WCET analysis with procedure arguments as parameters

Sandro Grebant, Clément Ballabriga, Julien Forget, Giuseppe Lipari

WCET analysis with procedure arguments as parameters

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

Parametric Worst-Case Execution Time (WCET) is a static analysis that computes a WCET formula that depends on various parameters. The formula can be used off-line for fast parameter space exploration, and on-line for adaptive scheduling.

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

We use the TACLeBench benchmark programs to illustrate the impact of procedure arguments on the WCET. Our tool generates C code with bounded WCET that can be embedded in the analysed program to evaluate the formula on-line. We show that, for some programs, the potential gain is significantly larger than the cost of evaluating the formula, thus opening the possibility of adaptive scheduling.

1 Introduction

The schedulability analysis of a real-time system requires upper-bounds to the Worst-Case Execution Time (WCET) of each task. Tasks' execution times can exhibit a large variability due to software parameters (e.g. program inputs) or hardware parameters (e.g. cache state). Static 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. The estimated WCET must also be *tight* (close to the actual WCET) to keep resource over-provisioning to a minimum.

Instead of producing a single numeric WCET that upper-bounds any possible parameter combination, *parametric WCET analysis* produces a *formula* that represents the WCET as a function of the parameters. The formula can later be *instantiated* with concrete parameter values to provide an upper-bound to the execution time for those parameter values.

Parametric WCET formulae have two main uses. First, when doing design space exploration, a formula can be instantiated repeatedly off-line to quickly explore the parameters space with low execution cost. This can for instance be combined with sensitivity analysis [13, 27] to determine which parameter values make the system schedulable.

Second, a formula can be instantiated on-line to efficiently implement *adaptive* real-time systems. As many parameter values become known only at run-time (e.g. program inputs), on-line instantiation can produce a lower WCET than a single WCET computed off-line. This can benefit many adaptive scheduling techniques, such as energy-aware scheduling based on Dynamic Voltage and Frequency Scaling (DVFS) [35], slack reclaiming [17, 32, 36], or the recent semi-clairvoyant scheduling for mixed-criticality systems [1, 15, 10].

In this paper, we present a novel parametric WCET analysis, which analyzes the binary code of a procedure to produce a WCET formula that represents the WCET of the procedure as a function of its arguments.

1.1 Motivating example

We motivate our work with the example of Figure 1. This procedure is part of an implementation of the G.723 speech encoding standard, taken verbatim from TACLeBench [25].

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[1] compared to the quantization step `y` (line 6), the reconstructed difference is set to 0 (line 7[2]). 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 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] Addition on logarithmic values (`dqln` and `y`) amounts to multiplication.

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

```
1  int reconstruct(int sign, int dqln, int y)
2  {
3    short dql, dex, dqt, dq;
4
5    dql = dqln + ( y >> 2 );
6    if ( dql < 0 )
7      return ( ( sign ) ? -0x8000 : 0 );
8    else {
9      dex = ( dql >> 7 ) & 15;
10     dqt = 128 + ( dql & 127 );
11     dq = ( dqt << 7 ) >> ( 14 - dex );
12     return (( sign ) ? ( dq - 0x8000 ) : dq);
13   }
14 }
```

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 tenths 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 2 in Section 8.3.3). However, other more complex functions show a much larger variability and computing the formula on-line may make sense for those functions (see Section 8 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

Our approach is based on two previous works on symbolic WCET computation [9] and abstract interpretation [8]. 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 methodology 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. In this paper we present these extensions.

First, we devise an analysis that infers *input conditionals*, 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 [8] and is presented in Section 6. Also, we introduce a new type of node in the CFT to represent conditional branches subject to input conditionals. This is presented in Section 7.1

Second, in Section 5 we show how to apply a hardware analysis previously proposed for the IPET approach[38] to the tree-based approach of [9].

Third, in Section 7.2 we extend the symbolic computation to support formulae where the input conditionals appear as parameters. Furthermore, in Section 7.3 we propose extensive simplification procedures to reduce the size of the formulae. Finally, we provide a compiler that generates C code, which is optimized to have low WCET, to evaluate the formula on-line (Section 7.4).

As demonstrated with our experiments on TACLeBench, 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.

2 Related works

The most widely used WCET analysis technique is the Implicit Path Enumeration Technique (IPET) [31]. It takes 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 presented in [43].

Symbolic techniques have been considered in WCET analysis for different purposes. [11, 12, 19] uses symbolic techniques to speed up the analysis. In [33], symbolic analysis is used to trade off analysis time against tightness. Wilhelm et al. [44] model the effect of pipelines on the WCET using symbolic states. Reineke et al. [37] demonstrate how to represent various architectural effects, e.g. processor frequency, memory latencies or memory sizes, using parametric 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 [42, 34, 20, 35]. 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 more pessimistic. Concerning binary-level analyses, in [5] Altmeyer et al. rely on parametric ILP [26] to adapt IPET analysis to the parametric case, but the approach is computationally inefficient. In [16], Bygde et al. propose a different non-IPET

approach, which is more efficient but less tight. Althaus et al. [3, 4] try to improve on both efficiency and tightness.

