* indicates the WCET of the instantiation program computed by OTAWA. *Max gain* is the difference between the highest and the lowest WCET. *WCET* reports the *Highest* WCET of Table 2. *Op* reports the number of operands in the formula, from Figure 9.

On-line instantiation can be considered only when *Max gain* is significantly larger than *Instantiation*. This is the case for most procedures of Table 2, and the difference is actually quite large. For instance, for *cjpeg_wrbmp_write_colormap*, the instantiation takes 105 cycles while $100,513$ cycles can be reclaimed for other tasks. On the other extreme, the instantiation time of *audiobeam_wrapped_dec_offset* is larger than its WCET, so on-line instantiation offers no benefit.

Table 4: Instantiation times (in cycles)

Procedure	Inst.	Max gain	WCET	Op
audiobeam_adjust_delays	155	7,665	9,383	5
audiobeam_calc_distances	137	176,210	176,550	19
audiobeam_find_max_in_arr	119	4,035	5,366	3
audiobeam_find_min_in_arr	119	4,045	5,429	3
audiobeam_wrapped_dec_offset	74	35	525	10
cjpeg_wrbmp_write_colormap	105	100,618	1,288,709	20
g723_enc_quan	143	4,950	5,291	8
g723_enc_reconstruct	235	273	702	18
gsm_dec_asl	232	587	855	30
ludcmp_test	1,472	101,100	110,841	42
minver_minver	2,564	56,782	57,141	87
mpeg2_dist2	100	63	134,368	18

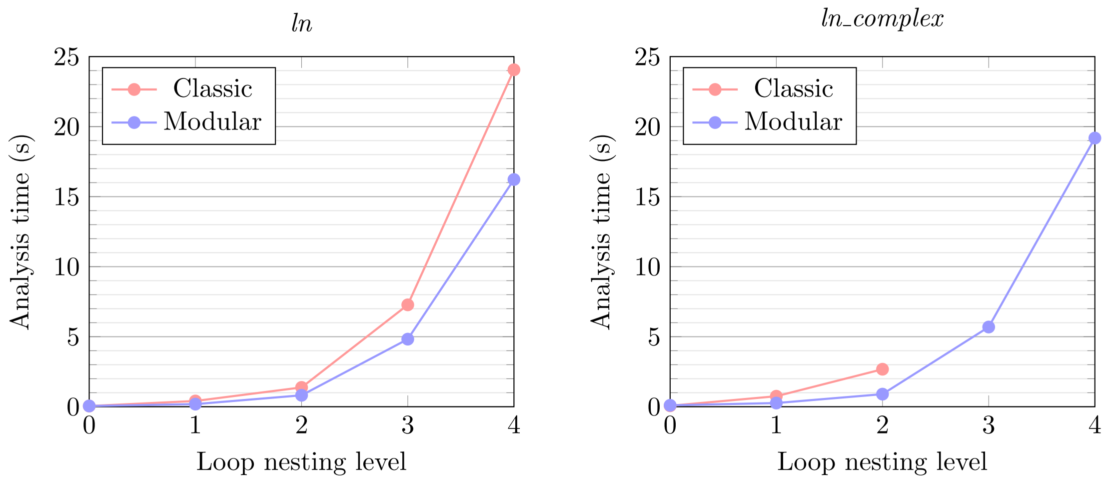


Figure 10: Comparison between classic and modular analysis time (in seconds)

8.1.6 Modular WCET analysis

We use two synthetic programs[11], *ln* and *ln_complex*, to emphasize the benefits of the modular analysis. They call a procedure at different loop nest levels: from *ln0* (no loop, only a procedure call), to *ln4* (loop > loop > loop > loop > procedure call). *ln* calls a simple procedure that performs 4 additions. *ln_complex* calls a procedure that contains conditional statements and performs an addition in each branch. Increasing the loop nest level stresses the analysis, because the number of times the procedure call is analyzed is exponential in the nesting level. Even though widening is applied to speedup analysis convergence, the body of a loop must be analyzed at least two times (possibly more depending on the widening operator).

