École doctorale Mathématiques, sciences du numérique et de leurs interactions (MADIS-631)

Calcul efficace du pire temps d'exécution symbolique à base d'arbres
Efficient tree-based symbolic WCET computation

Thèse

Informatique et applications

Préparée par

Sandro Grebant

En vue de l'obtention du grade de

Docteur de l'Université de Lille

Soutenue publiquement le

07/11/2023

Membres du Jury

Isabelle Puaut	PR, Université de Rennes	*Rapporteure*
Matthieu Moy	MCF (HDR), Université Lyon 1	*Rapporteur*
Emmanuel Grolleau	PR, ENSMA	*Examinateur/Président*
Clément Ballabriga	MCF, Université de Lille	*Examinateur*
Julien Forget	MCF (HDR), Université de Lille	*Co-directeur de thèse*
Giuseppe Lipari	PR, Université de Lille	*Directeur de thèse*

Centre de Recherche en Informatique, Signal et Automatique de Lille

Résumé

Un système temps réel est un système informatique soumis à des contraintes de temps, et en particulier à des échéances. Les systèmes temps réels critiques sont une sous-classe de ces systèmes qui doivent impérativement respecter leurs échéances. Dans ce genre de systèmes, manquer une échéance peut avoir des conséquences catastrophiques pouvant aller jusqu'à la perte de vies humaines. Pour s'assurer que ces systèmes ne manquent jamais leurs échéances, l'analyse de pire temps d'exécution (PTE, ou WCET en anglais) calcule une limite supérieure au temps d'exécution des programmes qui doivent être exécutés.

Traditionnellement, l'analyse statique du PTE produit un entier qui représente une borne supérieure au nombre de cycles du processeur nécessaires à l'exécution de ce programme. Cependant, le temps d'exécution d'un programme peut dépendre de paramètres matériels, l'état du cache notamment, ou de paramètres logiciels, comme les paramètres d'entrée du programme. Calculer un seul PTE sous la forme d'un entier, peu importe la valeur des paramètres, est donc souvent pessimiste. Pour résoudre ce problème, l'analyse paramétrique du PTE produit une formule arithmétique qui dépend de différents paramètres choisis. Instancier cette formule avec des valeurs pour chaque paramètre permet d'obtenir un PTE plus précis qui prend en compte la valeur de ces paramètres.

Parmi les différentes analyses paramétriques, les techniques à base d'arbres montrent une complexité faible. En effet, elles utilisent une structure arborescente qui peut facilement être transformée en une formule arithmétique, ce qui permet de calculer le PTE en incluant des symboles, et donc des paramètres, de manière efficace. Néanmoins, ces approches ont aussi des inconvénients. Tout d'abord, ces méthodes sont connues pour leurs difficultés à tenir compte des effets des composants matériels sur le PTE. Ensuite, elles montrent également des difficultés à prendre en compte certains aspects sémantiques du programme, qui peuvent avoir un impact conséquent sur le PTE. Finalement, tout comme les autres analyses paramétriques, les paramètres proposés doivent être déterminés par l'utilisateur.

Dans cette thèse, nous étendons une technique de calcul symbolique du PTE pour aborder trois problèmes liés à des techniques de calcul de PTE paramétrique :

1. nous avons développé une technique qui élimine les chemins sémantiquement infai-

sables de la représentation du programme utilisée pour calculer le PTE ;

2. nous avons adapté une technique existante d'analyse de pipeline, qui fonctionne avec les techniques non paramétriques de calcul du PTE à base de graphes, afin de l'utiliser dans une technique d'analyse paramétrique du PTE utilisant des arbres ;
3. nous avons développé une technique paramétrique qui prend automatiquement en compte les effets des paramètres d'entrée du programme sur le PTE. Cette technique produit une formule dont les paramètres sont les paramètres d'entrée du programme et ne requiert aucune connaissance du programme de la part de l'utilisateur.

Mots-clés – Pire temps d'exécution, analyse statique, calcul symbolique

Abstract

A real-time system is a computer system subject to timing constraints, and in particular to deadlines. Critical real-time systems are a subclass of those systems that absolutely must meet their deadlines. In such systems, missing a deadline can have disastrous consequences and may even cause the loss of human lives. So as to ensure that such a system never misses a deadline, the worst-case execution time (WCET) analysis computes an upper bound to the execution time of the programs to be executed.

Static WCET analysis traditionally takes a program and produces an integer upper bound to the number of processor cycles required to execute this program. However, the execution time of a program may vary according to various parameters, such as hardware parameters, e.g. the state of the cache, or software parameters, e.g. input values passed to the program. Thus computing a single WCET, regardless of the parameter values, is often pessimistic. To overcome this issue, parametric WCET analysis produces an arithmetic formula that depends on various chosen parameters. Instantiating this formula with actual parameter values enables to produce a more precise WCET that takes into account these parameters.

Among the different parametric techniques, tree-based WCET computation techniques have a low complexity. Indeed, they rely on a tree structure that can easily be transformed into an arithmetic formula, which enables to handle symbols, and thus parameters, efficiently. Nevertheless, these approaches also exhibit some drawbacks. First, they struggle to take the effect of the hardware components on the WCET into account. Second, they also struggle to take some aspects of the program semantics into account, that can have a big impact on the WCET. Third, as other parametric WCET analysis techniques, the proposed parameters must be determined by the user.

In this thesis, we extend a symbolic WCET computation technique, to tackle three issues with parametric tree-based WCET computation techniques:

1. We developed a technique that eliminates program paths that are infeasible, due to the program semantics, from the program representation used to compute the WCET;

2. We adapted an existing pipeline analysis, that works with graph-based non-para-

metric WCET analysis, to use it in a parametric tree-based WCET analysis;

3. We developed a parametric technique that automatically takes into account the effect of the program input values on the WCET. This technique produces a formula whose parameters are the argument values passed to the program and it does not require the user to have any knowledge about this program.

Keywords – Worst-case execution time, static analysis, symbolic computation

Remerciements

Cette thèse présente le fruit de trois années de travail et bien que mon nom soit le seul qui y apparaisse en tant qu'auteur, ce document n'aurait probablement pas pu aboutir sans de nombreuses personnes que je souhaite remercier dans ce chapitre.

Pour commencer, j'aimerais remercier les personnes ayant accepté de participer à mon jury. Je tiens tout d'abord à exprimer toute ma reconnaissance envers Isabelle Puaut et Matthieu Moy, rapporteurs, qui ont pris le temps de relire une première version de ce document et de me faire part de leurs remarques constructives et pertinentes, me permettant de ce fait d'améliorer le contenu ainsi que la présentation de cette version finale. Je souhaite également remercier chaleureusement Emmanuel Grolleau, examinateur, qui a pris le temps de lire cette thèse et de présider le jury lors de la soutenance.

Je souhaite également remercier les différentes personnes ayant participé à l'encadrement de ma thèse. Clément Ballabriga a non seulement co-encadré ma thèse, mais m'a aussi beaucoup appris sur Otawa. Julien Forget, co-directeur de ma thèse, s'est considérablement impliqué dans mon encadrement et m'a initié aux principes de la rédaction scientifique. Enfin, je tiens à exprimer ma gratitude envers Giuseppe Lipari, mon directeur de thèse, dont les conseils et l'expérience ont souvent permis de solutionner les problèmes rencontrés.

Je souhaite à présent remercier l'équipe SyCoMoRES de CRIStAL ainsi que les anciens doctorants de l'équipe, en particulier pour leur accueil et la bonne ambiance qu'ils ont su instaurer dans mon environnement de travail.

Enfin, je tiens à exprimer ma reconnaissance envers toutes les autres personnes, sans nul doute trop nombreuses pour être individuellement mentionnées, qui ont directement ou indirectement contribué à ma thèse, je pense notamment à ma famille et à mes amis. J'aimerais tout de même remercier particulièrement Flore-Anne Grebant, ma sœur, qui a relu les parties françaises de ce document.

Contents

Chapter 1

Introduction

Contents

This chapter introduces the topic of the thesis. We first contextualize this thesis with a general background. We then present the motivations for this work. We end this chapter by introducing the different contributions of the thesis.

1.1 Context

We begin by introducing some basic notions and definitions regarding the thesis context, so as to emphasize the crucial aspect of the execution time of a program.

1.1.1 Embedded systems

An embedded system is a computer system that is embedded within a physical device. It is designed such that it has a specific function in that device. Many devices that we use everyday rely on embedded systems: televisions, cars, printers and even fridges are common examples of such devices. In fact, most of the microprocessors that are currently sold are used in embedded systems. This demonstrates the importance of such systems in our society. A typical example of embedded system is the personal navigation assistant (PNA) of a car: it is embedded in a car and its function is to indicate to the driver how to to get from his current location to another one.

1.1.2 Real-time systems

A real-time system is an embedded system subject to real-time constraints, typically deadlines. We distinguish two kinds of real-time systems: *soft* real-time systems and *hard* real-time systems.

Soft real-time systems are systems in which we assume that it is not crucial to always meet the deadlines. For instance, the music player software in a car is not critical. However, it is better if the music plays fluidly and the loading time between two tracks is not too long. Nevertheless, if the loading time between two tracks is a bit longer than expected, it does not have any consequence on the car safety.

Hard or critical real-time systems are systems that must always meet their deadlines. Most of the time hard real-time systems safety critical and meeting the deadlines is crucial. For instance, consider the autonomous emergency braking system of a car. If the system detects that something is in the way, it must activate the brakes fast enough to avoid collision. Activating the brakes too late can have disastrous consequences for the driver and the passengers in the car.

A real-time system is often decomposed into several tasks, which perform various operations. Each task should complete before its own deadline to optimally execute the

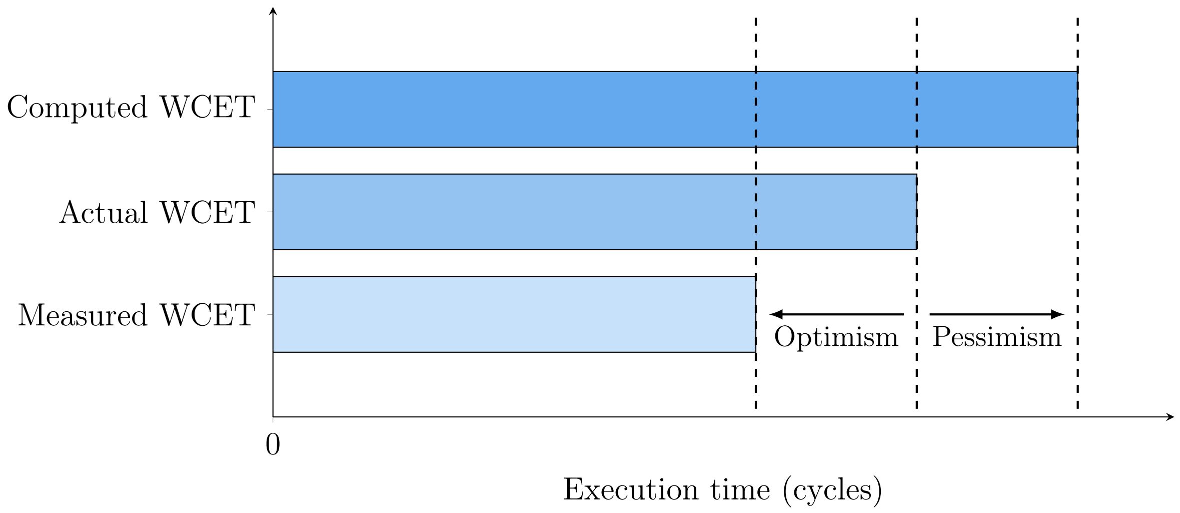


Figure 1.1: Approximation of WCET

system. In order to check that the system is *schedulable*, which means that all the tasks in the system meet their deadlines, a *schedulability analysis* is performed. This analysis takes as inputs all the tasks that are executed on the system as well as their execution times.

1.1.3 Worst-case execution time

Depending on the current state of the system, the behavior of a task may vary and thus its execution time may not be constant. To perform the schedulability analysis, an upper-bound to the execution time of the task must be provided: the worst-case execution time (WCET) of the task, counted in processor cycles[1]. In practice, it is most of the time not possible to obtain the exact WCET for a program. Indeed, obtaining the exact WCET for any task would also solve the halting problem[2]. However, it is possible to determine an approximation of the WCET of a program (or task) assuming that [Wil+08]:

1. There is no recursion in the program, or the depth of the recursion must be bounded;
2. The maximum number of iterations of the program loops is known.

The *WCET analysis* is performed to derive an approximate value to the WCET of a program. There exists three kinds of methods, all with their advantages and drawbacks,

[1]The time in a processor is measured in processor cycles. During each cycle, transistors within the processor open and close. The speed of a processor, also called clock speed, is often expressed in megahertz (MHz) or gigahertz (GHz) and is the number of processor cycles that are executed in the processor at each second.

[2]The halting problem is a well-known problem in computer sciences, which is to determine whether a program will terminate or not. This problem is undecidable, which means that there is no algorithm that is able to say, for any program, if this program will end.

to obtain a WCET: *measurement-based* approaches, methods based on *static analysis* and *hybrid* techniques.

1.1.3.1 Measurement-based analysis

A first class of approaches are *measurement-based* analyses. These consist in measuring the execution time of the program repeatedly, under different initial conditions. It requires to execute the program on the processor that will be used to run the system, and easily produces realistic values. Nevertheless, it is difficult to ensure that the set of tests used is exhaustive. Thus, the resulting WCET approximation can be underestimated, which is problematic for hard real-time systems. It also requires to use the actual hardware that will execute the system to perform measurements. In Figure 1.1, the resulting value of this kind of approaches is represented by the "Measured WCET".

1.1.3.2 Static analysis

A second category of methods are based on *static analysis*. These techniques examine the code of a program so as to derive its WCET. The analysis consists of two steps. First, the analysis represents the *control-flow* of the program, most of the time with a graph or a tree, to represent all the possible program executions, also called *program paths*. Then, the WCET of each node of the tree or graph is computed using a hardware model, that represents the system on which the program should be executed. The WCET is computed by combining the results of theses two steps. This technique has the advantage to derive a *safe* WCET bound, which means that the computed WCET cannot be lower than the actual WCET. Since it uses a hardware model, it does not require the actual hardware to perform the analysis. However, this category of methods also suffers from different problems. First, the computed WCET is often overestimated, because we may consider program executions that are infeasible in practice. Second, a precise hardware model is difficult to design: many hardware feature behaviors are very difficult to predict statically, which leads to pessimism. In Figure 1.1, the resulting value of such analyses is represented by "Computed WCET".