Tree-based parametric WCET analyses have also been considered as a parametric alternative to IPET. Colin et al. [22] introduced a tree-based model of programs dedicated to symbolic WCET analysis. However, they did not consider the problem of producing such a model from a program. Ballabriga et al. [9] proposed a tree-based symbolic 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. We could use parametric execution blocks to represent parametric conditional statements: replace 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, 41, 18], symbolic execution [28, 29], or SMT solvers [14, 39, 40]. A survey about infeasible paths analysis can be found in [24]. These works focus on detecting *statically* infeasible paths, i.e. a program path that can never be executed because of some exclusive branch conditions and assignments. In comparison, our approach detects *dynamically* infeasible paths (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, none consider 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, existing works usually support a single parameter, or additions between a single parameter and a constant, which is insufficient to represent many input conditionals, 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 proceeds as follows.

CFG extraction the binary code is translated into a Control-Flow Graph, where nodes are basic blocks[3] and edges represent the program control-flow. We obtain a

[3] 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.

```
1  f:                          @  int f(int n) {
2    @ ...                     @    // A
3    str r0, [fp, #-32]        @    // A
4    @ ...                     @    // A
5    ldr r3, [fp, #-32]        @    // A
6    cmp r3, #10               @    if (n <= 10) // A
7    bgt .L2                   @    {   // still A
8    @ ...                     @        // C
9    b  .L3                    @    }   // C
10 .L2:                        @    else {
11   @ ...                     @        // B
12 .L3:                        @    }
13   @ ...                     @    // D
14   ldr r3, [fp, #-32]        @    // D
15   cmp r3, #-1               @    if (n <= -1)  // D
16   bgt .L4                   @    {   //still D
17   @ ...                     @        // F
18   b .L5                     @    }   // F
19 .L4:                        @    else {
20   @ ...                     @        // E
21 .L5:                        @    }
22   @ ...                     @    // G
23   bx lr                     @    return; // still G
24 .global  main               @  }
25 main:                       @  int main() {
26   @...                      @    // ...
27   ldr r0, [fp, #-8]         @    // Set arguments
28   bl f                      @    f(i); // call
29   @ ...                     @  }
```

Figure 2: Running example

CFG with basic blocks A to G. We rely on OTAWA [7] 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 extended the existing analysis to take into account the effect of the pipeline between blocks in the CFT (see Section 5).

Inferring input conditionals 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 `r0` ≥ 11 as the *input conditional* for branching to label *L2* (a.k.a. block B) and `r0` ≤ 10 if we do not branch. Similarly, the input conditionals `r0` ≥ 0 and `r0` ≤ -1 are inferred at line 16. Note that we use the notation `r0` for the sake of simplicity, but it corresponds to the value contained in register `r0` at the beginning of the procedure here, which is an invariant. We extended the abstract interpretation analysis of [8] to infer predicates on conditional branches and loops which depend on function arguments (see Section 6).

CFT with symbolic input conditionals 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 conditionals inferred by the abstract interpreter. We ex-

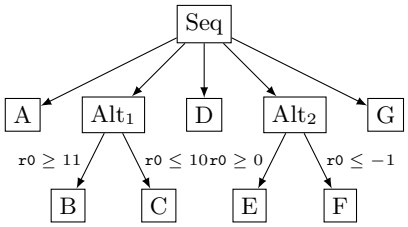


Figure 3: Control-Flow Tree for function f of Figure 2

tended the CFT of [9] with a new type of alternative node to model conditional branches (see Section 7.1).

WCET formula The CFT is translated into a WCET formula. Essentially: the WCET of a *Seq* node is the sum of the WCETs of its subtrees (denoted $\oplus$); the WCET of an *Alt* node is the maximum among the WCETs of its subtrees (denoted $\uplus$); the WCET of an alternatives' subtree is multiplied by its input conditional (denoted $\circledast$, where the input conditional can be seen as its binary equivalent, i.e. 1 if the input conditional is true, 0 otherwise). Thus, we obtain:

$$10 \oplus (((\texttt{r0} \geq 11) \circledast \omega(B)) \uplus ((\texttt{r0} \leq 10) \circledast 5)) \oplus \omega(D) \oplus$$
$$(((\texttt{r0} \geq 0) \circledast 10) \uplus ((\texttt{r0} \leq -1) \circledast 10)) \oplus 5$$

With respect to [9], we introduce input conditionals and the $\circledast$ operator, as described in Section 7.2.

Formula simplification The formula contains symbolic values, therefore it cannot be reduced to a numeric value. Instead, we reduce its size using special simplification rules. We obtain:

$$25 \oplus (((\texttt{r0} \geq 11) \circledast \omega(B)) \uplus ((\texttt{r0} \leq 10) \circledast 5)) \oplus \omega(D)$$

It is important to underline that, for the sake of clarity, in this example we show a simplified version of the formula. In reality, in order to correctly model the impact of caches, each WCET is represented by a list (see Section 4). Therefore, the operators used in the symbolic formula are special operators defined on lists. This means that, unfortunately, we could not reuse classical simplification procedures for integer formulae; instead, we had to establish and prove the correctness of our own simplification rules. This work is described in Section 7.3.