Figure 10 details the abstract interpretation time for different loop nest levels. *Modular* corresponds to the modular analysis time and *Classic* to the non-modular analysis time. Results show that when there is no loop in the program (*ln0*), the modular abstract interpretation is slightly

[11] https://gitlab.cristal.univ-lille.fr/sgrebant/artificial-benchmarks.

slower. This is due to the overhead for computing the procedure summary and instantiating it. However, when the procedure is analyzed repeatedly (i.e. *ln1*, *ln2*, *ln3* and *ln4*), the modular analysis is significantly faster. This is especially true for *ln3* and *ln4* of *ln_complex*, where the non-modular analysis fails after 5 hours, with a segmentation fault, whereas the modular analysis completes the analysis in less than 20 seconds.

We also ran the complete modular WCET analysis on compatible procedures of TACLeBench. In comparison to the non-modular analysis, resulting WCET values are unchanged. In terms of analysis time, the impact of the modular analysis on the symbolic WCET computation part is negligible, because this part has a low complexity.

8.2 Application to adaptive real-time systems

In this section, we discuss the application of our WCET estimation approach to adaptive real-time systems. Real-time literature usually focuses on schedulability analysis for such systems. Instead, here we consider practical implementation aspects.

8.2.1 Semi-clairvoyant mixed-criticality scheduling

Recently, adaptive scheduling has gained interest following work on semi-clairvoyant scheduling for mixed-criticality systems [1]. The system model is based on the *dual-criticality* model of Vestal [45], where a system has two distinct criticality levels, LO (for *low*) and HI (for *high*). The workload consists of a set of tasks $\{\tau_i(\chi_i, [C_i^L, C_i^H]), T_i\}_{0 \leq i < n}$, where:

- $\chi_i \in \{LO, HI\}$ denotes the criticality of the task;
- C_i^L and C_i^H denote the LO-criticality and HI-criticality WCET of the task, such that $C_i^L \leq C_i^H$
- T_i is the *period* of the task and defines the minimum duration between two successive releases, also called *job*, of the task[12].

In *semi-clairvoyant scheduling*, the WCET of a job is estimated at its release. This estimate $\gamma_{i,j}$ equals either C_i^L or C_i^H. The system starts in LO-criticality mode, where every job of must complete before its deadline (the next job released by the same task). Whenever the estimate $\gamma_{i,j}$ of any job equals C_i^H, the system switches to HI-criticality mode, where only HI-criticality jobs need to complete before their deadlines.

Figure 11 depicts a possible implementation of such a system in C. Each job (one step of the loop) first acquires current input values (`getInputs`). Its WCET estimate is obtained by applying the WCET instantiation function of the task to the input values (`fWCET(inputs)`). If it exceeds the LO-WCET of the task, the system switches to HI-criticality. Note that there is no distinction between the code of LO and HI-criticality tasks. However, only LO-criticality tasks are suspended at mode switch (by `suspendAllLo`). Function `doWork` implements the actual task functionality.

The scheduler function (`schedule`) is called at periodic time intervals (as defined by the scheduler time granularity) and also when a task starts waiting for its next release (when it executes `waitPeriod`). Before switching to the new higher priority task, it tests whether the system can transition back to LO-criticality mode (`goBackToLo`), in which case it does so by resuming all LO-criticality tasks (`resumeAllLo`). Suspended tasks are simply ignored when selecting the next task to schedule. Resuming a task puts it back into the list of tasks ready to be scheduled.

[12] [1] assumes a more general model of jobs that may or may not be released periodically. We opt for a periodic model to make the discussion more concrete.

```
1  void mixedCritTask() {
2    int inputs[4];
3
4    while(1) {
5      getInputs(inputs);
6      if(fWCET(inputs)>CLo)
7        suspendAllLo();
8      doWork();
9      waitPeriod();
10   }
11 }
```

(a) Task code (LO or HI task)

```
void schedule() {
  saveContext();

  if(goBackToLo())
    resumeAllLo();
  selectNextTask();

  restoreContext();
}
```

(b) Scheduler code

Figure 11: Implementing semi-clairvoyant mixed-criticality scheduling

There is a slight difference between the implementation proposed in Figure 11 and the theoretical semi-clairvoyant model: in Figure 11, the WCET estimation occurs at the *start time* of the job (i.e. at the time when it is first selected for execution by the scheduler), while in the theoretical model it occurs at the *release time* of the job. To adhere more closely to the theoretical model, we can simply move L5-7 out of the task function and into the callback function of the periodic timer of the task. This timer is the actual trigger for new job releases; its callback is usually triggered by interruption and is thus not delayed by the scheduler. The pros and cons of both options (at release time or at start time) will be explored in future works.