1.1.3.3 Hybrid WCET analysis

The last category of techniques tries to benefit from the advantages of the two other categories of approaches. It is hybrid in the sense that it uses static analysis for the control-flow, and often relies on measurements to estimate the WCET of nodes of the graph or tree. The WCET can also be estimated using other techniques, e.g. machine learning.

1.1.4 Parametric WCET analysis

The WCET of a task can exhibit large variability related to various hardware parameters, e.g. the state of the cache, and software parameters, e.g. the input of a procedure. Parametric WCET analysis, instead of computing a single numerical value for the WCET of a task, produces an arithmetic formula that depends on such parameters. This technique can be used for programs that contain values that may impact the WCET but are unknown statically or may change depending on the state of the system. This thesis mostly focuses on enhancing parametric WCET analysis.

1.2 Motivations

In this section, we exhibit the limitations of the current parametric WCET analysis approaches and state our objectives.

1.2.1 Limitations of current approaches

The first parametric approaches were inspired by regular static analysis approaches. The implicit path enumeration technique (IPET) [LM95], which relies on integer linear programming (ILP), is the predominant technique to compute the WCET. Since this technique has been extensively studied, it supports a wide range of hardware and software facts, most of the time represented with linear constraints[3]. The first parametric approaches tried to extend this technique to support parameters using parametric integer programming (PIP) [Fea88], which seemed to be a solution to keep the advantages of IPET to compute a parametric WCET. However, this approach is computationally very expensive and works only with very small programs.

Other techniques have been developed, that do not rely on PIP. However, they suffer from several limitations:

1. Less techniques have been developed to model hardware and software features;

2. These approaches often lack diversity regarding the kind of parameters they support: most techniques focus on loop bounds as parameters or consider that parts of the program can have an unknown WCET. Most of the time, the user must identify the parameters and their impact on the compute WCET;

3. The supported parametric expressions are quite simple. For instance, in all the works that have been conducted on parametric WCET so far, the most powerful expression

[3]In integer linear programming, a linear constraint is a linear inequality

supported for a loop bound is a sum between a parameter and a constant, which is insufficient to express the impact of some parameters on the program WCET.

1.2.2 Starting point and objectives

In this work, we start from the tree-based parametric WCET approach proposed by Ballabriga et al. [BFL17]. Tree-based techniques have two advantages:

1. A low complexity, i.e. polynomial in the size of the tree;

2. Since WCET computation is performed inductively on the tree structure, it is rather simple to support parameters.

The objective of this work is to extend this tree-based technique to contribute on the three drawbacks previously enumerated:

- Hardware and software features modeling;
- Diversity of parameters;
- Expressivity of the parametric expressions.

1.3 Contributions

In this section, we present the various contributions of this work.

My first contribution concerns software features. It focuses on *infeasible paths*. An infeasible path is a sequential execution of several nodes of the program that is structurally feasible in the tree but that is infeasible if we take into account the program semantics. I present an algorithm that removes the infeasible paths from the tree-based control-flow representation.

My second contribution is related to hardware features. The pipeline processes several instructions in parallel by splitting their execution into different steps, which generally reduces the execution time of a program. I propose an approach that adapts the pipeline modeling technique of Rochange et al. [RS09] to the tree-based program representation.

My third contribution is concerned with a better support for software parameters. For many programs, the variations of the WCET depend on the arguments passed to the program (or to a procedure in that program). I propose to analyze the program to automatically infer the impact of the program input on the control-flow of the program. This technique combines abstract interpretation with symbolic WCET computation to produce a parametric formula whose parameters are the program inputs.

1.3.1 Related publications

- An outline of the thesis objectives and the foundations of our algorithm for infeasible paths in tree-based WCET computation was presented in [GBF21];
- The pipeline model adaptations as well as the program inputs support have been published in [Gre+23];
- Some related new challenges regarding adaptive real-time systems were presented in [Bal+23].

1.4 Thesis outline

We begin with a presentation of the state of the art regarding static WCET analysis in Chapter 2. Chapter 3 presents our algorithm for infeasible paths representation in tree-based WCET analysis. Then, our adaptations of the pipeline analysis to tree-based WCET computation are presented in Chapter 4. We continue by detailing how to use procedure arguments as parameters of the WCET analysis in Chapter 5. Finally, we conclude the thesis in Chapter 6.

Chapter 2

Static WCET analysis: state of the art

Contents

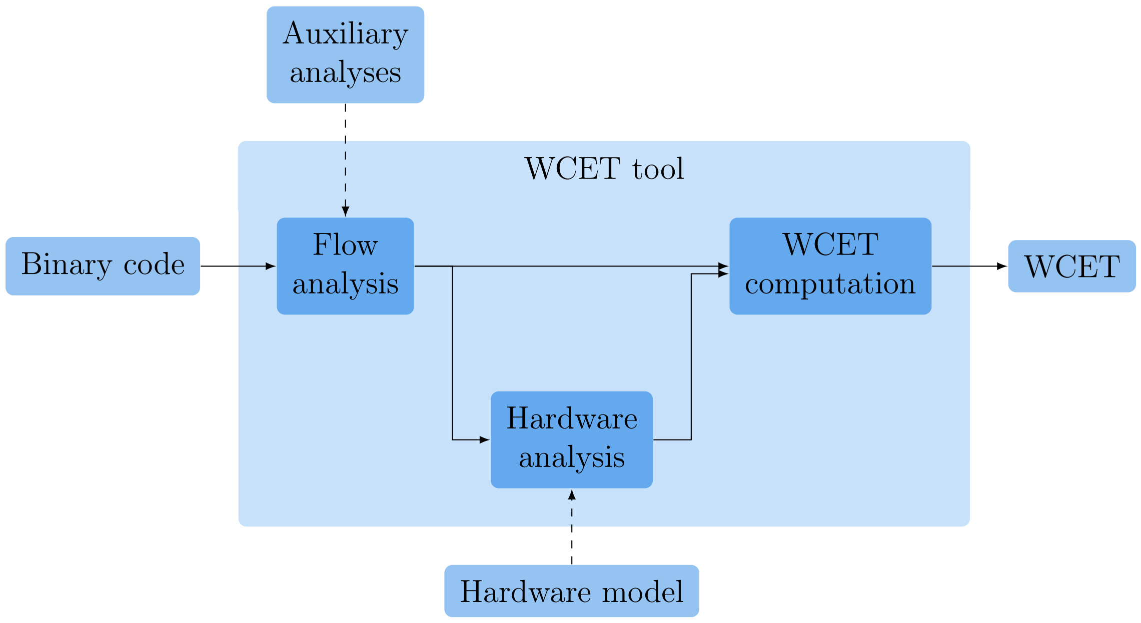


Figure 2.1: General approach to static WCET computation

In this chapter, we present a state of the art regarding static WCET analysis. Indeed, since parametric WCET analysis is based on static WCET analysis, and thus so as to understand how and why we developed our contributions, it is essential to review the existing techniques and works related to static WCET analysis. To discover more techniques that are not all based on static analysis, a good survey by Wilhelm et al. [Wil+08] recaps most of the WCET analysis works.

2.1 General framework

We begin with the general framework of static WCET analysis. This framework is depicted in figure 2.1.

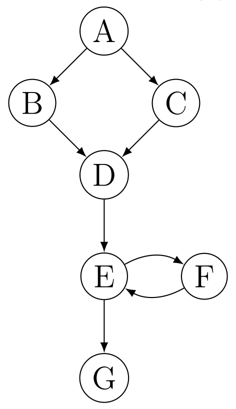
Most recent approaches rely on *binary code*, that is to say the code of the program that is actually executed on the system. Some older approaches rely on the source code, which is the code written by the developer. However, the execution time of the source code may be hard to predict since the compiler performs a lot of optimizations.

With this code, the WCET computation tool first performs a *flow analysis*, which produces a program representation that represents all the *execution paths* of the program. Each execution path is defined as a succession of basic blocks that leads to a complete execution of the program.

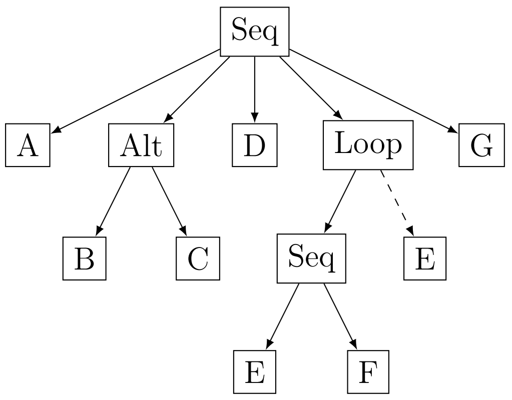
The flow analysis often rely some external information, provided by some *auxiliary analyses*. For instance, some tools can analyze the program to find an upper-bound to the number of iterations of loops. We present the different works on the flow analysis in

```
void f(int a, int b){
  // A
  if(a > 10)
    // B
  else
    // C
  // D
  for(int i=0; i<4; i++) // E
    // F
  // G
}
```

(a) A procedure with its basic blocks



(b) Control-flow graph of the procedure



(c) Control-flow tree of the procedure

Figure 2.2: A program with some of its control-flow representations

Section 2.2. Some auxiliary analyses are presented in section 2.5. Our work in Chapter 3 uses the results of the analyses presented in Section 2.5.2

Then, the *hardware analysis* models the behavior of the processor in order to estimate the execution time of each basic block. It relies on a *hardware model*, that describes the architecture of the processor. We introduce various hardware analysis methods in Section 2.3. Our work in Chapter 4 relies on the feature presented in Section 2.3.1.

Finally, the analysis uses both the program representation produced by the flow analysis and the timing estimations of the hardware analysis to perform the *WCET computation*. Various approaches to WCET computation are presented in Section 2.4.

In the remainder of this chapter, we detail each of these analyses. We also present some parametric WCET analyses in Section 2.6 and a background about the parametric WCET computation technique that we use in this thesis is presented in Section 2.7.

2.2 Flow analysis

We first concentrate on the flow analysis. This analysis is performed to produce a representation of the control-flow of the program, that is to say to represent all the possible execution paths in a program.

2.2.1 Basic blocks

First of all, to represent those paths, the program must be cut into different pieces. The flow of the program changes because of *branching instructions*. In essence, a program is a sequence of instruction executed in order. However, a branching instruction can change the control-flow to jump to an instruction that is not the next instruction in the sequence.

Thus, to represent the control-flow of the program, that is to say the sequences of instructions that are possible to execute in a program, we cut the program into small parts, called *basic blocks*.

Definition 2.1 (Basic block) *A basic block is a sequence of instructions with a single entry point and a single exit point, such that if the first instruction of this sequence (the entry point) is executed, then all the other instructions in this sequence must be executed before executing any other instruction of the program.*

In other words, a basic block is a sequence of instructions, where a branching instruction, that is to say an instruction that have different possible successor instructions, can only be the last instruction of this block. These basic blocks enable to represent the different execution paths in a program.

Definition 2.2 (Execution path) *An execution path is a sequence of basic blocks that represents one complete execution of a program. We denote "." the concatenation operator for sequences.*

Example 2.1 (Basic blocks and execution paths) *Consider Figure 2.2a, which is a simple function in C. The comments represent the different basic blocks of the program: A, B, C, D, E, F and G. The program contains several execution paths, e.g. A.B.D.E.G.*

2.2.2 Flow representation

The flow representation is an abstract representation of the program, that represents the different execution paths that are possible in a program. We now present the two main representations.

2.2.2.1 Control-flow graph

The most common representation is the *control-flow graph* (CFG) [All70], which represents the control-flow of a program with directed edges between basic blocks. More formally, a control-flow graph is a graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of vertices that correspond to basic blocks and $\mathcal{E}$ is a set of directed edges between these basic blocks.

Each directed edge $(X, Y) \in \mathcal{E}$ between the basic blocks X and Y in the CFG indicates that, after executing the basic block X, it is possible to execute the basic block Y.

Example 2.2 (Control-flow graph of a program) *Consider Figure 2.2b, which represents the CFG of the function f of Figure 2.2a. This control-flow graph indicates that we first execute A, then either B or C before executing D. After that, we execute E, and then we can either enter the loop and execute F and E repeatedly, or exit the loop and execute G.*

A path in a control-flow graph is a sequence of basic blocks $B_1, B_2, \ldots, B_n$ such that $\forall x \in [1, n-1] : (B_x, B_{x+1}) \in \mathcal{E}$. Less formally, a path in a control-flow graph is a sequence of basic blocks such that there exists an edge in the control-flow graph between each pair of successive basic blocks in the sequence.

Example 2.3 (Path in control-flow graph) *Consider Figure 2.2b. In this graph, the execution path of Example 2.1, $A.B.D.E.G$, is valid. However, a sequence of basic block like $A.D.G$ is not a path in the CFG.*

2.2.2.2 Tree-based representations

Another way to represent the control-flow of a program is to use a tree. Tree-based representations are very similar to CFG, albeit with a tree structure. In general, tree-based representations are composed of several kinds of trees:

- Leaves, which are the basic blocks of the program;
- Sequences, which represent the execution in sequence of several trees;
- Alternatives, which represent the execution of a single tree among several trees;
- Loops, which represent the repeated executions of the loop test and the loop body;
- Calls, which enable to separate the trees of several functions.

The *syntax tree* representation [Sun+98; CP00; CP01; BB06] is the first tree-based representation that was used in WCET analysis. The specificity of this representation is that it enables to compute the WCET inductively on the tree structure. Most other tree-based control-flow representations are similar to the syntax tree. In our work, we use the *control-flow tree* representation from [BFL17].

Example 2.4 (Control-flow tree) *Consider Figure 2.2c, which represents the control-flow tree of function f of Figure 2.2a. This tree uses all the kinds of nodes, except the call node. We can see that the root tree, a.k.a. Seq, is a sequence that indicates (as in the CFG) that in the function we first execute A, then either B or C before executing D. After that, we execute E and F repeatedly before exiting the loop after executing the last loop test E to execute G (the Loop dashed edge indicates the tree that represents the exit tree of the loop, which is the last loop test).*

2.2.2.3 Auxiliary control-flow information

The flow representation of the program is crucial to analyze the different execution paths of the program. However, this analysis alone is not enough to fully represent the control-flow of the program: since the control-flow representation is an abstraction of the program, some information are lost during the analysis.