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 7.4, 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 on symbolic WCET

We start by recalling the main concepts of symbolic WCET computation [9]. It starts from a CFG representation of the binary program under analysis. First, it translates the CFG into a *control-flow tree* (CFT), whose nodes can be one of:

- Leaf(b), which holds the basic block b of the program;
- Seq($t_1, \ldots, t_n$), which represents the sequential execution of trees $(t_1, \ldots, t_n)$;
- Alt($t_1, \ldots, t_n$), which represents the execution of one tree among $(t_1, \ldots, t_n)$;
- Loop(l, t_b, n, t_e), which represents the loop, identified uniquely by l, that repeats the execution of t_b at maximum n times and exits by executing the tree t_e.

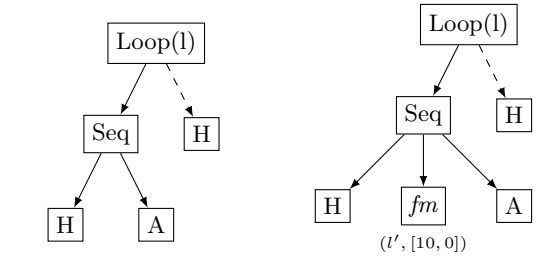
When located inside a loop, successive iterations of a CFT node can yield different WCETs. We represent the WCET of a CFT as an *abstract WCET*, defined as a pair (l, w), where l is a loop identifier and w is a list of integers sorted in non-increasing order. The list can contain duplicates and its smallest element is implicitly repeated infinitely. So for instance $(l, [10, 10, 5, 3])$ represents the WCET of a node inside loop l, whose WCET is at most twice 10, once 5, and 3 for all other iterations of loop l. In the following, $\top$ refers to the program top-level scope, and θ is the null WCET, where $\theta = (\top, [0])$. The abstract WCET $\omega(t)$ of a CFT t is computed inductively on the CFT structure as follows:

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

The operator $\oplus$ is a point-wise sum, for instance: $(l, [5, 3]) \oplus (l, [4]) = (l, [9, 7])$ (recall that the smallest element is implicitly repeated infinitely). The operator $\uplus$ is a list union, except that elements smaller than infinitely repeated ones are dropped, for instance: $(l, [5, 3]) \uplus (l, [4]) = (l, [5, 4])$. Concerning operation $(l, w)^{n,l'}$, there are two possible cases. When $l' = l$, the execution time is always the same, in this case we sum the first n elements of w; for instance: $(l, [5, 4])^{4,l} = (\top, [17])$. When $l' \neq l$, l' refers to a loop that contains l, thus successive executions yield different execution times, so we sum the elements of w by packs of n; for instance $(l, [5, 4])^{4,l'} = (l, [17, 16])$.

The abstract WCET representation is a bit involved, but it enables to represent a wide range of hardware and software facts. For instance, the CFT model can represent the effect of the instruction cache categorization technique of [2], as illustrated in the following example.

Example 1. *Figure 4a shows a simple CFT, which loops over basic block A. Node H is the loop test, repeated at the beginning of each iteration and also when exiting the loop (the dashed edge indicates the exit node). Assume that the miss-penalty of the instruction cache is of* 10 *and that the basic block A contains a* first-miss *cache access, i.e. the instruction is in the cache at all iterations except*



(a) Before transformation (b) After transformation

Figure 4: Instruction cache transformation

the first one. The effect of the instruction cache miss is modeled in Figure 4b by a new leaf fm, which represents the miss penalty of the cache at the first iteration only. The WCET of this leaf is expressed as $(l', [10, 0])$*, which means that the execution time of fm is* 10 *at one iteration and* 0 *for all the other iterations of* l'*.*

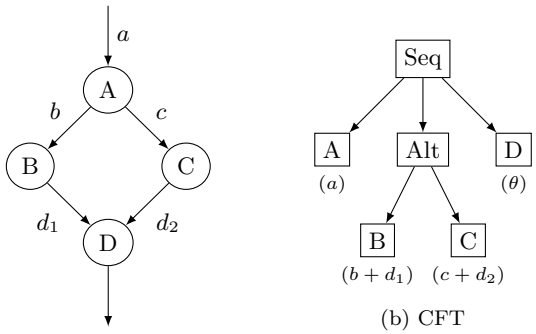
In the specific case of a *first-miss* cache access, that is to say the access is a miss each time we enter a certain loop, the abstract WCET loop identifier might not correspond to the loop that directly encloses the miss-penalty block, because the miss may not occur each time we enter this loop but each time we enter an outer loop. This corresponds to the $l \neq l'$ case presented above.

When some parameters of the CFT are unknown, $\omega(t)$ produces a formula containing symbolic values. In [9], symbols can be of two kinds: a symbolic WCET (e.g. $X \uplus (l, [4])$, where X is an unknown WCET) or a symbolic loop bound (e.g. $(l, [5, 3])^{N,l'}$, where N is an unknown integer loop bound). In order to decrease their size and evaluation time, WCET formulae are reduced using simplification rules. For instance, $((l, [5]) \oplus X) \uplus ((l, [4]) \oplus X)$ reduces to $(l, [5]) \oplus X$. Finally, the reduced formula is translated into C code, that can either be used off-line or on-line to instantiate the formula when symbol values become known.