8.2.2 Adaptive control

In *adaptive control*, the controller of the system adapts to parameters which vary or are initially uncertain. Such control is commonly used in embedded systems, as illustrated in the simple example of Figure 1. The parameter-space is often large, making control law computation very intensive. Implementing such adaptive control in real-time systems induces a tradeoff between control precision and computation time.

Figure 12 depicts the implementation of an adaptive control task using our WCET estimation approach. The time budget for a job is estimated after input acquisition (`getBudget`). The estimated WCET for the job is compared against its budget. If the estimation exceeds the WCET, the job executes a simplified version of the control law (`simpleWork`), which gives imprecise results but executes quickly. Otherwise, it executes a more refined control law (`complexWork`) that gives better results but takes more time to execute.

9 Conclusion

We presented a parametric WCET analysis that accounts for the effect of procedure argument values on the control-flow of the procedure. It first infers input conditions by abstract interpretation. Then, based on these results, the analysis produces a parametric WCET formula that depends on the procedure arguments. We also detailed a modular version of the analysis. Experiment show that our approach is adaptive and embeddable. We also illustrated how this approach can be used to implement adaptive real-time systems.

For future works, we plan to extend the modular analysis to support non-pure functions. The main challenge lies in developing an inter-procedural abstract interpretation procedure that

```
void adaptiveTask() {
  int inputs[4];

  while(1) {
    getInputs(inputs);
    if(fWCET(inputs)>getBudget())
      simpleWork();
    else
      complexWork();
    waitPeriod();
  }
}
```

Figure 12: Implementing an adaptive control task

supports procedures with side-effects.

A Rewriting rules equivalence proofs

In the following proofs, e_k and e_l are input conditions, w_1 and w_2 are abstract WCETs, *it* is an integer and l is a loop identifier. For the sake of readability, *true* and *false* values are replaced respectively by 1 and 0. The proofs of all the rules of Figure 6 are presented, except for rules (10), (11) and (12) since those are direct consequences of the application of the $\circledast$ operator semantic and thus are correct by construction. All the proofs are case by case proofs on the possible values of e_k and e_l.

Proof of rule (1). Property: $(e_k \wedge e_l) \circledast w_1 = (e_l \wedge e_k) \circledast w_1$. The proof directly follows from the commutativity of $\wedge$. □

Proof of rule (2). Property: $e_k \circledast w_1 \oplus e_l \circledast w_2 = e_l \circledast w_2 \oplus e_k \circledast w_1$

1. Case $e_k = 0$
$$0 \circledast w_1 \oplus e_l \circledast w_2 = \theta \oplus e_l \circledast w_2 = e_l \circledast w_2$$
$$e_l \circledast w_2 \oplus 0 \circledast w_1 = e_l \circledast w_2 \oplus \theta = e_l \circledast w_2$$

2. Case $e_l = 0$
$$e_k \circledast w_1 \oplus 0 \circledast w_2 = e_k \circledast w_1 \oplus \theta = e_k \circledast w_1$$
$$0 \circledast w_2 \oplus e_k \circledast w_1 = \theta \oplus e_k \circledast w_1 = e_k \circledast w_1$$

3. Case $e_k = e_l = 1$
$$1 \circledast w_1 \oplus 1 \circledast w_2 = w_1 \oplus w_2$$
$$1 \circledast w_2 \oplus 1 \circledast w_1 = w_2 \oplus w_1 = w_1 \oplus w_2$$

□

Proof of rule (3). Property: $e_k \circledast w_1 \uplus e_l \circledast w_2 = e_l \circledast w_2 \uplus e_k \circledast w_2$

1. Case $e_k = 0$
$$0 \circledast w_1 \uplus e_l \circledast w_2 = \theta \uplus e_l \circledast w_2 = e_l \circledast w_2$$
$$e_l \circledast w_2 \uplus 0 \circledast w_1 = e_l \circledast w_2 \uplus \theta = e_l \circledast w_2$$

2. Case $e_l = 0$

$$e_k \circledast w_1 \uplus 0 \circledast w_2 = e_k \circledast w_1 \uplus \theta = e_k \circledast w_1$$
$$0 \circledast w_2 \uplus e_k \circledast w_1 = \theta \uplus e_k \circledast w_1 = e_k \circledast w_1$$