A first piece of information required to compute the WCET are loop bounds: if there is no upper-bound to the number of iterations of loops, then the number of paths in the flow representation is unbounded.

Another missing piece of information is the presence of *infeasible paths* in the program representation.

Definition 2.3 (Infeasible path) *An infeasible path is a path that is feasible in the program representation but that is not actually feasible in the program due to the program semantics.*

Example 2.5 (Infeasible paths) *Consider function f in Figure 2.2. If function f is never called with a value such that $a > 10$, then all the paths that pass through the basic block B are infeasible. However, those paths remain feasible in both the control-flow graph and the control-flow tree representations.*

To overcome these shortcomings, several auxiliary analyses can be used to add supplementary information from the program into the flow representation. Of course the developer could specify information like loop bounds by himself to the tool with an *annotation language* [Kir+11], but it involves that the user has knowledge about the program under analysis. Furthermore, for big programs it can be a tedious work, which is prone to errors. Some auxiliary analyses, that are not directly related to the WCET computation, can be used to address this problem and are presented later in Section 2.5.

Now that we represented the control-flow of the program, we will focus on the WCET estimation. First, we will focus on the WCET at the basic block level, computed by the hardware analysis. Then, we will show how the WCET of these basic blocks can be used to compute the WCET of a whole program.

2.3 Hardware analysis

In this section, we focus on the hardware analysis. This analysis models the behavior of the the computer hardware that will execute the system, such that we can determine the execution time of each basic block of the program representation. So as to produce realistic results, three kinds of hardware components are often considered:

- The pipeline, which decomposes the execution of each instruction in several *stages* so that several instructions can be executed at the same time by the processor;
- The instruction and data caches, which respectively speed up the access to the next instructions of the program and to the data used by these instructions;
- The branch predictor, that tries to predict which instructions are executed after a branching instruction.

The remainder of this section presents various techniques that model the effect of these hardware components to produce precise WCET for each basic block of the program.

2.3.1 Pipeline

We start with the pipeline. In a processor, the pipeline enables to process several instructions in parallel at the same time by splitting the processor into several *stages*. A stage manages a particular processing phase of an instruction. The ARM7 pipeline, which is a great example of embedded processor pipeline, is composed of three stages:

1. Fetch (FE), which reads the instruction from the memory;
2. Decode (DE), which decodes the instruction;
3. Execute (EX), which executes the instruction.

Since it is possible to have a different instruction in each stage, this processor can execute several instructions in parallel, which impacts the WCET [Eng02].

Example 2.6 (Pipelining effect) *Consider Figure 2.3. Figure 2.3a details the instructions that are executed in the pipeline, whereas Figure 2.3b represents the execution of these*

```
ldr r2, [fp, #-12]   @  Load a value from the memory
add r3, r3, #1       @  Increment a register
str r2, [fp, #-16]   @  Store the loaded value in memory
```

(a) Instructions sample

(b) Pipelined execution

Stage \ Cycle	1	2	3	4	5	6	7	8	9
Fetch	░	▒	▓	▓	▓				
Decode		░	▒	▒	▒	▓			
Execute			░	░	░	▒	▓	▓	▓

ldr	░
add	▒
str	▓

Figure 2.3: Arm instructions and the matching pipeline execution

instructions in each pipeline stage. We consider here that each stage takes one cycle to execute, except the execute stage, which takes three cycles when a memory access occurs. If we look at each instruction independently, the first and the third instructions take 5 cycles each to execute, while the second instruction takes 3 cycles to execute. Without the pipeline, this would take 13 processor cycles to execute the three instructions. However, thanks to the pipelining effect, Figure 2.3b shows that the execution of the three instructions takes 9 cycles only and thus saved 4 processor cycles.

2.3.1.1 Pipeline modeling

Numerous approaches detail how to model the processor pipeline in order to take its effect into account during the WCET computation. Some approaches target a single processor model [ZBN93; HWH95; Lim+95], while other techniques present more general frameworks that can be used to model various pipelines. We present some of these general frameworks here.

Abstract interpretation Different approaches propose to use abstract interpretation so as to model the pipelining effect on the WCET [SF99; LTH02; HRW15]. We illustrate the use of abstract interpretation with the approach of Schneider et al. [SF99]. This approach considers the pipeline as a set of stages $S = \{s_1, \ldots, s_m\}$ as well as a set of resources $R = \{r_1, \ldots, r_m\}$ of the processor. For each instruction, the set of required resources at each stage of the pipeline is specified, and depends on the instruction type (e.g. `add`) and on the operand types (e.g. register, constant value). The allocation of a resource to an instruction is modeled by a sequence of pairs $(s, \mathcal{R} \subseteq R)$, where s is the stage and $\mathcal{R}$ is the set of resources used to process the instruction in that stage. Each pair in the sequence represents the execution of the instruction in the pipeline during a single processor cycle.

Example 2.7 (Pipeline: Abstract instruction representation) *Consider i, an instruction whose execution is represented by the sequence $(s_1, \{r_1, r_2\}).(s_2, \{r_3\}).(s_2, \{r_3\})$. This sequence means that this instruction first occupies s_1 and use resources r_1 and r_2 for one cycle, then occupies s_2 and uses the resource r_3 during two cycles.*

A concrete state p of the pipeline represents the occupancy of the pipeline stages by instructions, as well as the current and future resource allocations for these instructions. The state of some special resources like the prefetch queue are also indicated. Let IS denote the instruction set of the processor and P denote the set of concrete pipeline states. The update function $\mathcal{U} : P \times IS \rightarrow P$ updates the state of the pipeline each time an instruction enters the pipeline. Its implementation depends on the modeled processor. The cycle function $\mathcal{C} : P \times IS \rightarrow \mathbb{N}_0$ computes the number of processor cycles needed by a new instruction to enter the pipeline, i.e. the number of cycles needed to reach $\mathcal{U}(p, i)$. The empty function $\mathcal{E} : P \rightarrow \mathbb{N}_0$ computes the number of processor cycles needed to reach the empty pipeline state, i.e. to finish the execution of the instructions in the pipeline.

An abstract state of the pipeline $\hat{p}$ is defined as the set of all the possible concrete pipeline states at a program point. The three functions defined for concrete states can easily be adapted for abstract states, as $\hat{\mathcal{U}}(\hat{p}, i) = \{\mathcal{U}(p, i) | p \in \hat{p}\}$, $\hat{\mathcal{C}}(\hat{p}, i) = max(\{\mathcal{C}(p) | p \in \hat{p}\})$ and $\hat{\mathcal{E}}(\hat{p}) = max(\{\mathcal{E}(p) | p \in \hat{p}\})$.

When a program point has several predecessors, a join function $\hat{\mathcal{J}}(\hat{p}_1, \ldots, \hat{p}_n) = \bigcup_{i=1}^{n} \hat{p}_i$ is used to produce a single abstract state from the abstract states of the predecessors.

With all these functions, we can derive the execution time of the program by performing the abstract interpretation on the whole CFG.

Execution graphs Another method to model the effect of the pipeline is to use *execution graphs* [LRM04; LRM06; Bar+06; RS09]. Since the execution time of a basic block depends on the pipeline state at the beginning of the execution of the basic block, the approach proposes to represent the WCET of a basic block as a function of its execution context. The *execution cost* of a basic block is defined as the time between the completion of the last instruction preceding this basic block and the completion of its last instruction. The execution cost of the first basic block of the program is equal to its execution time.

An execution graph is used to model the execution of a basic block. Each node of this graph represents the processing of an instruction by a pipeline stage or unit. Directed edges in this graph express a precedence constraint. Different kinds of precedence constraints can be represented:

- the program order, which defines the order in which the instructions are fetched;
- the structure of the pipeline, that orders the stages of the pipeline;

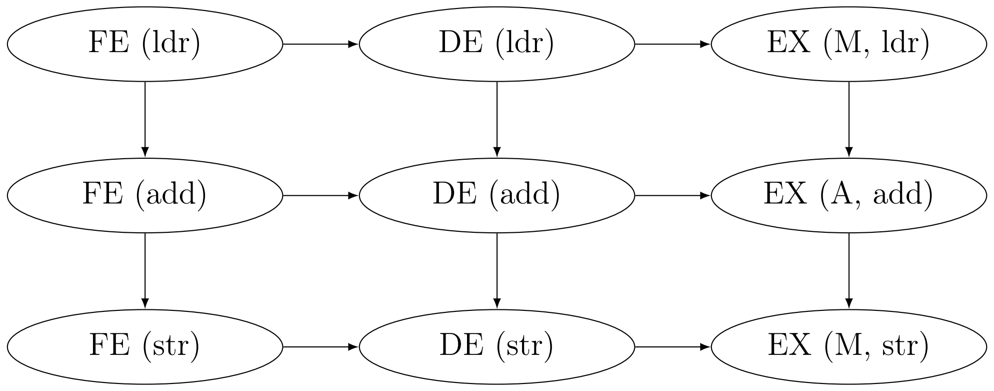


Figure 2.4: Execution graph of the instructions of Figure 2.3a

- the capacity of the instruction queues;
- the data dependencies.

Example 2.8 (Execution graph) *Consider Figure 2.4, which represents the execution graph that corresponds to the program of Figure 2.3a on a scalar in-order pipeline. On this graph, the path of each instruction through the pipeline is depicted: each instruction passes through the three stages of the pipeline. For the execute stage,* M *indicates that the instruction uses the memory processor unit (MEM) while* A *indicates that the instruction uses the arithmetic and logic unit (ALU). Two kinds of precedence constraints are represented here:*

1. *the horizontal directed edges represent the precedence constraints due to the pipeline structure;*
2. *the vertical edges represent the precedence constraints related to the execution on a* scalar in-order *processor. Scalar means that the processor has a single pipeline. In-order means that this processor executes the instructions in the same order as in the program code.*

Note that the data dependency between the load and the store on r_2 is not represented since the scalar in-order nature of the pipeline already guaranties that the load is executed before the store.

For each node n, a latency l_n indicates the execution time of the instruction in the specified pipeline stage. The node *ready time* of each node, that depends on the end of the execution of the preceding nodes, is denoted ρ_n. The end of each node execution (its completion moment) is defined as $\tau_n = \rho_n + l_n$.

So as to compute ρ_n, the context of the basic block has to be specified. Let $\mathcal{R}$ be the set of resources of the processor (in particular its functional units), then for each node the context is specified by:

- a vector of parameters $\mathcal{A} = \{a^r | r \in \mathcal{R}\}$, which corresponds to the release time of a resource in $\mathcal{R}$;
- a vector $\mathcal{E}_n = \{e_n^r \in \{0, 1\} | r \in \mathcal{R}\}$ of booleans, that represents the dependence of the node ready time to each of the context parameter;
- a vector $\mathcal{D}_n = \{d_n^r | r \in \mathcal{R}\}$ of delays, that represents the minimal distance between the release time and the node ready time.

Then, the node ready time is computed as $\rho_n = max(e_n^r \times (d_n^r + a^r))$. e_n and d_n are then propagated from predecessors to successors in the graph. After the propagation, the WCET of the basic block B can be expressed as a function of its execution context: $max(\{\tau_{last(B)} - \tau_{last(x)} | x \in predecessors(B)\})$, where $last(X)$ is the last node of the last instruction of the basic block X.

In practice, considering the complete list of predecessors for a basic block in the context would lead to a very high complexity. Thus, the technique presented in [RS09] proposes to consider only a small number of basic blocks in the execution context to compute the WCET of a basic block. The authors also demonstrated that considering such contexts leads to a safe WCET.

2.3.2 Cache

Caches also have a big impact on the WCET of programs. Indeed, caches reduce the average access time to data (for the data cache) or to instructions (for the instruction cache). For instance, when the processor loads an instruction from memory, the cache stores a small part of the memory, called a *block*, that contains this instruction as well as several other instructions. Thus, when the processor use another instruction that is located in the same block, the cache will provide it much faster than the memory, which reduces the execution time.

In essence [TFW00], a cache is defined by:

- its *capacity*: the maximum number of bytes that it can contain;
- its *block size* (or line size): the number of bytes transferred from the memory when a block is added into the cache;
- its *associativity*: the number of cache locations where a given block can reside.

With this, the number of blocks that the cache can contain is $n = capacity/block\ size$, and the number of *sets* is defined as $n/associativity$.

If a block can reside in any cache location, the cache is said to be *fully associative*, which means that the cache has only one set. At the opposite, if a block can reside only

Age	Block
0	D
1	C
2	B
3	A

$\xrightarrow{+E}$

Age	Block
0	E
1	D
2	C
3	B

(a) Replacement: new block in the cache

Age	Block
0	D
1	C
2	B
3	A

$\xrightarrow{+B}$

Age	Block
0	B
1	D
2	C
3	A

(b) Replacement: block already in the cache

Figure 2.5: LRU cache replacement policy

in a single location, the cache is said to be *direct mapped*, which means that the number of sets in the cache corresponds to the number of blocks that the cache can contain. For the sake of clarity, we will only consider fully-associative caches for the explanations.

When the cache is full and that a new block is added to the cache, a *replacement policy* removes one of the block in the cache to store the new block. There exist many different replacement policies, which detail which block is evicted from the cache when it is full:

- LRU: the least recently used block;
- MRU: the most recently used block;
- FIFO (first in first out): the first block that entered and is still in the cache;
- RANDOM: a randomly chosen block.

Generally, the LRU replacement policy is used for WCET analysis. Even if it is not commonly used in caches, it is more predictable than the other policies, which makes it easier to analyze without involving too much pessimism in the computed WCET.

Example 2.9 (LRU cache replacement policy) *Consider Figure 2.5a. In this figure, the array on the left represents the state of the cache before adding E to the cache. When E is added, the cache is full and thus the block with the highest age (A) is evicted. The age of all the other blocks is incremented and E is added with the age 0.*

When a memory access occurs, two events can happen: if the accessed element is not in the cache, a *cache miss* occurs, which means that the accessed element has to be retrieved from the memory. At the opposite, if the accessed element is in the cache, a *cache hit* occurs and the element is retrieved faster from the cache.

In the particular case of the data cache, we must consider a writing policy among:

- *write-through*, which means that every time a write access is performed, it is written immediately in the main memory. Generally, the modified element is also stored into the cache, which is called *write allocate* policy. If the element is not stored into the cache during the write access, it is called a *no write allocate* policy;

- *write-back*, which means that the changes are buffered, marking a cache block as dirty. These changes are then written back into the main memory when the dirty block is evicted from the cache.