5 Improving hardware analysis

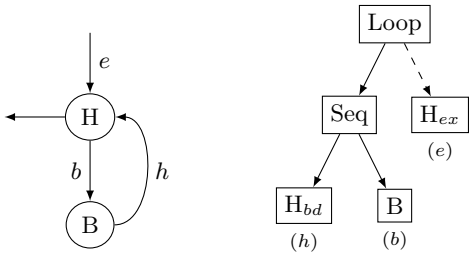
Computation of the WCET of each basic block of the CFG is performed during the *hardware analysis* step. We rely on the hardware analyses implementations of OTAWA [7]. In this section, we describe how we improved the representation of the effect of the processor pipeline in the CFT model. Note that those improvements are not required for our technique to be safe, but they greatly reduce its pessimism and provide more realistic results in our experiments.

The processor pipeline causes the execution time of an instruction to vary depending on the execution path that was taken before it. To model this dependency, the *execution graphs* method (Exegraph [38]) represents the WCET of a basic block of a CFG on its incoming edges. For instance, in Figure 5a, d_1 is the WCET of block D when it is preceded by the execution of B. In [9], the pipeline modeling with execution graphs is already supported. However, it does not consider the timing overlap between basic blocks, which is pessimistic. We adapt the Exegraph



(a) CFG (b) CFT

Figure 5: An alternative between two paths



(a) CFG (b) CFT

Figure 6: Particular case of loops

method to the CFT model as follows. First, we run the Exegraph analysis on the CFG. Then, whenever a basic block in the CFG has more than a single incoming edge, there are two possible cases:

1. *The basic block is a loop header.* In the corresponding CFT, the loop header appears twice: in the loop body (for each iteration) and in the loop exit (the last test of the loop condition, see for instance Figure 6). We take the WCET of the back-edge in the CFG for the WCET of the loop header in the loop body in the CFT. We take the WCET of the entry-edge in the CFG for the WCET of the loop header in the loop exit in the CFT.

2. *Otherwise.* In the corresponding CFT, the considered basic block is the successor of an Alt node. In that case, we add the WCET of each edge to the WCET of its source basic block.

Example 2. *In the CFG of Figure 5, the WCET of D is either* d_1 *or* d_2 *depending on the execution path. In the corresponding CFT, we set* $\omega(B) = b + d_1$*,* $\omega(D) = c + d_2$ *and* $\omega(D) = \theta$*. The result is sound since the WCET of path B.D is* $b + d_1$*, while that of path C.D is* $c + d_2$*.*

Example 3. *In the CFG of Figure 6, the WCET of H is either e or h depending on the execution path. In the corresponding CFT, we set* $\omega(H_{bd}) = h$ *and* $\omega(H_{ex}) = e$*. The total WCET of the loop is* $n \times (h + b) + e$ *(assuming n iterations).*

6 Inferring input conditionals

In this section, we explain how we extended the abstract interpretation analysis from [8] to infer the input conditionals 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.

6.1 Background

We start by recalling the main concepts of the interpretation procedure of [8]. *Abstract interpretation* [23] is a general static analysis method that infers program invariants. It propagates an *abstract state* of the program that 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 by Ballabriga et al. [8], because we want to inject the inferred invariants into our WCET analysis, which is applied to binary code. In abstract interpretation of binary code, the abstract state associated to a program location represents the set of reachable values for registers and memory addresses at this program location. The term *data location* is used to refer indistinctly to registers or memory addresses. We summarize the main features of the interpretation procedure proposed in [8] below.

Polyhedra The *abstract domain* used by an abstract interpreter defines the type of invariants it can infer. The analyzer of [8] relies on polyhedra, thus an invariant consists in a conjunction of inequalities on data locations.

Dimension-to-data-location mapping In [8], each *dimension* of a polyhedron represents the value of a data-location. The correspondence, or *mapping*, between dimensions and data-locations changes as the interpreter progresses. For instance, consider the assembly instruction `add r0, r0, #1`, which corresponds to incrementing a variable in the C source code. Suppose that, before this instruction, the dimension d_0 is mapped to register `r0`. In the polyhedron resulting from the interpretation of this instruction, `r0` is mapped to a new dimension d_1, with constraint $d_1 = d_0 + 1$. As the interpreter progresses, new dimensions are added and old dimensions that are not referred anymore are removed to reduce the size of the polyhedron.

6.2 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 dimensions that are associated to input registers. As the dimension-to-data-location mapping evolves during the interpreter progression, a dimension 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.

Example 4. *In Figure 2, we identify the dimension to which `r0` is associated in the abstract state at line 1, say d_0. Assume that the block A assigns a new value into `r0` at line 4. As a consequence, `r0` is associated to a different dimension, say d_1, in the subsequent abstract states (e.g. at the branch at line 7). The analysis identifies that d_0 corresponds to an argument of the procedure, not d_1.*

6.3 From filtering conditions to input conditionals

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[4] the resulting polyhedron over the dimensions previously identified as 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 inequations on input registers, which we call *input conditionals*.

Example 5. *In Figure 2, in the abstract state at line 7, assume that `r3` is associated to dimension d_3. Then, the filtering condition is $d_3 \geq 11$. After adding the filtering condition, the constraints at line 10 contain $d_3 \geq 11 \wedge d_3 = d_0$. The projection on procedure inputs reduces these constraints to the input conditional $d_0 \geq 11$, or equivalently `r0` ≥ 11.*

In the general case, the input conditionals are passed unchanged to the CFT builder. There are however a few particular cases to consider. First, if the projected polyhedron has no constraints (universe polyhedron), this either means that the branch condition contains no constraints on procedure arguments, or that the constraints cannot be represented by a polyhedron (e.g. a disjunction of constraints). From a WCET point-of-view, we can safely over-approximate the input conditional to *true*. Second, if the projected polyhedron has unsatisfiable constraints (empty polyhedron), the branch target is dead code, then the input conditional is set to *false*.