3. Case $e_k = e_l = 1$

$$1 \circledast w_1 \uplus 1 \circledast w_2 = w_1 \uplus w_2$$
$$1 \circledast w_2 \uplus 1 \circledast w_1 = w_2 \uplus w_1 = w_1 \uplus w_2$$

□

Proof of rule (4). Property: $e_k \circledast w_1 \oplus e_l \circledast w_1 = w_1 \quad \text{if } e_l \Leftrightarrow \neg e_k$

1. Case $e_k = 1 \wedge e_l = 0$

$$1 \circledast w_1 \oplus 0 \circledast w_1 = w_1 \oplus \theta = w_1$$

2. Case $e_k = 0 \wedge e_l = 1$

$$0 \circledast w_1 \oplus 1 \circledast w_1 = \theta \oplus w_1 = w_1$$

□

Proof of rule (5). Property: $e_k \circledast w_1 \uplus e_l \circledast w_1 = w_1 \quad \text{if } e_l \Leftrightarrow \neg e_k$

1. Case $e_k = 1 \wedge e_l = 0$

$$1 \circledast w_1 \uplus 0 \circledast w_1 = w_1 \uplus \theta = w_1$$

2. Case $e_k = 0 \wedge e_l = 1$

$$0 \circledast w_1 \uplus 1 \circledast w_1 = \theta \uplus w_1 = w_1$$

□

Proof of rule (6). Property: $e_k \circledast w_1 \oplus e_l \circledast w_2 = e_k \circledast (w_1 \oplus w_2) \quad \text{if } e_k \Leftrightarrow e_l$

1. Case $e_k = e_l = 0$

$$0 \circledast w_1 \oplus 0 \circledast w_2 = \theta \oplus \theta = \theta$$
$$0 \circledast (w_1 \oplus w_2) = \theta$$

2. Case $e_k = e_l = 1$

$$1 \circledast w_1 \oplus 1 \circledast w_2 = w_1 \oplus w_2$$
$$1 \circledast (w_1 \oplus w_2) = w_1 \oplus w_2$$

□

Proof of rule (7). Property: $e_k \circledast w_1 \uplus e_l \circledast w_2 = e_k \circledast (w_1 \uplus w_2) \quad \text{if } e_k \Leftrightarrow e_l$

1. Case $e_k = e_l = 0$

$$0 \circledast w_1 \uplus 0 \circledast w_2 = \theta \uplus \theta = \theta$$
$$0 \circledast (w_1 \uplus w_2) = \theta$$

2. Case $e_k = e_l = 1$

$$1 \circledast w_1 \uplus 1 \circledast w_2 = w_1 \uplus w_2$$
$$1 \circledast (w_1 \uplus w_2) = w_1 \uplus w_2$$

□

Proof of rule (8). Property: $e_k \circledast w_1 \oplus (e_k \wedge e_l) \circledast w_2 = e_k \circledast (w_1 \oplus e_l \circledast w_2)$

1. Case $e_k = 0$
$$0 \circledast w_1 \oplus (0 \wedge e_l) \circledast w_2 = \theta \oplus 0 \circledast w_2 = \theta \oplus \theta = \theta$$
$$0 \circledast (w_1 \oplus e_l \circledast w_2) = \theta$$
2. Case $e_l = 0$
$$e_k \circledast w_1 \oplus (e_k \wedge 0) \circledast w_2 = e_k \circledast w_1 \oplus 0 \circledast w_2 = e_k \circledast w_1 \oplus \theta = e_k \circledast w_1$$
$$e_k \circledast (w_1 \oplus 0 \circledast w_2) = e_k \circledast (w_1 \oplus \theta) = e_k \circledast w_1$$
3. Case $e_k = e_l = 1$
$$1 \circledast w_1 \oplus (1 \wedge 1) \circledast w_2 = w_1 \oplus 1 \circledast w_2 = w_1 \oplus w_2$$
$$1 \circledast (w_1 \oplus 1 \circledast w_2) = w_1 \oplus w_2$$

□

Proof of rule (9). Property: $e_k \circledast w_1 \uplus (e_k \wedge e_l) \circledast w_2 = e_k \circledast (w_1 \uplus e_l \circledast w_2)$