2.3.2.1 Cache analysis

Various cache analyses has been developed for WCET analysis. The first techniques modeled the instruction cache with static simulation [MW95] or using cache conflicts graphs [LMW95; LMW96]. However, more recent methods [FW99; SS07; SR10; HJR11; FGG18; Tou+19] based on abstract interpretation enable to represent both the instruction cache and the data cache. We illustrate the principle of these techniques with the approach of Ferdinand et al. [Alt+96; FW99; TFW00]. This analysis works on the control-flow graph, assuming that for each basic block the sequence of memory accesses is known. In this approach, the cache is modeled as a set of cache lines $L = \{l_1, \ldots, l_n\}$, and the elements stored in the cache are represented as a set of memory blocks $S = \{s_1, \ldots, s_m\}$, and I represents the absence of block in a cache line.

Let $S' = S \cup \{I\}$ The concrete state of the cache is the function $c : L \to S'$, which returns the content of any cache line. C_c denotes the set of all concrete cache states. The concrete cache state can be updated whenever a memory access occurs with an update function $\mathcal{U} : C_c \times S \to C_c$. The effect of this function on the concrete cache state depends on the cache replacement policy.

Example 2.10 (LRU cache update function) *Consider Figure 2.5, which illustrates the update function with an LRU replacement policy on a fully-associative cache. On Figure 2.5a, a new block that was not in the cache is stored. Thus, the block with the highest age (A), which is the least recently used block, is evicted from the cache. The age of each block of the cache is incremented and the new block is added with the age 0. Figure 2.5b depicts what happens when the stored block is already in the cache: the age of each block with an age inferior or equal to the stored block are incremented and the age of the stored block is reset to 0.*

An abstract cache state is defined as the function $\hat{c} : L \to 2^S$, which maps a cache line to sets of memory blocks. An abstract update function $\hat{\mathcal{U}}$ is defined as an extension of the update function $\mathcal{U}$. For the LRU policy, storing a block into the cache only set the age of the block to 0 (not the whole set that contains the block), and then updates the age of all the sets of blocks in the same way as it is done on the concrete state.

The approach proposes to categorize the cache memory accesses as follows:

- *always hit*, which means that the accessed memory block is always in the cache;

- *always miss*, which means that the accessed memory block is never in the cache;
- *persistent*, which is related to loops and means that the first access may be a miss, but that all the other accesses are hits;
- *not classified*, which means that the memory reference could not be classified (we do not know if the data is in the cache or not).

Three analyses enable to classify the memory accesses:

1. the *may* analysis, which guarantees the absence of a memory block in the cache at a program point. In an abstract state of this analysis at any program point, we are sure that the block is not in the cache if the block is not in this abstract state, which corresponds to an always-miss cache access;
2. the *must* analysis, which determines which blocks are always in the cache at a program point. If the block is in the abstract state, then the block is definitely in the cache, which corresponds to an always-hit cache access;
3. the *persistence* analysis, related to loops, detects the blocks that, for a given loop, will never be removed from the cache once they have been loaded until we leave this loop. At each program point, if the block is in an abstract state of this analysis, it means that the block is persistent, i.e. that for a complete execution of a given loop, it misses at most once.

The difference between these three analyses resides in the join operation that happens when a node of the CFG has several predecessors. This join function $\hat{\mathcal{J}} : \hat{C} \times \hat{C} \rightarrow \hat{C}$ merges the different abstract states of the predecessors into a single abstract state.

Example 2.11 (LRU cache join functions) *Consider Figure 2.6. The LRU join operation for the must analysis and the may analysis are presented respectively on Figures 2.6a and 2.6b. The must analysis is pessimistic in the sense that it detects only the blocks that, considering the two abstract states to join, are definitely in the cache. An intersection between the abstract states is used, which removes the blocks that are not in the two abstract states, and then the maximum age of each block is kept. In comparison, the may analysis is optimistic and tries to determine which blocks may be in the cache. It performs an union of the abstract states (it keeps all the blocks that can be in the cache) and keeps the minimum age for each block. The persistence analysis is not depicted here, but it uses the union and keep maximal age in order to detect the blocks that miss the first time and hit all the other times.*

Age	Block
0	{C,D}
1	{}
2	{B}
3	{A}

+

Age	Block
0	{D}
1	{B}
2	{C}
3	{}

=

Age	Block
0	{D}
1	{}
2	{B,C}
3	{}

(a) Must analysis: intersection and maximal age

Age	Block
0	{C,D}
1	{}
2	{B}
3	{A}

+

Age	Block
0	{D}
1	{B}
2	{C}
3	{}

=

Age	Block
0	{C,D}
1	{B}
2	{}
3	{A}

(b) May analysis: union and minimal age

Figure 2.6: LRU policy join functions on abstract states

Using these three analyses, each memory block is categorized, and their categorization can be taken into account during the WCET computation.

What we presented is sufficient for the instruction cache and the read accesses of the data cache. However, with the data cache, we must also take the writing policy into account for write accesses:

- In the case of a write-through/write allocate policy, the access can be treated exactly as a read access;
- For write-through/no write allocate, the access is treated as a read access if the data is in the cache. Otherwise, the access is ignored (the update function is the identity function in this case);
- Regarding write-back, the block that is written into the cache is marked as dirty, which means that its value changed into the cache but has not been saved into the memory yet. When a block is evicted from the cache, we must consider a different time depending on if the block is dirty (it must be written back into the memory) or not.

2.3.3 Branch prediction

When the processor executes a branching instruction, the target address of the branching is known. However, before the execution of this instruction, the processor does not know which instructions will be executed after this branching instruction. If the the branch is taken, we jump to another location into the program. Otherwise the instruction that follows the branching instruction is executed. This means that the processor cannot load

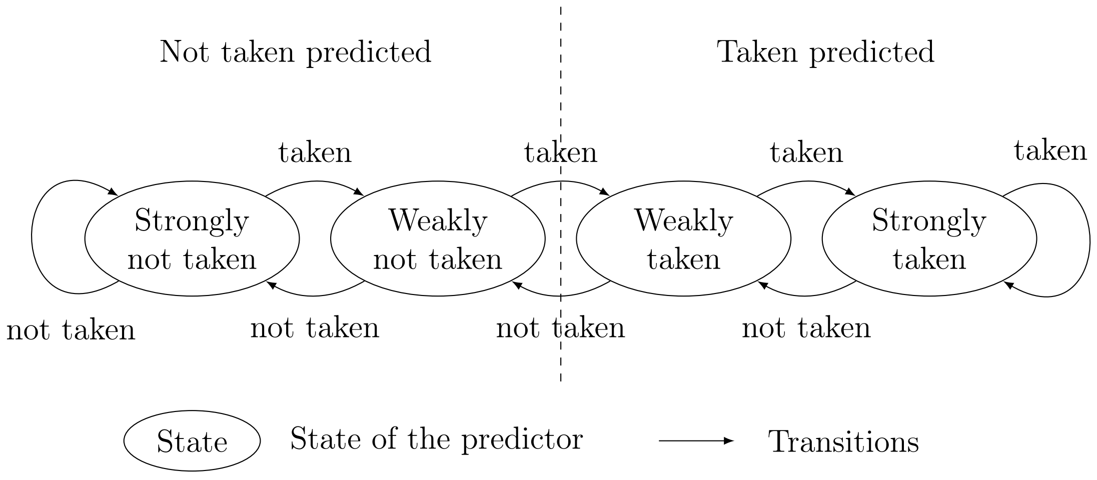


Figure 2.7: State machine BTB representation

the instructions executed after the branching instruction in the pipeline before actually executing it. Nevertheless, modern processors include a *branch predictor*. This branch predictor tries to guess if the branch will be taken or not in advance in order to fetch the instructions into the processor pipeline in advance. Then, at the end of the execution of the branching instruction, if the guess of the branch predictor was correct, the execution continues normally. Otherwise, the instructions that entered the pipeline after the prediction must be removed from the pipeline so as to execute the correct instructions, which results in a timing overhead.

Predictions of the taken branch are based on the *branch target buffer* (BTB), a cache that stores the history of the branches taken for each branching instruction address [CP00]. The branch predictor uses this history so as to predict the branch that will be taken the next time the branching instruction will be executed.

Example 2.12 (Branch predictor representation) *Consider Figure 2.7, which represents a branch predictor with a state machine. Four states (strongly not taken, weakly not taken, weakly taken, strongly taken) represent the history of the decision to branch or not for a branching instruction: taken means that we took the branch, not taken that we executed the instruction that follows the branching instruction. If the state of the BTB is at the right of the dashed line, the BTB will guess that the branch is taken. Otherwise, the BTB guesses that the branch is not taken.*

Colin and Puaut [CP00] proposed to use abstract interpretation so as to bound the worst-case number of mispredictions. Their approach computes *abstract buffer states* (ABSs) for each program point in the CFG. These ABSs indicate, for each BTB entry, which control-transfer instructions (denoted CTIs), e.g. branching instructions, can be in the BTB of the processor (here, a Pentium). The ABSs calculation depends on the replacement policy used for the BTB. Each basic block B_i is associated to two ABSs: ABS_i^{in} for the ABS at the entry of the basic block and ABS_i^{out} for the ABS at the exit of the basic block. For instance, with a n-way set associative with m entry-sets for each way, $ABS_i^{in}[s,k]$ is a set containing all the CTIs that could be in the k^{th} entry of the s^{th} entry-set of the BTB before the execution of B_i. For the LRU replacement policy, k corresponds to the age of the entry.

Note that since a basic block cannot contain any CTI other than its last instruction, there is at maximum one update between the input state and the output state of each basic block. Adding a new CTI to the BTB updates only the set where the CTI is stored. Thus, ABS_i^{out} is obtained from ABS_i^{in} and the instruction i that is added to the ABS using the following update function:

$$\hat{\mathcal{U}}(ABS_i^{in}, i)[s,k] = \begin{cases} i & \text{if } s = set(i) \wedge k = 0 \\ ABS_i^{in}[s, k-1] & \text{if } s = set(i) \wedge k \neq 0 \\ ABS_i^{in}[s,k] & \text{Otherwise} \end{cases}$$

Example 2.13 (BTB update with LRU policy) *Since the BTB is a cache, the behavior of this update function is the same as for the instruction cache. Thus, if we consider a BTB with only one set (i.e. $n = 1$) with the LRU replacement policy, we have the same updates as on Figure 2.5, where E and B are the CTI to store in the BTB.*

Again, since the control-flow graph represents many different paths, when a basic block has several predecessors, a join function is used to merge the abstract states of all the predecessors. We denote:

- $preds(i)$ the set of direct predecessors of B_i in the control-flow graph;
- $\uplus$ the ABS union operator such that $ABS_1 \uplus ABS_2 \triangleq \forall s \forall k, ABS_1[s,k] \cup ABS_2[s,k]$;

Thus, the join function, which produces ABS_i^{in}, is expressed as the union between all the ABS^{out} of the predecessors, such that:

$$ABS_i^{in} = \biguplus_{p \in preds(i)} ABS_p^{out}$$

To categorize each CTI, the loop nesting level *lnl* of each CTI should be known. Indeed, there exist a relation $l_x \succeq l_y$ that indicates that l_x “contains” the loop l_y. The loop nesting level of an instruction corresponds to the nesting level of the loop that directly contains the instruction. Then, each CTI can be categorized as either:

- *Always D-predicted*, which means that the default prediction is used. In the paper, it is “not taken”, but that depends on the modeled processor;
- *First D-predicted*, that means that the instruction is default predicted the first time and deduced from the history the other times;
- *First unknown*, which means that we do not know if the instruction is default predicted or not the first time, but we know that it is history predicted all the other times.
- *Always unknown*, which represents the default classification where we never know if CTI_i is default-predicted or history predicted. This classification occurs when CTI_i is compatible with none of the other classifications.

These classifications can then be used during the WCET computation by considering a penalty when the branch prediction is incorrect.

2.4 WCET computation

Now that we have presented both the flow analysis and the hardware analysis, we can combine the results of these two analyses so as to compute the WCET of a program.

2.4.1 Graph-based techniques

We begin with two classes of approaches that compute the WCET of a program using the CFG.

2.4.1.1 Path-based approaches

Path-based approaches [LS99; Hea+99] explicitly explore the paths in the control-flow graph. They often rely on classic graph algorithms to find the longest path. For instance, Engblom et al. [EES00] implemented a path-based approach using the algorithm of Dijkstra [Dij59]. Generally, these techniques compute a precise WCET bound. However, they are computationally expensive and do not scale for large programs.

2.4.1.2 Implicit path enumeration technique

Implicit path enumeration technique [LM95; LMW95] is the most widely used static WCET computation technique. First, it can efficiently compute the WCET using the CFG without explicitly exploring the paths by transforming the graph into a set of linear constraints. Then, a *solver* can derive the WCET from the linear constraints. Since integer linear programming is not only related to the WCET computation but also to many other domains, this kind of solver implements many optimizations, which results in a fast computation. A second advantage of this approach is that its usage for WCET has been extensively studied and thus many hardware and software facts can be integrated efficiently to the WCET computation (e.g. see section 2.3).

Integer linear programming In essence, an integer linear problem is a mathematical optimization problem, where all the variables are integers. This problem is described with:

- An *objective function*, which is a linear expression that we either want to *maximize*, i.e. to find its greatest possible value, or to *minimize*, i.e. to find its lowest possible value;
- *Linear constraints*, that represent the constraints that the optimization function is subject to. These linear constraints are linear inequalities, that is to say expressions of the form $a_1 x_1 + \cdots + a_n x_n \ OP \ b$, where $(a_1, \ldots, a_n)$ and b are integer values, $(x_1, \ldots, x_n)$ are integer variables and $OP \in \{<, >, \leq, \geq =, \neq\}$.

Then, an ILP solver solves the optimization problem while taking into account all the constraints.

Applying ILP to the WCET problem Regarding the WCET computation, it is possible to encode the control-flow graph as a set of linear constraints, and to express the WCET of the program as a linear expression.

First, for each basic block B_i in the CFG, two values are considered:

- x_i, that represents the number of times B_i is executed. This value is a variable of the problem to solve;
- c_i, which represents the WCET of B_i. This value is constant, which is required to ensure that the optimization problem is linear. It is computed using the hardware analysis.