Finally, if the branch instruction targets a loop header, we compute a loop bound as follows. As in [8], a "ghost" register is added for each loop to count its number of iterations. It is set to 0 when entering the loop and incremented at each loop iteration. To obtain the loop bound, we project the polyhedron of the loop header over the dimensions of the procedure arguments (say $d_0, \ldots, d_3$) and

[4] To get the intuition behind a projection, think of geometric figures in a 3D space: a projection on the dimensions (x, y) of a cube in (x, y, z) is simply the geometric projection of the cube on the plane (x, y).

Commutativity

$$(e_k \wedge e_l) \circledast w_1 \mapsto (e_l \wedge e_k) \circledast w_1 \quad \text{if } e_l \lhd e_k \tag{1}$$
$$e_k \circledast w_1 \oplus e_l \circledast w_2 \mapsto e_l \circledast w_2 \oplus e_k \circledast w_1 \quad \text{if } e_l \lhd e_k \tag{2}$$
$$e_k \circledast w_1 \uplus e_l \circledast w_2 \mapsto e_l \circledast w_2 \uplus e_k \circledast w_2 \quad \text{if } e_l \lhd e_k \tag{3}$$

Factorization

$$e_k \circledast w_1 \oplus e_l \circledast w_1 \mapsto w_1 \quad \text{if } e_l \Leftrightarrow \neg e_k \tag{4}$$
$$e_k \circledast w_1 \uplus e_l \circledast w_1 \mapsto w_1 \quad \text{if } e_l \Leftrightarrow \neg e_k \tag{5}$$
$$e_k \circledast w_1 \oplus e_l \circledast w_2 \mapsto e_k \circledast (w_1 \oplus w_2) \quad \text{if } e_k \Leftrightarrow e_l \tag{6}$$
$$e_k \circledast w_1 \uplus e_l \circledast w_2 \mapsto e_k \circledast (w_1 \uplus w_2) \quad \text{if } e_k \Leftrightarrow e_l \tag{7}$$
$$e_k \circledast w_1 \oplus (e_k \wedge e_l) \circledast w_2 \mapsto e_k \circledast (w_1 \oplus e_l \circledast w_2) \tag{8}$$
$$e_k \circledast w_1 \uplus (e_k \wedge e_l) \circledast w_2 \mapsto e_k \circledast (w_1 \uplus e_l \circledast w_2) \tag{9}$$

Multiplication

$$e_k \circledast \theta \mapsto \theta \tag{10}$$
$$e_k \circledast w_1 \mapsto \theta \quad \text{if } e_k \Leftrightarrow \mathit{false} \tag{11}$$
$$e_k \circledast w_1 \mapsto w_1 \quad \text{if } e_k \Leftrightarrow \mathit{true} \tag{12}$$
$$e_k \circledast (e_l \circledast w_1) \mapsto e_k \circledast w_1 \quad \text{if } e_k \Leftrightarrow e_l \tag{13}$$

Loops

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

Figure 7: Rewriting rules with input conditionals

of the ghost register (say d_c). If we obtain a single linear inequation, we transform it so that it is of the form $d_c \leq \mathit{linexpr}(d_0, \ldots, d_3)$, and then the linear expression $\mathit{linexpr}(d_0, \ldots, d_3)$ is the loop bound. If we obtain a conjunction of inequations, we are unable to bound the loop, in which case a numeric bound must be provided by the user.

7 Symbolic WCET with input conditionals

In this section, we detail how we extend the symbolic WCET computation approach from [9] to support input conditionals.

7.1 Control-flow Tree with input conditionals

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

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

Example 6. *Figure 3 depicts the CFT obtained for the program of Figure 2. For instance, we can see that the input conditional $\mathtt{r0} \geq 11$, whose inference was detailed in Example 5, appears as an input conditional 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 7. *The node $\mathit{Loop}(l, t_1, 4 \times \mathtt{r0} + \mathtt{r1}, t_2)$ represents a loop identified by l, that executes $4 \times \mathtt{r0} + \mathtt{r1}$ times the tree t_1 and exits by executing the tree t_2.*

7.2 WCET formulas with input conditionals

We define a new operator $\circledast$ that multiplies a WCET by an input conditional. It is used to compute the WCET of an Alt node:

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

Definition 2. *Let e be an input conditional and w be an abstract WCET.*

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

Example 8. *The subtree Alt_1 of Figure 3 is translated into the formula $(\mathtt{r0} \geq 11) \circledast \omega(B) \uplus (\mathtt{r0} \leq 10) \circledast (\top, [5])$. This corresponds to $\omega(B)$ if $\mathtt{r0} \geq 11$, or to $(\top, [5])$ otherwise.*

7.3 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 [21].

7.3.1 Simplification rules

The new simplification rules for WCET formulae that contain input conditionals are detailed in Figure 7. e_k and e_l are input conditionals, w_1 and w_2 are abstract WCETs, it is an integer and l is a loop identifier. These rules are added to the rules of [9]. 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.