1. Case $e_k = 0$
$$0 \circledast w_1 \uplus (0 \wedge e_l) \circledast w_2 = \theta \uplus 0 \circledast w_2 = \theta \uplus \theta$$
$$0 \circledast (w_1 \uplus e_l \circledast w_2) = \theta$$
2. Case $e_l = 0$
$$e_k \circledast w_1 \uplus (e_k \wedge 0) \circledast w_2 = e_k \circledast w_1 \uplus 0 \circledast w_2 = e_k \circledast w_1 \uplus \theta = e_k \circledast w_1$$
$$e_k \circledast (w_1 \uplus 0 \circledast w_2) = e_k \circledast (w_1 \uplus \theta) = e_k \circledast w_1$$
3. Case $e_k = e_l = 1$
$$1 \circledast w_1 \uplus (1 \wedge 1) \circledast w_2 = w_1 \uplus 1 \circledast w_2 = w_1 \uplus w_2$$
$$1 \circledast (w_1 \uplus 1 \circledast w_2) = w_1 \uplus w_2$$

□

Proof of rule (13). Property: $e_k \circledast (e_l \circledast w_1) = e_k \circledast w_1 \qquad \text{if } e_k \Leftrightarrow e_l$

1. Case $e_k = e_l = 0$
$$0 \circledast (0 \circledast w_1) = \theta$$
$$0 \circledast w_1 = \theta$$
2. Case $e_k = e_l = 1$
$$1 \circledast (1 \circledast w_1) = w_1$$
$$1 \circledast w_1 = w_1$$

□

Proof of rule (14). Property: $(e_k \circledast w_1)^{it,l} = e_k \circledast (w_1)^{it,l}$

1. Case $e_k = 0$
$$(0 \circledast w_1)^{it,l} = (\theta)^{it,l} = \theta$$
$$0 \circledast (w_1)^{it,l} = \theta$$
2. Case $e_k = 1$
$$(1 \circledast w_1)^{it,l} = (w_1)^{it,l}$$
$$1 \circledast (w_1)^{it,l} = (w_1)^{it,l}$$

□

References

[1] K. Agrawal, S. Baruah, and A. Burns. Semi-Clairvoyance in Mixed-Criticality Scheduling. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 458–468, Hong Kong, China, Dec. 2019. IEEE. ISSN: 2576-3172. doi:10.1109/RTSS46320.2019.00047.

[2] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In R. Cousot and D. A. Schmidt, editors, *Static Analysis*, Lecture Notes in Computer Science, pages 52–66, Berlin, Heidelberg, 1996. Springer. doi:10.1007/3-540-61739-6_33.

[3] E. Althaus, S. Altmeyer, and R. Naujoks. Precise and efficient parametric path analysis. *SIGPLAN Not.*, 46(5):141–150, Apr. 2011. URL: http://doi.org/10.1145/2016603.1967697, doi:10.1145/2016603.1967697.

[4] E. Althaus, S. Altmeyer, and R. Naujoks. Symbolic Worst Case Execution Times. In A. Cerone and P. Pihlajasaari, editors, *Theoretical Aspects of Computing – ICTAC 2011*, Lecture Notes in Computer Science, pages 25–44, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-23283-1_5.

[5] S. Altmeyer, C. Hümbert, B. Lisper, and R. Wilhelm. Parametric Timing Analysis for Complex Architectures. In *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 367–376, Kaohsiung, Taiwan, Aug. 2008. IEEE. ISSN: 2325-1301. doi:10.1109/RTCSA.2008.7.

[6] Arm. Procedure Call Standard for the Arm® Architecture, 2023. URL: https://developer.arm.com/Additional%20Resources/ABI-Procedure%20Call%20Standard%20for%20the%20Arm%20Architecture.

[7] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1):3–21, 2008.

[8] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In S. L. Min, R. Pettit, P. Puschner, and T. Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, Lecture Notes in Computer Science, pages 35–46, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-16256-5_6.

[9] C. Ballabriga, J. Forget, L. Gonnord, G. Lipari, and J. Ruiz. Static Analysis Of Binary Code With Memory Indirections Using Polyhedra. In *VMCAI'19 - International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 11388 of *LNCS*, pages 114–135, Cascais, Portugal, Jan. 2019. Springer. URL: https://hal.archives-ouvertes.fr/hal-01939659, doi:10.1007/978-3-030-11245-5_6.