With these two values for each basic block, the WCET of the whole program, which is the objective function we want to maximize, is expressed as:

$$\sum_{i=1}^{N} c_i \times x_i$$

Example 2.14 (IPET: objective function) *Consider again the CFG of Figure 2.2b. We assume that the low level analysis derived a WCET of 10 for all the basic blocks except for C and E where it derived 5. The objective function of the integer linear problem is then:*

$$10x_A + 10x_B + 5x_C + 10x_D + 5x_E + 10x_F + 10x_G$$

We ask the solver to maximize this function since we want to compute the maximum execution time of the program.

Then, the structure of the CFG is encoded as a set of linear constraints, that implicitly represents the paths that are possible in the graph. These constraints constitute the *structural constraints* of the program.

Example 2.15 (IPET: structural constraints) *Consider again the CFG presented in Figure 2.2b. The structural constraints that translate the graph into a linear program are:*

$$x_A = e_{AB} + e_{AC} \tag{2.1}$$
$$x_B = e_{AB} \tag{2.2}$$
$$x_B = e_{BD} \tag{2.3}$$
$$x_C = e_{AC} \tag{2.4}$$
$$x_C = e_{CD} \tag{2.5}$$
$$x_D = e_{BD} + e_{CD} \tag{2.6}$$
$$x_D = e_{DE} \tag{2.7}$$
$$x_E = e_{DE} + e_{FE} \tag{2.8}$$
$$x_E = e_{EF} + e_{EG} \tag{2.9}$$
$$x_F = e_{EF} \tag{2.10}$$
$$x_F = e_{FE} \tag{2.11}$$
$$x_G = e_{EG} \tag{2.12}$$

In these constraints, e_{XY} is a variable that represents the number of times the edge between the basic blocks X and Y is taken. For each basic block, except for the first and the last block of the CFG, two constraints represent the number of times this basic block is executed depending on its incoming and outgoing edges. For instance, equation (2.2) represents the fact that the number of times the edge between A and B is taken equals the number of times B is executed. Also, equation (2.3) represents the fact that the number of times B is executed is equal to the number of times the edge between B and E is taken.

Then, some *functionality constraints*, which represent information about the flow of the program, are needed to complete the CFG encoding. These constraints represent various information about the flow of the program. In particular, they must bound the number of iterations of loops, and specify the number of times the program is executed.

Example 2.16 (IPET: functionality constraints) *Consider the linear program deduced from the CFG in Example 2.15. We add the two following constraints to the integer linear problem:*

$$x_A = 1 \tag{2.13}$$

$$e_{FE} \leq 4 \times e_{DE} \tag{2.14}$$

Equation (2.13) *gives the number of times the program should be executed, which is one since we want to know the WCET for a single execution of the program. The loop bound is encoded with inequality* (2.14)*.*

Once this problem is formulated, other constraints to represent various hardware and software facts can be added to the problem so as to take them into account during the WCET computation. After that, the integer linear problem is passed to a solver that optimizes the objective function. As a result, the maximum value found for the objective function is the WCET of the program.

Example 2.17 (IPET: Full problem with lp_solve) *Consider Figure 2.8. This figure summarizes the whole integer linear problem translation of the program of Figure 2.2 into the lpsolve 5 [lps] format. Solving the integer linear problem gives a WCET of 105.*

2.4.2 Tree-based techniques

Another way to compute the WCET is to use a tree-based flow representation. This kind of representation has the advantage to make the computation easy by using simple mathematical operators such as sums, products and maximums. Another advantage of this technique is that it is fast: its complexity polynomial in the size of the tree, while the complexity of IPET is exponential in the number of linear constraints in the problem. However, tree-based techniques lack support regarding hardware and software features, which implies that the computed WCET is less accurate.

With tree-based representations, the WCET can be computed using a *timing schema*. For the syntax tree, Puschner and Koza [PK89] introduced a first timing schema that indicates for each type of tree a rule to compute the WCET of this tree using the WCET of its children. Most tree-based WCET computation techniques are derived from this schema. We present a simplified version of the schema used in the control-flow tree representation [BFL17] to demonstrate how the WCET can be computed inductively on the tree structure:

```
/* objective function */
max: 10 x_A + 10 x_B + 5 x_C + 10 x_D + 5 x_E + 10 x_F +
  10 x_G;

/* structural constraints */
x_A = e_AB + e_AC; /* output constraint */

x_B = e_AB; /* input constraint */
x_B = e_BD; /* output constraint */

x_C = e_AC; /* input constraint */
x_C = e_CD; /* output constraint */

x_D = e_BD + e_CD; /* input constraint */
x_D = e_DE; /* output constraint */

x_E = e_DE + e_FE; /* input constraint */
x_E = e_EF + e_EG; /* output constraint */

x_F = e_EF; /* input constraint */
x_F = e_FE; /* output constraint */

x_G = e_EG; /* input constraint */

/* functionality constraints */
x_A = 1; /* function entry constraint */
e_FE <= 4 e_DE; /* loop constraint */
```

Figure 2.8: Integer linear problem under lp_solve

- For a leaf, the WCET is obtained with the hardware analysis;
- For a sequence, the WCET is the sum of the WCET of the children trees;
- For an alternative, the WCET is a maximum among the WCET of the children trees;
- For loops, the WCET is the WCET of the body multiplied by the loop bound, plus one time the WCET of the loop header.

Example 2.18 (Tree-based WCET computation) *Consider the tree-based control-flow representation in Figure 2.2c. Let $\omega(b)$ be the WCET of the basic block b that is computed with the hardware analysis. Assume that the loop bound of the loop on the figure is 4 and the WCET of each basic block is 10, except for C and E where we assume a WCET of 5. Thus, the WCET of the tree is:*

$$\begin{aligned} w &= \omega(A) + max(\omega(B), \omega(C)) + \omega(D) + (n \times (\omega(E) + \omega(F)) + \omega(E)) + \omega(G) \\ &= 10 + max(10, 5) + 10 + (4 \times (5 + 10) + 5) + 10 \\ &= 10 + 10 + 10 + 60 + 5 + 10 \\ &= 105 \end{aligned}$$

2.4.3 Model checking

The *model checking* approach [Hen+14; Met+16; Bec+19] has also been considered to derive the WCET of a program. The principle of this technique is to construct a model from the program, where the basic blocks are represented by their WCETs. Then, a model checker is used to determine an upper bound to the WCET of the program. We illustrate the principle of these methods with the approach of Becker et al. [Bec+19].

A global variable `_time = 0` is added into the source code of the program to represent its execution time. The WCET of each basic block is inserted into the source code at the corresponding location as a value added to the global execution time variable (e.g. `_time += x`). Then, all the lines of code that are not related to the WCET computation, that is to say all the lines that are not related to the control-flow of the program or to the `_time` computation, are removed from the source code. An initial estimation of the WCET is then inserted at the end of the program code (e.g. `assert(_time <= estimated)`). A model checker, e.g. CBMC [CBM] in this work, is then used to check if the estimated WCET is a valid upper bound to the execution time of the program.

The analysis is parameterized with a precision level, that represents the maximum difference between the highest lower bound to the WCET (i.e. the greatest value for which the assertion fails) and the lowest upper bound to the WCET (i.e. the lowest value for which the assertion is true). The model checker is called iteratively with different estimations of the WCET until the precision level is reached.

2.5 Auxiliary analyses

In this section, we present an overview of the techniques used to address two common problems of the flow analysis:

- Bounding the number of iterations of loops;
- Eliminating infeasible paths.

Different approaches have been developed to address these issues, but none of them are exact. For instance, being able to find a loop bound to any loop would solve the halting problem, which is known to be an undecidable problem.

2.5.1 Loop bound analyses

The loop bound analysis is a complex problem and existing tools only solve it partially. Thus, the bounds that the analyses are not able to infer must be provided by the user. We present two example approaches that show how the loop bounds can be derived.

Healy et al. [Hea+98] proposed a method, implemented in a compiler, that bounds the number of iterations in the case of loops with multiple exits. First, the method detects the conditional branches inside of the loop body that can affect the loop bound. Then, it produces a directed acyclic graph (DAG), a graph similar to the CFG except that it does not contain any cycle, that represents the loop body paths for a single iteration. Then, by analyzing the branch conditions in the DAG and the the variable value modifications within each path of the program, it derives upper-bounds to the number of iterations of loops.

Another method [KKZ12] relies on recurrence solving and theorem proving techniques to derive loop bounds when the update of loop iterations can be represented as linear expressions on the program variables. In this work, loops with multiple paths are approximated to single path loops, which are then translated into recurrence equations over the variables of the program. These equations are then solved to derive a loop bound.

2.5.2 Infeasible paths analysis

We now concentrate on some other approaches that remove infeasible paths. Various techniques to detect infeasible paths exist [DT13] and are used in various domains, e.g.

program verification. Regarding the WCET analysis, the detection of infeasible paths enables to reduce the pessimism. Nevertheless, since these analyses are computationally expensive, a trade-off between analysis time and precision must be found. We present various techniques that detect infeasible paths targeted at WCET computation.

Abstract interpretation Some approaches use abstract interpretation. This kind of approach analyzes the possible values of the program variables so as to extract properties including loop bounds [Gus00; Bal+19], and even infeasible paths [HW02; Suh+06; Che+07; RC15; RCM17]. We illustrate the principle of this approach with the technique presented in [Suh+06]. This approach only considers the infeasible paths inside of loop bodies that are *pairwise conflicts*. A pairwise conflict is defined as a conflict between two edges, denoted (e, e'), or between an assignment and an edge, denoted (u, e) where u is the basic block that contains the assignment. The technique only supports assignments of the form $x = c$ and branch conditions of the form $x \ OP \ c$, where $OP \in \{<, >, \leq, \geq, =, \neq\}$, x is a variable and c a constant value.

The analysis is performed on the CFG of the program. Each loop body is transformed into a DAG. Then, an abstract interpretation of this DAG keeps trace of:

- The last assignment of each variable in the loop body;
- The branching condition for each edge in the DAG.

For a conflict between an assignment and an edge, the interpreter determines if the conjunction of the assignment and the current edge branching condition is *satisfiable*. If the conjunction is not satisfiable, it means that the corresponding path can be marked as infeasible. A similar principle is used for conflicts between two edges.

Symbolic execution *Symbolic execution*, an analysis method that executes a program with symbolic inputs, is also used by some other techniques in order to detect infeasible paths [Alt96; GEL06]. Some of those techniques detect both infeasible paths and loop bounds [Keb06; Gus+06]. Symbolic execution is very similar to the actual execution of a program. The difference is that instead of using real program input values for the execution, the program inputs are represented by symbolic values. As a result, all the values of the variables that depend on the input values are represented as expressions over these symbolic values.

We present the principle of these approaches with the work presented in [GEL06]. The approach presents three algorithms that detect the infeasible paths using *abstract execution*, an analysis that combines both abstract interpretation and symbolic execution.

A first algorithm, which is also the simplest one, detects the nodes that are never executed in a control-flow graph. The analysis uses an array of bits, where each bit represents the feasibility of a basic block in the CFG. At the beginning of the analysis, all the bits are set to 0, which means that the nodes are infeasible. Then, the symbolic execution is performed on the graph, and each time a basic block is traversed, the array is updated and the value 1 is set for this basic block. At the end of the analysis, the basic blocks that were never executed by the symbolic execution are still represented by 0 in the array, thus we know that they are infeasible.

A second algorithm detects pairs of nodes that are mutually exclusive within specific scopes. For instance, the scope can be the body of a loop, which means that in that loop, the two nodes of the pair are never executed during the same iteration. The analysis records all the paths that are used during the symbolic execution of the scope. A triangular matrix of size $N \times N$, where N is the number of nodes in the CFG, is built. This matrix can either contain 1 to indicate that the two nodes are not mutually exclusive 0 to indicate that they are or $\perp$ to indicate that the pair has not been analyzed. The analysis iterates over the paths of the program. For each pair of nodes in each path, the nodes are marked as not mutually exclusive. All the pairs that correspond to one node of the path and an alternative to a node of the path are marked as mutually exclusive, but only if they were not analyzed previously (i.e. only if they are $\perp$). At the end, all the 0 in the matrix indicate the infeasible pairs of nodes.

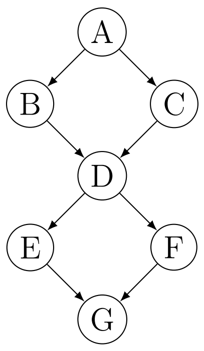
The last algorithm uses trees to represent the paths taken in the CFG. Each path symbolically executed is represented by a tree, which contains both the nodes of the path and their alternatives in the CFG. Then, all the trees are merged into a single tree, that represent feasible and infeasible paths.