Property 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 conditionals. The equivalence test is detailed in Section 7.3.2. For distributivity, we rely on an order relation $\lhd$ on input conditionals (see Section 7.3.3 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 $\circledast$.

7.3.2 Testing input conditionals equivalence

Checking the equivalence of an input conditional 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 conditional is the predicate *true* or the predicate *false*.

In other cases, to test the equivalence of two input conditionals, we first put them in *normal form*. Then, equivalence amounts to a syntactic equality. An input conditional is in normal 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 parameter identifiers;
4. The last term is a constant.

Example 9. *The normal form of input conditional* $10 \geq 15 + r1 + r0$ *is* $0 \leq -r0 - r1 - 5$.

7.3.3 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 conditionals 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 conditionals is defined as follows:

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

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

Example 10. *Consider the input conditionals* $0 \leq r0 + r1 + 10 \wedge 0 \leq r2$. *We have:*

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

7.4 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 improve the performance for on-line usage, we ensured that the C compiler optimizations could be applied efficiently thanks to the following rules: 1) the resulting program is standalone, i.e. no library dependencies; 2) WCET lists are represented by several integer variables, one for each list value; 3) only simple conditional statements are allowed: no loops, no pointers and no function calls. In this way, we can easily bound the execution time of the evaluation formula, and its WCET is very low.

Note that since the WCETs are computed for any possible hardware state, the sequential execution of the instantiation WCET and the procedure code is lower (if for any reason the hardware state is favorable) or equals to $w_{instantiation} \oplus w_{procedure}$. Thus, it can be safely used as the global WCET for this procedure

8 Evaluation

In this section we present the experiments we conducted to evaluate our approach. We first detail our experimental setup, to enable the reproduction of our experiments. Then, we detail our benchmarks selection process. Finally, we provide metrics obtained by running our tool on the selected benchmarks.

8.1 Experimental setup

We implemented our approach[5] as an extension to OTAWA, an open-source WCET analysis tool [7]. We used the following hardware setup:

- Modeled processor: 1 ALU, 1 FPU, 1 MU. Integer addition costs 1 cycle, floating point addition 3 cycles, multiplication 6 cycles, division 15 cycles. It has a 4 stages pipeline (fetch, decode, execute, commit), a fetch queue of size 3, fetches 2 instructions per cycle, and executes up to 4 instructions in parallel;

[5] An artifact of our prototype can be found at `https://gitlab.cristal.univ-lille.fr/sgrebant/rtns_2023_artifact`. The repositories of the tools developped are also available:

- POLYMALYS (abstract interpretation): `https://gitlab.cristal.univ-lille.fr/otawa-plugins/polymalys`;
- WSymb (symbolic WCET computation): `https://gitlab.cristal.univ-lille.fr/otawa-plugins/WSymb`.

Table 1: Analysis time and WCET adaptivity

Procedure	IPET		CFT				
	WCET (cyc)	Time (s)	WCET (cyc)			Time (s)	
			Lowest	Highest	Diff (%)	Polyhedra	Symb WCET
audiobeam_adjust_delays	9,261	1.120	1,718	9,383	81.7	1.006	1.096
audiobeam_calc_distances	174,295	222.809	340	176,550	98.1	20.881	216.863
audiobeam_calculate_energy	303	0.242	303	303	0.0	0.099	0.246
audiobeam_find_max_in_arr	5,274	0.869	1,331	5,366	75.2	0.346	0.827
audiobeam_find_min_in_arr	5,327	0.852	1,384	5,429	74.5	0.471	0.820
audiobeam_wrapped_dec	525	0.303	490	525	6.7	0.034	0.297
audiobeam_wrapped_dec_-offset	316	0.163	281	316	11.1	0.022	0.162
audiobeam_wrapped_inc	563	0.463	528	563	6.2	0.039	0.455
audiobeam_wrapped_inc_-offset	344	0.241	309	344	10.2	0.015	0.238
cjpeg_wrbmp_write_colormap	1,266,466	7.234	1,188,091	1,288,709	7.8	113.109	7.383
fft_modff	319	0.140	319	319	0.0	0.007	0.141
g723_enc_quan	4,621	0.247	341	5,291	93.6	0.598	0.244
g723_enc_reconstruct	702	24.510	335	702	52.3	0.045	24.790
gsm_dec_APCM_inverse_-quantization	15,024	6.551	15,259	15,297	0.2	8.199	6.441
gsm_dec_APCM_quantization_-xmaxc_to_exp_mant	1,311	1.067	1235	1,353	8.7	0.184	1.033
gsm_dec_asl	855	0.495	268	855	68.7	0.059	0.484
gsm_dec_asr	420	0.272	290	420	31.0	0.028	0.266
gsm_dec_Long_Term_-Synthesis_Filtering	47,389	2.175	48,652	48,703	0.1	2.844	2.095
gsm_dec_sub	343	0.226	305	343	11.1	0.022	0.220
gsm_enc_asl	855	0.498	268	855	68.7	0.057	0.483
gsm_enc_asr	420	0.274	290	420	31.0	0.025	0.266
gsm_enc_div	5,072	0.904	3,287	5,092	35.4	0.409	0.874
gsm_enc_sub	343	0.225	305	343	11.1	0.015	0.219
lift_do_impulse	1,117	0.391	1,135	1,197	5.2	0.058	0.385
ludcmp_test	108,705	4.702	9,741	110,841	91.2	21.641	4.636
minver_minver	53,356	72.026	359	57,141	99.4	645.606	71.018
minver_mmul	12,300	1.714	380	12,492	97.0	6.300	1.640
mpeg2_dist2	134,023	9.410	134,305	134,368	0.0	37.567	9.154
ndes_getbit	383	0.381	349	383	8.9	0.035	0.357
rijndael_dec_fseek	470	0.259	380	470	19.1	0.053	0.252
rijndael_enc_fseek	449	0.212	381	449	15.1	0.057	0.204