[10] C. Ballabriga, J. Forget, and G. Lipari. Symbolic WCET Computation. *ACM Transactions on Embedded Computing Systems*, 17(2):1–26, 2017. URL: https://dl.acm.org/doi/10.1145/3147413, doi:10.1145/3147413.

[11] S. Baruah and P. Ekberg. Graceful Degradation in Semi-Clairvoyant Scheduling. In B. B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13940, doi:10.4230/LIPIcs.ECRTS.2021.9.

[12] B. Benhamamouch, B. Monsuez, and F. Védrine. Computing WCET using symbolic execution. In *Second International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2008)*, pages 1–12, Leeds, UK, July 2008. ScienceOpen. Publisher: BCS Learning & Development. URL: https://www.scienceopen.com/hosted-document?doi=10.14236/ewic/VECOS2008.12, doi:10.14236/ewic/VECOS2008.12.

[13] A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr. The Auspicious Couple: Symbolic Execution and WCET Analysis. In C. Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 53–63, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2013/4122, doi:10.4230/OASIcs.WCET.2013.53.

[14] B. Blackham, M. Liffiton, and G. Heiser. Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 169–178, Berlin, Germany, Apr. 2014. IEEE. doi:10.1109/RTAS.2014.6926000.

[15] A. Burns and R. I. Davis. Schedulability Analysis for Adaptive Mixed Criticality Systems with Arbitrary Deadlines and Semi-Clairvoyance. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–24, Houston, TX, Dec. 2020. IEEE. doi:10.1109/RTSS49844.2020.00013.

[16] S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric WCET calculation. *Journal of Systems Architecture*, 57(6):614–624, June 2011. URL: https://www.sciencedirect.com/science/article/pii/S1383762110000676, doi:10.1016/j.sysarc.2010.06.009.

[17] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra. Exploiting Branch Constraints without Exhaustive Path Enumeration. In R. Wilhelm, editor, *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*, volume 1 of *OpenAccess Series in Informatics (OASIcs)*, pages 46–49, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://doi.org/10.4230/OASIcs.WCET.2005.816, doi:10.4230/OASIcs.WCET.2005.816.

[18] D. Chu and J. Jaffar. Symbolic simulation on complicated loops for WCET Path Analysis. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 319–328, New York, NY, USA, Oct. 2011. Association for Computing Machinery.

[19] J. Coffman, C. Healy, F. Mueller, and D. Whalley. Generalizing parametric timing analysis. *SIGPLAN Not.*, 42(7):152–154, June 2007. URL: http://doi.org/10.1145/1273444.1254795, doi:10.1145/1273444.1254795.

[20] J. S. Cohen. *Computer alegebra and symbolic computation: mathematical methods.* AK Peters, Natick, Mass, 2003.

[21] A. Colin and G. Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*, pages 50–59, Vienna, Austria, June 2002. IEEE. doi:10.1109/EMRTS.2002.1019185.

[22] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, Jan. 1977. Association for Computing Machinery. URL: http://doi.org/10.1145/512950.512973, doi:10.1145/512950.512973.

[23] S. Ding and H. B. K. Tan. Detection of Infeasible Paths: Approaches and Challenges. In L. A. Maciaszek and J. Filipe, editors, *Evaluation of Novel Approaches to Software Engineering*, Communications in Computer and Information Science, pages 64–78, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-45422-6_5.

[24] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis*, pages 2:1–2:10, Toulouse, France, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: https://hal.archives-ouvertes.fr/hal-02610690, doi:10.4230/OASIcs.WCET.2016.2.

[25] P. Feautrier. Parametric integer programming. *RAIRO - Operations Research*, 22(3):243–268, 1988. URL: http://www.rairo-ro.org/10.1051/ro/1988220302431, doi:10.1051/ro/1988220302431.

[26] C. Ferdinand and R. Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17(2):131–181, Nov. 1999. doi:10.1023/A:1008186323068.

[27] S. Grebant, C. Ballabriga, J. Forget, and G. Lipari. WCET analysis with procedure arguments as parameters. In *The 31st International Conference on Real-Time Networks and Systems (RTNS 2023)*, RTNS 2023, Dortmund, Germany, June 2023. ACM, New York, USA. doi:10.1145/3575757.3593655.