Example 2.19 (Infeasible paths with trees) *Consider Figure 2.9. The program and its CFG are presented in Figure 2.9a and Figure 2.9b. Consider that the symbolic execution produced two paths represented by trees of Figure 2.9c and Figure 2.9d. On these trees, the colored nodes correspond to the nodes not taken in the CFG. In the merged tree, the paths passing through B, E and C, F are feasible whereas the paths passing through B, F and C, E are not.*

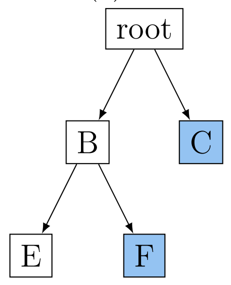
SMT The method proposed in [BLH14] relies on a SMT solver to detect infeasible paths in binary code. As the other approaches, it starts from the CFG of the program. To make it amenable to a SMT solver, the control-flow graph is transformed into its single static assignment (SSA) form. The SSA form is a representation of the program where there is only one assignment for each variable. If the original program contains several assignments to the same variable, we then use a different variable for each assignment.

```
int f(int x){
  //A
  if(x > 0)
    // B
  else
    // C
  // D
  if(x > 0)
    // E
  else
    // F
  // G
}
```

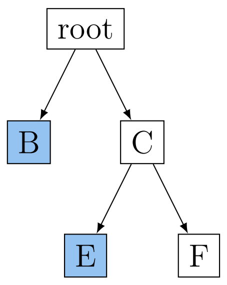
(a) “Diamond” program



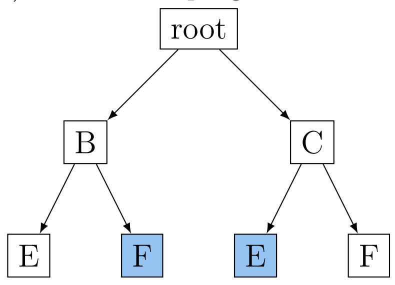
(b) CFG of the program



(c) Path through B and E



(d) Path through C and F



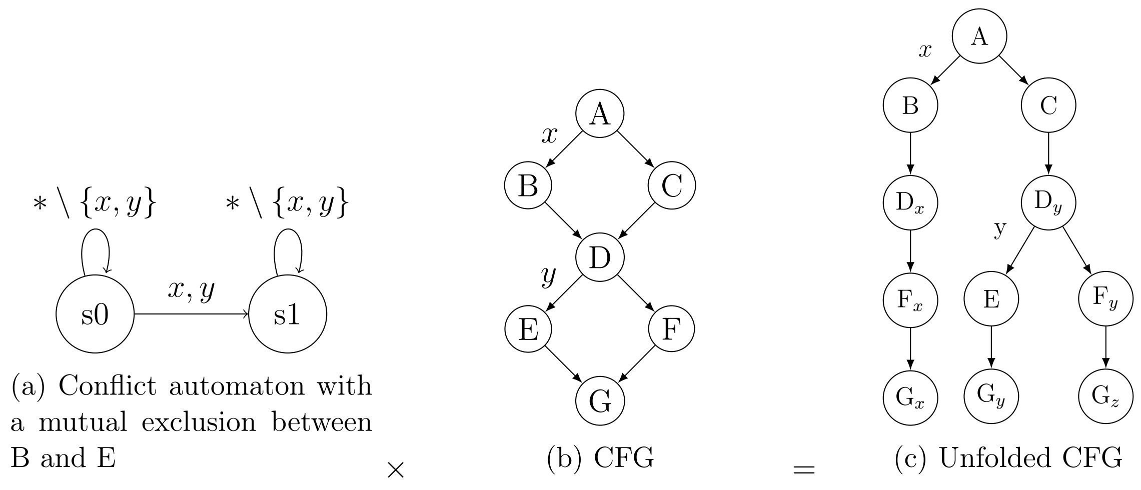
(e) Merged tree

Figure 2.9: Infeasible paths with tree algorithm

The approach computes the WCET with IPET technique iteratively. First, the path that triggered the WCET is reconstructed from the IPET solution (in particular the number of times each basic block is executed). The SMT solver checks that the corresponding path is feasible. If the path is feasible, then the WCET of the program is found. Otherwise, the CAMUS algorithm [LS08] is used to find the subset of CFG nodes that are incompatible (that makes the path infeasible). The paths that contain all these nodes are then excluded from the IPET representation using an ILP constraint and a new WCET is computed. The process is repeated until the path that triggers the WCET is feasible.

2.5.2.1 Using infeasible paths during WCET computation

With these analyses, infeasible paths are detected. However, another problem is to remove these infeasible paths from the paths that are taken into account while computing the WCET. In this section, we detail some generic approaches to take infeasible paths into account during the WCET computation. All of these approaches rely either on CFG transformations or ILP constraints to exclude the infeasible paths from the WCET computation.



(a) Conflict automaton with a mutual exclusion between B and E

(b) CFG

(c) Unfolded CFG

Figure 2.10: CFG transformation using automata product

Graph unfolding using automata product Mussot et al. [MS15; Mus+16] proposed a technique to perform *graph unfolding*. By representing both the CFG and the infeasible paths as automata, it is possible to use an automaton product to remove the infeasible paths from the CFG.

First, the CFG is viewed as a deterministic automaton, where each basic block of the CFG becomes a node of the automaton and each edge of the CFG becomes a labelled edge in the automaton. Second, an infeasible path can be represented as a simple automaton, the *conflict automaton*, that accepts any path except the infeasible ones.

Example 2.20 (Infeasible paths as an automaton) *Consider the automaton in Figure 2.10a. It represents the mutual exclusion between the nodes B and E from Figure 2.10b. This infeasible paths uses the edges of the CFG so as to express the infeasibility: x is the transition from A to B and y the transition between D and E. Indeed, if we pass through one of these edges, we take the transition to $s1$. After this transition, we cannot pass through B or E.*

Then, the technique proposes to perform an automata product so as to remove the infeasible paths. We do not detail the automata product algorithm here. The result of this product is an automaton that represents the CFG without the infeasible paths, that can then be used to perform IPET.

Example 2.21 (CFG unfolding using automata) *Consider Figure 2.10. This figure represents the product between a conflict automaton and a CFG. As a result, the CFG is unfolded into a CFG without the infeasible path. As expected, the paths that passes through B and E does not exist in the resulting CFG.*

As demonstrated in [Mus+16], unfolding the CFG enables to take into account the infeasibile paths even during the hardware analysis, thus it can produce a tighter WCET. Nevertheless, this technique is also computationally expensive: for a conflict automaton with n edges the resulting CFG contains up to $2^n - 1$ nodes since it duplicates many nodes.

Encoding Infeasible paths as ILP constraints Many different works tried to express infeasibility as ILP constraints. Most of the time, the works presented here detect some specific infeasible paths in a program and explain how to express these specific constraints into the integer linear program that computes the WCET.

A first approach [HW02; BLH14] consists in representing simple conflicts between nodes. It only considers that infeasible paths are conflicting sets of n nodes $\{B_1, \ldots, B_n\}$ where all the basic blocks are located outside of any loop (or only in a single loop iteration). The approach then adds, for each conflicting set, an ILP constraint of the form $x_1 + \cdots + x_n \leq n-1$. This constraint expresses the fact that the sum of the number of executions of all nodes in a conflicting set can only be lower than the number of nodes in the conflicting set.

Example 2.22 (IPET: mutual exclusion outside loops) *Consider the CFG in Figure 2.10b. If, as in Example 2.20, we want to express that we cannot use a path that passes through B and E, we can add a constraint on the number of executions of these nodes:*

$$x_B + x_E \leq 1$$

This constraint simply expresses that the number of executions of B, denoted x_B and the number of executions of E, denoted x_E cannot be both set to 1 in the solution found by the ILP solver, which excludes all the paths that pass through both B and E.

A second class of techniques [EE00; Gus+06; HW02] goes further by considering sets of nodes that conflict over a complete loop execution. In such cases, the loop bound must be used to consider the infeasible paths. The approach is very similar to the previous one: consider that n is the bound of the loop to which all the nodes of the conflicting set belong, then, the infeasible path can be abstracted as $x_1 + \cdots + x_n \leq n$.

Example 2.23 (IPET: mutual exclusion within loops) *Consider the CFG in Figure 2.10b. As before, we want to express the mutual exclusion between B and E. This time, we assume that the CFG is located in a loop that iterates at maximum 4 times. Then, the mutual exclusion with the following linear constraint:*

$$x_B + x_E \leq 4$$

This constraint simply means that when the number of iterations of the loop is 4, the sum of the number of executions of B and E is at maximum 4, which abstracts the fact that we cannot execute the two nodes in the same loop iteration.

Raymond [Ray14] also showed that in certain cases, it is possible to express conflicts between different iterations of the same loop or across loop iterations (i.e. conflicts that depend both on nodes in and out of a loop), with some ILP constraints. However, this technique relies on ad-hoc reasoning, and is not a general approach. A more generic solution can be to mix some ILP constraints and graph unfolding [KBC10; Ray14].

2.6 Parametric WCET computation

Now that we presented the state of the art regarding static WCET computation, we focus on parametric WCET computation techniques. The specific nature of these methods is that the WCET that they produce is not expressed as an integer value representing the number of processor cycles needed to execute the program. Instead, a parametric formula that depends on some parameters is generated. This formula can later be instantiated to derive the WCET for some given parameter values.

This kind of technique can have different applications. 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 [BDB08; GC11] to determine which parameter values make the system schedulable. Second, it can be instantiated on-line to perform adaptive scheduling, such as:

- Dynamic voltage and frequency scaling [Moh+05; Moh+11], that is an energy-aware scheduling technique which consists in adjusting the voltage and the frequency of a processor when the execution times of tasks in the system is lower than their deadlines;

- Slack reclaiming [CBS00; LB00; Pal+08], which is able to attribute the unused execution time of a task to another task;

- Semi-clairvoyant scheduling for mixed-criticality, which considers several WCETs for each task. Depending on the WCET values that will be used by highly-critical tasks, some low-critical tasks can be discarded to ensure that the highly-critical tasks are executed before their deadlines.

2.6.1 Parametric WCET from intermediate code

A first approach presented in [Viv+01; Cof+07] proposes to derive a parametric WCET formula from an intermediate code, that is compiled from the source code but that is not as low-level as binary code.

This approach relies on a compiler to obtain the control-flow graph of the program. Then, a timing analyzer uses the control-flow graph to estimate the WCET of each loop and function (functions are considered as loops that only iterate a single time). From these estimations, a timing analysis tree is constructed, where each node of the tree corresponds to a loop or a function.

The WCET of the program is then computed in a bottom-up manner on this tree, i.e. the WCET of a node of the tree can be computed when the WCET of all its children trees are computed. For each node of the tree, the WCET is computed as the product between the WCET of the path that correspond to a loop iteration (or the function body) and the number of iterations of the loop. The parametric extension of this tool consists in introducing a parameter, which represents the number of iterations of a loop, into the WCET computation. Thus, instead of an integer value, the analysis produces a polynomial formula that depends on the number of iterations of that loop.

This technique brings the first notion of parametric formula into WCET computation. However, the method only support a single parameter.

2.6.2 Parametric integer programming

Another category of methods [Lis03; BL08; Alt+08] relies on a parametric extension of integer linear programming, called parametric integer programming [Fea88], to compute a parametric formula. We detail the approach of Altmeyer et al. [Alt+08] to illustrate the principle of these approaches.

The first step of the analysis is the parameter analysis. This analysis attempts to detect which elements in the program are procedure arguments. A procedure argument is defined as a variable that is read by the program before it is written to in the binary code. The parameters of the procedure are detected using abstract interpretation of the code. This abstract interpretation, which represents the possible values of each variable as an interval, is also able to detect the other variables that depend on a parameter value, and to express their values, as a sum between the parameter and a constant value.

Then, the loops of the program are analyzed in four phases:

1. Collect the potential loop counters (the values that change in the loop body);
2. Derive the loop invariant;

3. Evaluate the loop exit;

4. Construct the loop bound.

Finally, the rest of the analysis is very similar to IPET: the hardware analysis is used on the CFG to estimate the WCET of each basic block, and a parametric integer program, akin to an integer linear program but with parameters, is derived from the CFG. The parametric integer programming solver then produces the parametric WCET formula.

This technique benefits from almost all the advantages of IPET, e.g. the hardware and software facts modeling. The drawback, however, is the parametric integer programming, which is known to be computationally very expensive.

2.6.3 The minimum propagation algorithm

Bygde et al. [BEL11] tried to address the complexity problem of the parametric integer programming approach. The proposed framework is very similar to the previous one, except that the method does not use parametric integer programming. Instead, the approach proposes the *minimum propagation algorithm* (MPA).

The principle of this algorithm is based on the assumption that a basic block in the CFG cannot be visited more time than the number of times its predecessors and successors are visited, exactly as IPET. The algorithm builds a *min-tree* for each node of the CFG. This min-tree is built from constraints similar to structural constraints derived from the CFG. It has three kinds of nodes:

- Leaves, which represent the number of executions of basic blocks related to the value of the number of executions of the basic block under analysis;

- Minimum nodes, represented by $\Diamond$, that represent a minimum among their children trees;

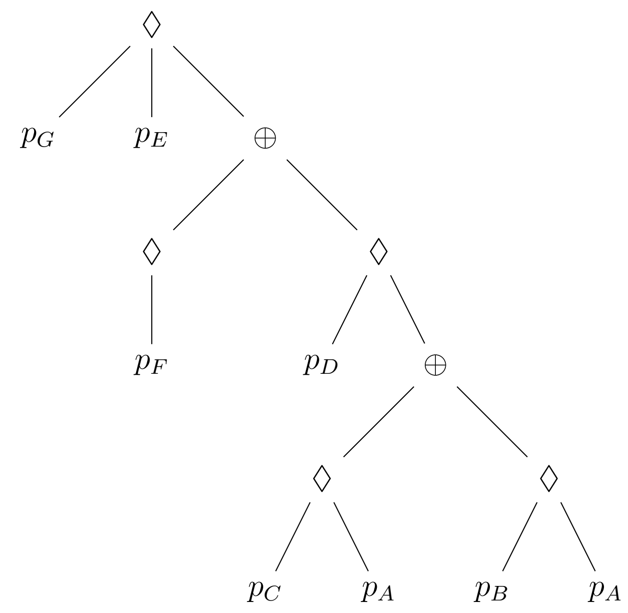
- Plus trees, represented by $\oplus$, which represent the sum between their children tree.

This tree is then translated into a formula that expresses the number of time the basic block is executed depending on the number of times its predecessors and successors are executed.

Example 2.24 (min-tree) *Consider again the CFG of Figure 2.2b. For each basic block, a constraint is generated on the execution counts of the basic blocks to bound it by the sum of the execution counts of its predecessors. A similar constraint is also generated for its successors, we thus have the constraints presented in Figure 2.11a. Using the MPA algorithm on these constraints generates the min-tree of Figure 2.11b for the basic*

$$
\begin{aligned}
&x_A \leq x_B + x_C\\
&x_B \leq x_A, x_D\\
&x_C \leq x_A, x_D\\
&x_D \leq x_B + x_C, x_E\\
&x_E \leq x_D + x_F, x_F + x_G\\
&x_F \leq x_E, x_E\\
&x_G \leq x_E
\end{aligned}
$$

(a) Constraints



(b) The min-tree for the basic block G

Figure 2.11: Minimum propagation algorithm illustration

block G. This tree is then translated to a formula that expresses the number of times G can be executed:

$$t_G = min(p_G, p_E, p_F + min(p_D, min(p_C, p_A) + min(p_B, p_A)))$$

where p_i represents a symbolic maximum bound to x_i.

Then, the WCET is computed using the following formula:

$$\sum_{i=1}^{N} c_i \times x_i$$

As for implicit path enumeration technique, c_i must be replaced by the WCET of the basic block, and x_i is replaced by the formula derived from the min-tree of the corresponding node (or a parameter). The formula is then simplified and can be instantiated to obtain the WCET of the program. Since the technique generates a formula, the execution count of basic blocks can also be replaced by symbols. This means that the approach supports parametric loop bounds.