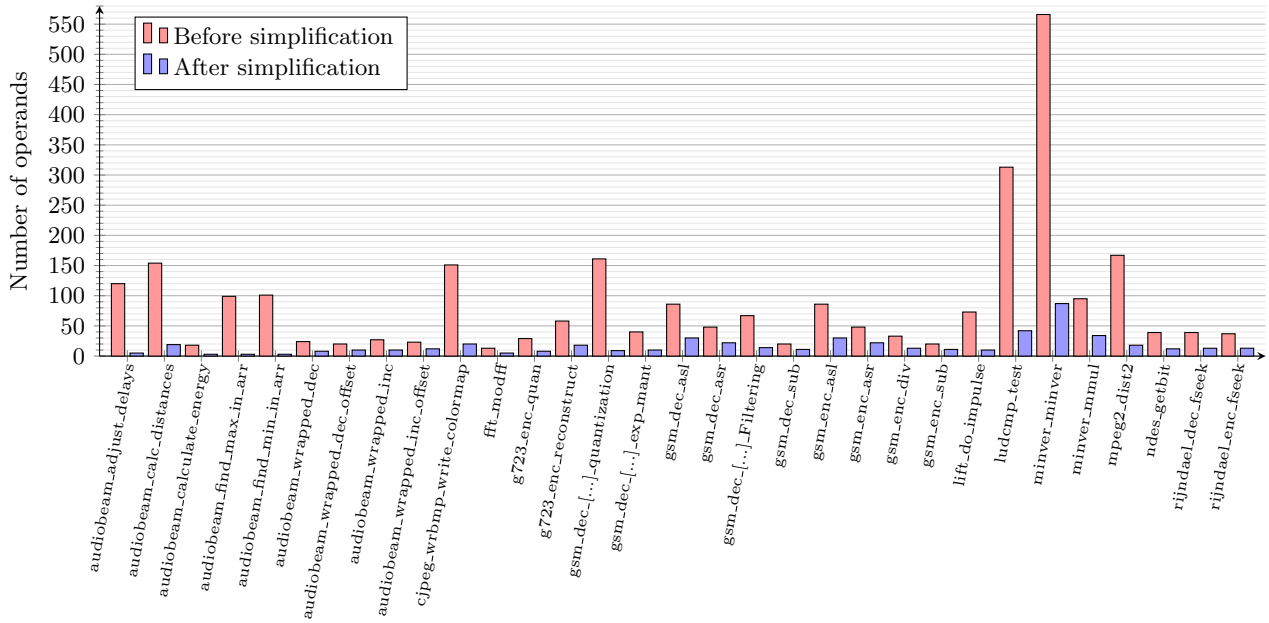


Figure 8: Parametric WCET formula size before and after simplification

- 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 -nostdinc -nostdlib -mtune=cortex-a8 -mfpu=neon -mfloat-abi=hard -g and -O0. Procedure *cjpeg_wrbmp* uses a custom memcpy implementation in order to compile with gcc, which does not compile without standard library otherwise;

- Analyses execution times: they are measured on an Intel® Core™ i7-8550U CPU @ 1.80GHz × 8 with 16 GB of RAM.

8.2 Benchmark selection

We run our experiments on the TACLeBench benchmarks suite [25]. 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*);

- Only 181 procedures have arguments, out of the 1032 procedures of the other programs;

- The polyhedra analysis only supports the integer data-type. 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 dimensions in the polyhedra.

Among the remaining procedures, we present in Table 1 the procedures for which the polyhedra analysis derived at least one input conditional. 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 other arguments.

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

8.3 Results

We compare our results with those of non-parametric IPET (from OTAWA) as a reference. We were unable to reproduce the results of the related works on parametric WCET analysis because the prototypes are unavailable, and results are insufficiently detailed to enable a comparison. Finally, related works can only analyse parametric loop bounds. Our work is the first to consider parametric conditional statements.

Table 1 summarizes our results regarding the off-line WCET analysis. The *Procedure* column contains the name of the analyzed procedure. In the *IPET* sub-columns, we report the obtained WCET (in cycles) and analysis time (in seconds). The results for our approach are reported in the *CFT* sub-columns. The *WCET* sub-columns detail the *Lowest* and the *Highest* WCETs (among a set of instantiated WCETs, see below for more details), as well as the difference between them in percentage (in the *Diff* column). The *Time* sub-columns detail the execution time of the Polyhedra analysis and of the symbolic WCET computation. The *Symb WCET* sub-column is the sum of the execution times for: symbolic WCET analysis, formula simplification, compilation of the formula into C code.