[28] J. Gustaffson, A. Ermedahl, and B. Lisper. Algorithms for Infeasible Path Calculation. In F. Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–6, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://doi.org/10.4230/OASIcs.WCET.2006.667, doi:10.4230/OASIcs.WCET.2006.667.

[29] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 57–66, Rio de Janeiro, Brazil, Dec. 2006. IEEE. doi:10.1109/RTSS.2006.12.

[30] C. Healy and D. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering*, 28(8):763–781, Aug. 2002. doi:10.1109/TSE.2002.1027799.

[31] J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, Nov. 1997.

[32] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, LCTES '95, pages 88–98, New York, NY, USA, Nov. 1995. Association for Computing Machinery. URL: http://doi.org/10.1145/216636.216666, doi:10.1145/216636.216666.

[33] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In J. Gustafsson, editor, *Proc. 3^{rd} International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, pages 77–80, Porto, July 2003.

[34] R. Metta, M. Becker, P. Bokil, S. Chakraborty, and R. Venkatesh. TIC: a scalable model checking based approach to WCET estimation. *ACM SIGPLAN Notices*, 51(5):72–81, June 2016. URL: http://doi.org/10.1145/2980930.2907961, doi:10.1145/2980930.2907961.

[35] 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 *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–242, Miami, FL, USA, Dec. 2005. IEEE. ISSN: 1052-8725. doi:10.1109/RTSS.2005.33.

[36] S. Mohan, F. Mueller, M. Root, W. Hawkins, C. Healy, D. Whalley, and E. Vivancos. Parametric timing analysis and its application to dynamic voltage scaling. *ACM Trans. Embed. Comput. Syst.*, 10(2):25:1–25:34, Jan. 2011. URL: http://doi.org/10.1145/1880050.1880061, doi:10.1145/1880050.1880061.

[37] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. *IEEE Computer*, 24(5):48–57, 1991.

[38] J. Reineke and J. Doerfert. Architecture-parametric timing analysis. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 189–200, Berlin, Germany, Apr. 2014. IEEE. ISSN: 1545-3421. doi:10.1109/RTAS.2014.6926002.

[39] C. Rochange and P. Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. In P. Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers II*, volume 5470, pages 222–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. URL: http://link.springer.com/10.1007/978-3-642-00904-4_12, doi:10.1007/978-3-642-00904-4_12.

[40] J. Ruiz and H. Cassé. Using SMT Solving for the Lookup of Infeasible Paths in Binary Programs. In F. J. Cazorla, editor, *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 95–104, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://doi.org/10.4230/OASIcs.WCET.2015.95, doi:10.4230/OASIcs.WCET.2015.95.

[41] J. Ruiz, H. Cassé, and M. de Michiel. Working Around Loops for Infeasible Path Detection in Binary Programs. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–10, Shanghai, China, Sept. 2017. IEEE. ISSN: 2470-6892. doi:10.1109/SCAM.2017.13.

[42] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences, 1978.

[43] P. Sotin and B. Jeannet. Precise interprocedural analysis in the presence of pointers to the stack. In *Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20*, pages 459–479. Springer, 2011.

[44] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 358–363, New York, NY, USA, July 2006. Association for Computing Machinery. URL: http://doi.org/10.1145/1146909.1147002, doi:10.1145/1146909.1147002.

[45] S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 239–243, Dec. 2007. ISSN: 1052-8725. doi:10.1109/RTSS.2007.47.

[46] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric Timing Analysis. In *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*, OM '01, pages 88–93, New York, NY, USA, Aug. 2001. Association for Computing Machinery. URL: http://doi.org/10.1145/384198.384230, doi:10.1145/384198.384230.

[47] T. Wei, J. Mao, W. Zou, and Y. 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.

[48] 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 Transactions on Embedded Computing Systems*, 7(3):1–53, Apr. 2008. URL: https://dl.acm.org/doi/10.1145/1347375.1347389, doi:10.1145/1347375.1347389.

[49] S. Wilhelm and B. Wachter. Symbolic state traversal for WCET analysis. In *Proceedings of the seventh ACM international conference on Embedded software - EMSOFT '09*, page 137, Grenoble, France, 2009. ACM Press. URL: http://portal.acm.org/citation.cfm?doid=1629335.1629354, doi:10.1145/1629335.1629354.