This approach has a lower complexity and thus is better-suited to bigger programs than techniques that use parametric integer programming. However, the experiments conducted on this technique show pessimism for most programs in comparison to parametric integer programming. This pessimism comes from the fact that the technique often consider that when two alternative paths are possible, the two of them are executed. This technique can be viewed as a better trade-off between precision and analysis time for bigger programs.

2.6.4 Parametric path analysis

An alternative to MPA is parametric path analysis [AAN11a; AAN11b]. It relies on a property that is valid for many programs: each loop in the analyzed CFG must have a single entry. With this assumption, the goal of the presented algorithm is to transform the CFG into a directed acyclic graph (DAG).

In order to operate this transformation, the analysis computes the longest path of each loop, from the inner-most loop to the outer-most loop. Once the longest path of an inner loop is computed, the loop is replaced by a synthetic node with the WCET of the longest path in that loop. Once all the loops have been replaced by a synthetic node, the CFG is a DAG, in which the longest path can easily be computed.

If a loop has a symbolic loop bound, its WCET is then a parametric expression that depends on this symbol.

In case of several paths, we must keep all the parametric paths as well as the longest non parametric path because it is not possible to determine which of these paths is the longest one statically.

2.6.5 Tree-based parametric WCET

Several tree-based techniques that support parameters have been developed. Since these approaches rely on recursive computations, they are particularly well-suited for parametric computations without increasing their complexity.

Černý et al. [Čer+15] introduced the *abstract segment tree* (AST). This tree is built from the CFG using segment abstraction [CC12; CHR13] and it represents the different paths of the CFG as a tree structure. Each node of the tree contains three pieces of information:

- A name, which uniquely identifies the segment;
- A shape, which describes how the set of paths of the segment are constructed from the paths of its children segments. For instance, it can represent an alternative between the paths in its children segments;
- A Transition predicate which is a formula over the values of program variables, that describe the values of the program variables at the beginning and at the end of segments. For instance, it can indicate a predicate on the initial value of a variable or the new value of a variable updated during the segment execution.

As for the other tree-based approaches, the WCET of the AST can computed depending on the shape of the node: alternatives produce maximums between the WCET of the children, sequences produce sums and a multiplication for loops. This approach, as the previously introduced ones, proposes to use loop bounds as parameters.

Colin and Bernat [CB02] presented a generalization of the syntax tree that supports symbolic computation. The main difference with the syntax tree is that each node of the tree has two expressions:

- A cost expression, that expresses how the WCET of the node should be computed. This expression can rely on its children tree and/or some variables (potentially some parameters);
- A frequency expression, that defines how many times the node of the tree is executed regarding a parent node (that may not be the direct parent of the node).

However, the authors do not provide any way to build this tree from the program.

In the next section, we detail a last tree-based approach on which this thesis focuses. In particular, we provide the background needed to understand the various contributions of the thesis.

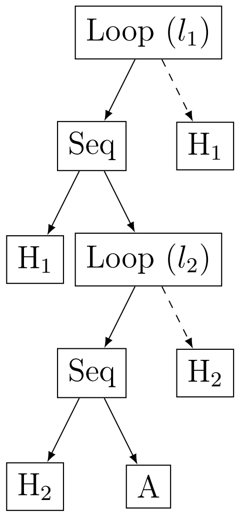
2.7 Background: Symbolic WCET computation

The technique of Ballabriga et al. [BFL17] also presented a tree-based symbolic computation technique. 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 fairly easily introduce symbolic values.

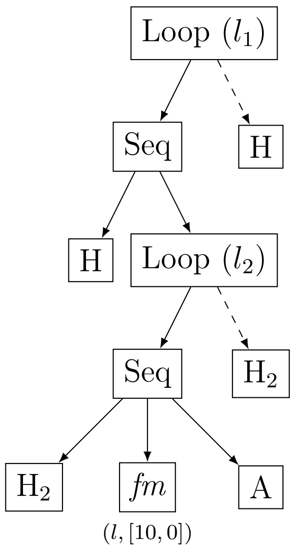
2.7.1 Control-Flow Tree

A Control-Flow Tree can be one of:

- Leaf(b), which holds the basic block b of the program;
- Seq($t_1, \dots, t_n$), which represents the sequential execution of trees $t_1, \dots, t_n$;
- Alt($t_1, \dots, t_n$), which represents the execution of one tree among $t_1, \dots, t_n$;



(a) Before transformation



(b) After transformation

Figure 2.12: Instruction cache transformation

- 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.

The set of *structurally feasible* paths in a CFT t is denoted tpaths(t).

2.7.2 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*, 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.

Example 2.25 (CFT: Abstract WCET) $(l, [10, 10, 5, 3])$ *represents the WCET of a node inside loop l. The WCET of the node is at most twice 10, once 5, and 3 for all other iterations of loop l.*

Example 2.26 (CFT: several WCET for several iterations) *Let us illustrate how we can represent the effect of the instruction cache. Consider the CFT of Figure 2.12a. Assume that a cache categorization technique [Alt+96] determines that A contains a* first-miss *cache access, i.e. the instruction is in the cache for all iterations except the first one. Assume also that the cache miss penalty is 10 cycles. This is modeled in Figure 2.12b by a leaf fm with WCET* $(l, [10, 0])$.

The following definitions on the program topology are required to define operations on abstract WCET:

- Loop l_1 is said to *contain* loop l_2, denoted $l_2 \sqsubseteq l_1$, if the header of l_2 is located inside the body of l_1;
- $\top$ is a fictive loop that refers to the program top-level scope;
- $\perp$ is a fictive empty loop;
- Let L denote the set of loops of the program. Then, $(L \cup \{\top, \perp\}, \sqsubseteq)$ is a lattice;
- $l_1 \sqcap l_2$ denotes the greatest lower bound of l_1 and l_2, that is to say the greatest element of $\{l | l \sqsubseteq l_1 \wedge l \sqsubseteq l_2\}$

We now remind operations on abstract WCETs. Let $a = (l, w)$ and $a' = (l', w')$ be abstract WCETs. Then:

- θ is the null abstract WCET, where $\theta = (\top, [0])$.
- $w[n]$ denotes the $(n+1)^{\text{th}}$ greatest element of w;
- $(l", w") = a \oplus a'$ is a pointwise sum, such that $w"[i] = w[i] + w'[i]$ and $l" = l \sqcap l'$. This operator is used to sum two WCETs;
- $a \uplus a' = (l \sqcap l', (w \cup w') \setminus \{k | k < \mathit{min}(w) \vee k < \mathit{min}(w')\})$ is an order-preserving list union, except that elements smaller than infinitely repeated ones are dropped. It is used to compute the maximum between two WCETs;
- $(l, w)^{n,l'}$ represents an iteration over (l, w), where n is the number of iterations and l' the loop identifier corresponding to the loop we are iterating on. There are two cases:
  - if $l = l'$, then it sums the n greatest elements of w;
  - if $l \neq l'$, then it sums the elements of w by packs of n.

  More formally (see Example 2.28 for an 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 2.27 (CFT: Abstract WCET operators) *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* $(l, [5, 4])^{4,l'} = (l, [5 + 4 \times 3, 4 \times 4]) = (l, [17, 16])$.

2.7.3 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 2.28 (CFT: Exact first-miss representation) *In Figure 2.12b, 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 2.12b, 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. 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 $(l, \{5\}) \oplus X$.

As a final step, the reduced formula is translated into C code, that can be used off-line or on-line to instantiate the formula when symbol values become known. The approach is implemented in a tool called WSymb [WSy], an OTAWA [Bal+10] extension.

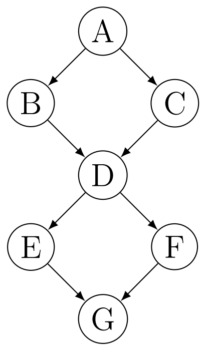
Chapter 3

Infeasible paths in parametric tree-based WCET analysis

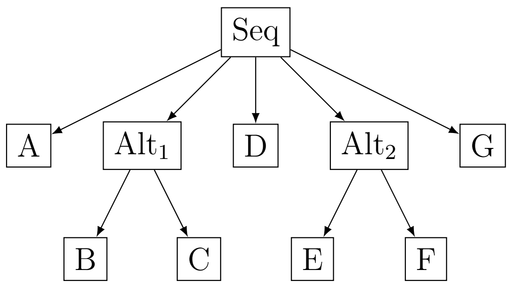
Contents

```
int f(int n){
  //A
  if(n > 10)
    //B
  else
    //C
  //D
  if(n < 0)
    //E
  else
    //F
  //G
}
```

(a) A C procedure with an infeasible path



(b) Control-flow graph



(c) Control-flow tree

Figure 3.1: A procedure and its control-flow representations

In this chapter, we introduce a first contribution in which we propose a method to take infeasible paths into account in a tree-based WCET computation technique. We consider in this chapter that each basic block is uniquely identified, that is to say a basic block cannot appear more than once (except in the loop exit tree[1]).

3.1 Introduction

An infeasible paths, as defined in Section 2.2.2.3, is a succession of basic block, that is:

1. *Structurally feasible* according to the control-flow representation of the program, which means that this succession of basic block is valid in this program representation;

2. *Semantically Infeasible*, which means that the path is infeasible if we take into account the program code semantics.

An example of infeasible path is depicted in Figure 3.1. We consider that in this program the value of n is not modified. Structurally, a path exists through B and E in the control-flow graph (CFG) as well as in the control-flow-tree (CFT). However, since the value of n does not change in the program, this path is infeasible semantically, because n cannot be both greater than 10 and lower than 0. Providing information about infeasible paths

[1]The exit tree of the loop always contains basic blocks that are in the loop body

to the WCET analyzer can reduce the pessimism of the computed WCET. Indeed, if the longest path in the program representation is infeasible, then the computed WCET is pessimistic since it uses the execution time of a path that is not actually feasible in the program. Note that this work does not address the problem of detecting infeasible paths. Instead, it focuses on eliminating infeasible paths from the control-flow tree representation used to compute the WCET.

To remove infeasible paths, we propose to operate transformations on the CFT such that no path marked as infeasible remain in the resulting CFT. As a consequence, the symbolic computation can be directly applied on the transformed CFT and no further extensions to the WCET computation are required to support infeasible paths.

3.2 Overview

We start by introducing the general workflow of our approach. Our goal is to remove infeasible paths from a CFT. The main challenge is to avoid the enumeration of all the paths in the program, which is intractable on most programs. We first define two key concepts of our method:

- *Infeasibility constraints* express the infeasible paths supported by our technique (Section 3.3.1). In essence, an infeasibility constraint is a set of basic blocks that cannot be all together in a path. For instance, assume $\{B, E\}$ is an infeasibility constraint in Figure 3.1. This constraint means that any path including both the basic blocks B and E, e.g. $A.B.D.E.G$ in the CFT, is infeasible;

- *Pseudo paths* are a compact representation of a set of paths (Section 3.3.2). For instance, Consider again the infeasibility constraint $\{B, E\}$ on Figure 3.1. The pseudo path $\{B\}$ for the constraint $\{B, E\}$ is an abstract representation of all the paths that pass through B but not through E. The other pseudo paths for this infeasibility constraint are:

    - $\{B, E\}$, that represents all the paths that pass through both B and E;
    - $\{E\}$, that represents the paths that pass through E but not through B;
    - $\{\}$, which represents the paths that pass through neither B nor E.

With the help of these two concepts, we transform the CFT so as to remove all the infeasible paths from it following the workflow depicted in Figure 3.2:

1. First, we compute all the pseudo paths for a CFT with respect to a set of infeasibility constraints (Section 3.3.3);

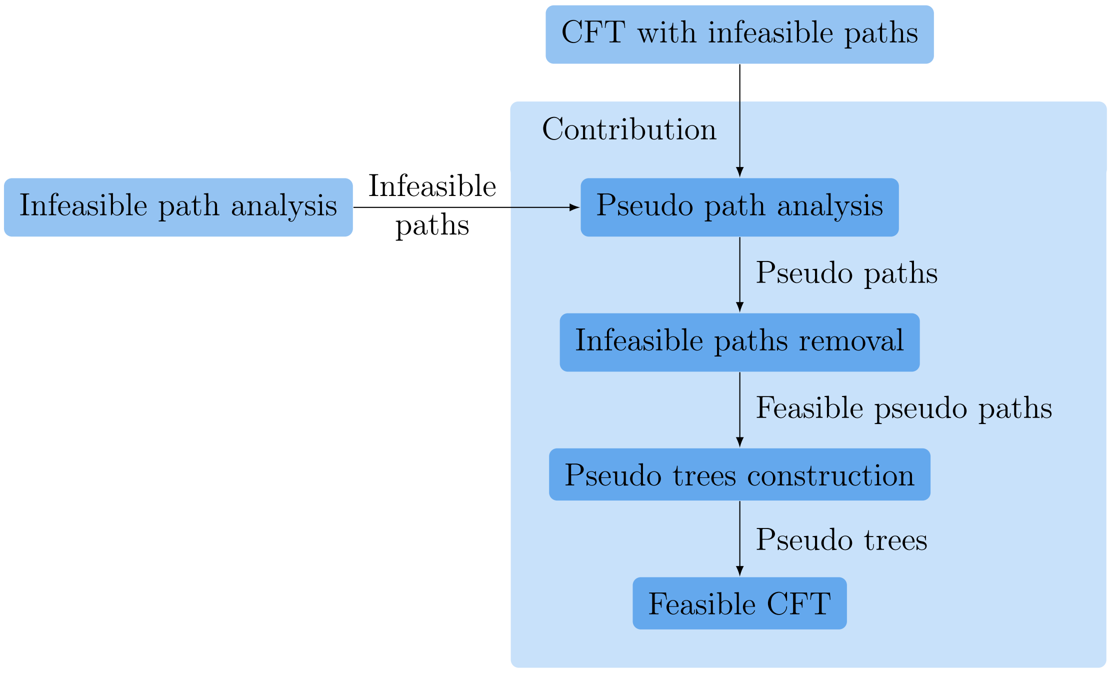


Figure 3.2: Workflow of the proposed infeasible paths representation technique

2. Then, we remove the pseudo paths corresponding to infeasible paths (Section 3.3.3). For instance, consider again the CFT of Figure 3.1c with the infeasibility constraint $\{B, E\}$. Here the pseudo path $\{B, E\}$ represents the infeasible paths expressed by the infeasibility constraint $\{B, E\}$, so we remove it from the set of pseudo paths;

3. We then build *pseudo trees* for each feasible pseudo path (Section 3.4.2). A pseudo tree is a CFT that represents the concrete paths abstracted by a pseudo path. As an example, with the infeasibility constraint $\{B, E\}$, Figure 3.4 represents the pseudo tree of the pseudo path $\{B\}$, that is to say all the paths passing through the basic block B but not through the basic block E since the paths passing through both the nodes are included in another pseudo path: $\{B, E\}$;

4. Finally, we build the feasible CFT with all the pseudo trees corresponding to feasible pseudo paths in Section 3.4.3. For example, the CFT on Figure 3.6 represents the *feasible tree* of the CFT on Figure 3.1c using an alternative between all the feasible pseudo trees. On this tree, children trees of the root Alt tree are the pseudo trees corresponding to the pseudo paths $\{B\}$, $\{E\}$ and $\{\}$;

3.3 Pseudo paths

In this section, we first detail the type of infeasible paths supported by our method, represented by infeasibility constraints. Then, we explain how we can build our abstract paths representation, called pseudo path, from these infeasibility constraints.