The results of the *Lowest* and *Highest* sub-columns are obtained by instantiating the formula with different argument values. They illustrate the impact of argument values on the WCET. For programs with parametric loop bounds, the maximum possible value for each loop bound is specified in TACLeBench. Thus we consider that the loop bounds can range from 0 to the TACLeBench bound. This may not correspond to realistic loop bound values, but nevertheless illustrates WCET adaptivity. For parametric conditional statements, we manually pick argument values such that we ensure that each alternative of each input conditional is inspected.

8.3.1 WCET adaptivity

For 26 out of 31 procedures, the adaptivity, i.e. the difference between the highest and the lowest WCET, is more than 5%. Many examples exhibit from 30% to 70% adaptivity, usually due to parametric conditional statements. Regarding loops, our tool supports linear loop bounds, which is not the case for related works supporting parametric loops bounds: they support only a single parameter or the sum of one parameter and an integer. However, the presented functions do not rely on bounds other than a single parameter.

The highest adaptivities (those over 90%) are exhibited when loop bounds can range down to 0, which can actually be considered unrealistic. Another case is procedure *minver_minver*, for which the lowest WCET corresponds to an unrealistic argument value: it occurs when the size of the matrix to inverse is lower than 2 or higher than 500, in which case the procedure returns immediately.

Only two functions exhibit no variability even though their WCET formula contains parameters. The *fft_modff* formula contains two alternatives, one of which has the input conditional *true* because the actual condition in the program contains a disjunction. The WCET of the

true alternative is higher than that of the other alternative, which explains the absence of adaptivity. The *audiobeam_calculate_energy* formula contains a parametric loop bound whose 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 [9] 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).

8.3.2 Analysis time

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 analyses times grow, and unexpectedly the analysis times of IPET and of 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 nesting. In some cases, the polyhedra analysis has higher execution times. This corresponds to programs with many memory accesses, which cause the polyhedra to have many dimensions and constraints. Furthermore, we also noticed that our extensions to support input conditionals 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 that extracts input conditionals. There remains a lot of room for improving 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.

8.3.3 Embeddability

The size of the initial and simplified formulae are reported in Figure 8. A simplified formula typically contains between 10 and 50 operands. Its size depends on the number of input conditionals in the non-simplified formula. The largest formula (*minver_minver*) is reduced to 15% of its initial size by our simplification procedure.

Table 2 reports instantiation times (in cycles) for a selection of procedures with various characteristics, in terms of WCET, adaptivity, 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 1. *Op* reports the number of operands in the formula, from Figure 8.

On-line instantiation can be considered only when *Max gain* is significantly larger than *Instantiation*. This is the case for most procedures of Table 1, and the difference is actually quite large. For instance, for *cjpeg_wrbmp_write_colormap*, the instantiation takes 105 cycles while there are $100,513$ cycles that 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 has no benefit.

Table 2: 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	111	100,612	1,288,709	20
g723_enc_quan	143	4,950	5,291	8
g723_enc_reconstruct	195	313	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

9 Conclusion

We presented a parametric WCET analysis that accounts for the procedure argument values effect on the control-flow of the procedure. We used abstract interpretation to extract input conditionals, that we used to produce a parametric WCET formula that depends on these arguments, opening the possibility of adaptive scheduling.

In future work, we plan to use this approach to implement a modular WCET analysis. The present work expresses the WCET of a procedure as a formula of its arguments. There remains to consider the modular aspect for both the abstract interpretation and the symbolic WCET computation.

Acknowledgement

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

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.

[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.

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

[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.

[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.

[6] Arm. Procedure Call Standard for the Arm® Architecture, 2023.

[7] 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.

[8] 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.

[9] C. Ballabriga, J. Forget, and G. Lipari. Symbolic WCET Computation. *ACM Transactions on Embedded Computing Systems*, 17(2):1–26, 2017.

[10] 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.

[11] 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.

[12] 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.

[13] E. Bini, M. Di Natale, and G. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. *Real-Time Syst*, 39(1):5–30, Aug. 2008.

[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.

[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.

[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.

[17] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings 21st IEEE Real-Time Systems Symposium*, pages 295–304, Orlando, FL, USA, Nov. 2000. IEEE.

[18] 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.

[19] 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.

[20] J. Coffman, C. Healy, F. Mueller, and D. Whalley. Generalizing parametric timing analysis. *SIGPLAN Not.*, 42(7):152–154, June 2007.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] P. Feautrier. Parametric integer programming. *RAIRO - Operations Research*, 22(3):243–268, 1988.

[27] L. George and P. Courbin. Reconfiguration of Uniprocessor Sporadic Real-Time Systems: The Sensitivity Approach. In *Reconfigurable Embedded Control Systems: Applications for Flexibility and Agility*. IGI Global, Hershey, PA, 2011. Publication Title: Reconfigurable Embedded Control Systems: Applications for Flexibility and Agility.

[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.

[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.

[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.

[31] 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.

[32] G. Lipari and S. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pages 193–200, Stockholm, Sweden, June 2000. IEEE. ISSN: 1068-3070.

[33] 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.

[34] 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.

[35] 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.

[36] L. Palopoli, L. Abeni, T. Cucinotta, G. Lipari, and S. K. Baruah. Weighted feedback reclaiming for multimedia applications. In *2008 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*, pages 121–126, Atlanta, GA, USA, Oct. 2008. IEEE. ISSN: 2325-1301.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.