3.3.1 Infeasibility constraints

We begin by defining the notion of infeasibility constraints, which represent the type of infeasible paths supported by our approach.

Definition 3.1 (Infeasibility constraint) *Let s be a set of basic blocks, b be a basic block, and p be a program path. We define the inclusion of a set of basic blocks in a path as:*

$$s \sqsubseteq p \Leftrightarrow \forall b \in s, b \in p$$

An infeasibility constraint is a set of basic blocks, which can represent one or several infeasible paths. Let c be an infeasibility constraint and p be a program path. The path p is infeasible if:

$$c \sqsubseteq p$$

Example 3.1 (Infeasibility constraint) *Consider the tree of Figure 3.1c with the infeasibility constraint $\{B, E\}$. This infeasibility constraint means that all the paths that pass through those two basic blocks are infeasible, e.g. $A.B.D.E.G$. However, the paths that pass through none or one of these two basic blocks are still feasible, e.g. $A.B.D.F.G$ and $A.C.D.E.G$.*

3.3.1.1 Scopes

Infeasibility constraints can also be associated to a *scope*, which extends Definition 3.1. In such cases, the infeasibility constraint holds only in this scope (i.e. part) of the program. In the control-flow tree model, the scope can be the root tree or any of its direct or indirect children trees.

Example 3.2 (Infeasibility constraints and scope) *Consider Figure 3.3 with the infeasibility constraint $\{B, E\}$. If the scope of the infeasibility constraint is the body of the loop (the Seq child of the Loop tree), then the infeasibility constraint holds for each iteration independently and we apply our transformation technique on the body of the loop. It means that a single iteration of this loop cannot include both B and E, but that it is possible that the concatenated path of several iterations contains all the basic blocks of the infeasibility constraint. In this case, it means that the path $H.B.E$ in a single iteration is not feasible, but the path $H.B.F.H.C.E$, composed of the two iterations $H.B.F$ and $H.C.E$ is feasible.*

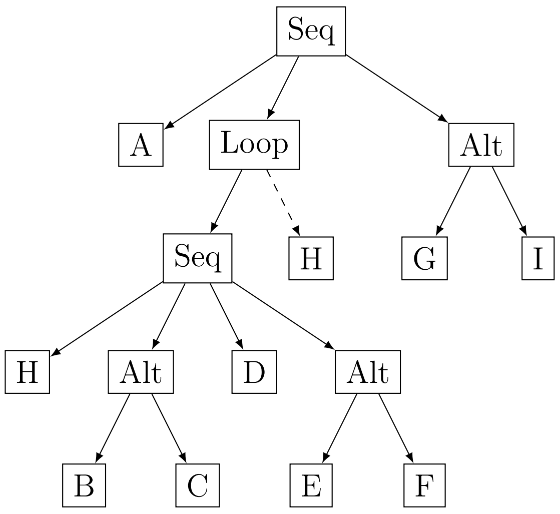


Figure 3.3: Scoped infeasibility constraint

When infeasibility constraints are associated to different scopes, we must apply several times our technique from the most inner scope to the most outer scope.

Example 3.3 (Handling multiple scopes) *Consider again Figure 3.3 with two infeasibility constraints $\{B, G\}$ and $\{B, E\}$, that are associated respectively to the root tree (the root Seq) and to the loop body (the Seq child of the Loop tree). First, we must first apply our technique to the most inner scope tree, i.e. the loop body with the constraint $\{B, E\}$. Then, we apply our technique on the root tree (that includes the transformed loop body) with the constraint $\{B, G\}$.*

In the remaining part of this chapter, we assume that the scope of a constraint is always the root tree, since the technique is easily extendable to the general case (as shown in Example 3.3).

Note that most of the infeasible paths found using existing infeasible path detection tools [HW02; Che+07; GEL06; BLH14; RCM17] can be expressed using this formalism. It is however not possible to express conflicts between distinct loop iterations or to limit the number of times we pass through a basic block like in [Ray14].

3.3.2 Pseudo paths

With the help of infeasibility constraints, we now detail how we can produce an abstract representation of the paths in the CFT. This abstract representation will help us to remove all the infeasible paths from the CFT without having to explicitly enumerate the feasible and the infeasible paths in the program.

First, we begin by defining a flattening function on sets of elements, which is simply a union between all the elements in all the sets:

$$\mathit{flatten}(s_1, \ldots, s_n) = \bigcup_{1 \leq i \leq n} s_i$$

With this definition, we can now define the notion of pseudo path.

Definition 3.2 (Pseudo path) *A pseudo path is a set of basic blocks that represents one or several paths in a program. It represents all the program paths that pass through all of its basic block elements and it is always composed of basic blocks that are included in at least one infeasibility constraint. Let t be a CFT, pp be a pseudo path and cs be a set of infeasibility constraints related to structurally feasible paths in t. Then, the concretization function ϕ associates program paths to the pseudo path representing them as follows:*

$$\phi(pp, t, cs) = \{p | p \in \mathit{tpaths}(t) \land \forall b \in pp, b \in p \ \land \nexists b' \in p, b' \in \mathit{flatten}(cs) \land b' \notin pp\}$$

where $\mathit{tpaths}(t)$ represents the structurally feasible paths in the CFT t.

Less formally, this function denotes that the concrete paths in a CFT t represented by a pseudo path pp consist in all the paths that are structurally feasible in t (a.k.a. paths that belong to tpaths(t)), that:

- pass through all the basic blocks that compose pp;
- do not pass through any other basic block that belongs to an infeasibility constraint.

Thus, note that the empty pseudo path does not represent all the paths in the tree, but only the paths that are not included in any other pseudo path. In other words, the empty pseudo path represents all the paths that do not pass through any basic block that belong to an infeasibility constraint.

Example 3.4 (Pseudo paths and concretization) *Consider the tree of Figure 3.1c, denoted t_1, with the infeasibility constraint $\{B\}$, which means that any path that contains the basic block B is infeasible. Then, consider the pseudo paths $\{B\}$ and $\{\}$. $\{B\}$ represents the paths that pass through B and $\{\}$ all the other paths, that do not pass through B. Thus, the concretization function ϕ gives the following results:*

$$\phi(\{B\}, t_1, \{\{B\}\}) = \{A.B.D.E.G, A.B.D.F.G)\}$$
$$\phi(\{\}, t_1, \{\{B\}\}) = \{A.C.D.E.G, A.C.D.F.G\}$$

3.3.3 Building the pseudo paths of a tree

Now that we defined the notion of infeasibility constraint and the notion of pseudo paths, we explain how a CFT can be built from pseudo paths.

Obtaining the pseudo paths of a tree is similar to finding all the paths in that tree, i.e. with a tree traversal algorithm, except that we do not keep track of all the traversed basic blocks (or leaves) of the tree. We only keep trace of the basic blocks that are included in at least one infeasibility constraint.

We first give the intuition behind the algorithm before formally describing it.

3.3.3.1 Algorithmic intuition

The computation of the pseudo paths of a tree can be done inductively on its structure:

- The basic block is in the pseudo path only if it is included in an infeasibility constraint;
- For a sequence, the pseudo paths are a concatenation of its children pseudo paths;
- For an alternative, the pseudo paths are a union of the pseudo paths of its children;
- For loops, the pseudo paths are a union of the pseudo paths in the body of the loop and the pseudo paths in the exit of the loop.

Example 3.5 (Pseudo paths from tree traversal) *Consider again the CFT in Figure 3.1c. We analyze this control-flow tree recursively with the set of infeasibility constraints $\{\{B, E\}\}$. Starting from the basic blocks, the set of pseudo paths for each basic block is $\{\{\}\}$, except for B and E. Indeed, B and E are included in the infeasibility constraints. Thus, their pseudo paths are respectively $\{\{B\}\}$ and $\{\{E\}\}$. For Alt_1 and Alt_2, we perform a union of the pseudo paths of their children. Their pseudo paths are then respectively $\{\{B\}, \{\}\}$ and $\{\{E\}, \{\}\}$. We then concatenate the pseudo paths of the children of the sequence to obtain the pseudo paths of this CFT. Since A, D and G have only the empty pseudo paths, we only have to create the combinations of paths between the pseudo paths of Alt_1 and Alt_2, and we finally obtain $\{\{\}, \{B\}, \{E\}, \{B, E\}\}$.*

3.3.3.2 Formal algorithm

Now that we presented the intuition behind our algorithm, we give a more formal definition that describes the computation of the pseudo paths of a CFT.

First, we define the notation $\Cup$ for sets of pseudo paths:

$$pset \Cup pset' = \{p \cup p' | p \in pset, p' \in pset'\}$$

In essence, the $\Cup$ operator returns the combinations of the possible concatenations between the two sets of paths. For instance, for two sets of pseudo paths $\{p_1, p_2\}$ and $\{p_3, p_4\}$, it would produce all the possible combinations of path concatenations between the first and the second set of pseudo paths $\{\{p_1 \cup p_3\}, \{p_1 \cup p_4\}, \{p_2 \cup p_3\}, \{p_2 \cup p_4\}\}$.

The function *allPPaths* returns the pseudo paths of a tree. This function is the core of the algorithm to infer pseudo paths from a tree. It takes a set of infeasibility constraints and the control-flow tree and returns the pseudo paths found in this tree for this set of infeasibility constraints.

$$\begin{aligned}
\mathit{allPPaths}(cs, \mathrm{Leaf}(b)) &= \begin{cases} \{\{b\}\} & \text{if } b \in \mathit{flatten}(cs) \\ \{\{\}\} & \text{otherwise} \end{cases} \\
\mathit{allPPaths}(cs, \mathrm{Seq}(t_1, t_2, \ldots, t_n)) &= \mathit{allPPaths}(cs, t_1) \ \Cup \mathit{allPPaths}(cs, \mathrm{Seq}(t_2, \ldots, t_n)) \\
\mathit{allPPaths}(cs, \mathrm{Alt}(t_1, \ldots, t_n)) &= \bigcup_{1 \le i \le n} \mathit{allPPaths}(t_i) \\
\mathit{allPPaths}(cs, \mathrm{Loop}(h, t_b, n, t_e)) &= \mathit{allPPaths}(t_b) \cup \mathit{allPPaths}(t_e)
\end{aligned}$$

Example 3.6 (Pseudo paths computation) *For instance, consider the tree of Figure 3.1c with the infeasibility constraint $\{B, E\}$. We use the leaves B and C and the Alt_1 tree as examples:*

$$\begin{aligned}
\mathit{allPPaths}(\{\{B, E\}\}, \mathit{Leaf}(B)) &= \{\{B\}\} \\
\mathit{allPPaths}(\{\{B, E\}\}, \mathit{Leaf}(D)) &= \{\{\}\} \\
\mathit{allPPaths}(\{\{B, E\}\}, \mathit{Alt}_1) &= \{\{B\}, \{\}\}
\end{aligned}$$

So as to build a feasible CFT from the pseudo paths, we must first remove all the pseudo paths that represent infeasible paths. We thus define a filtering function which takes a set of infeasibility constraints and a set of pseudo paths and returns the set of feasible pseudo paths:

$$\mathit{filterIPaths}(cs, ps) = \{p | p \in ps \wedge \nexists c \in cs, c \sqsubseteq p\}$$

In essence, it simply removes all the pseudo paths for which there exist an infeasibility constraint that is included in the pseudo path. Since a pseudo path represents all the paths that pass through its basic blocks (that belong to infeasibility constraints), all the pseudo paths that include an infeasibility constraint thus represent paths that are actually infeasible in the CFT. Also, since pseudo paths are an abstraction of all the paths in the

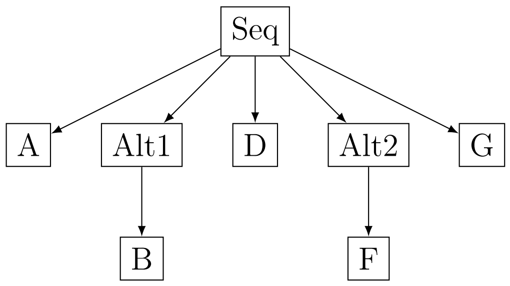


Figure 3.4: Pseudo tree corresponding to the pseudo path $\{B\}$

CFT, removing infeasible pseudo paths removes all the infeasible paths from the program.

Finally, we define the function *pPaths*, which takes a set of constraints and the CFT and returns the set of feasible pseudo paths inferred from this CFT:

$$\mathit{pPaths}(cs, t) = \mathit{filterIPaths}(cs, \mathit{allPPaths}(cs, t))$$

This function simply builds the set of pseudo paths for a tree and removes the infeasible paths from it, using the previously defined functions.

3.4 From pseudo paths to control-flow-trees

In this section, we show how we can create a CFT that contains only feasible paths with the previously computed feasible pseudo paths. First, we define the notion of pseudo tree. Then, we detail how to build a pseudo tree for each feasible pseudo path. Finally, we use these pseudo trees to build the feasible control-flow tree.

3.4.1 Pseudo trees

A *pseudo-tree* is a CFT whose paths corresponds to all the concrete paths represented by a pseudo-path.

Definition 3.3 (Pseudo tree) *Let t be a CFT, cs be a set of constraints, p a pseudo path and pt a pseudo tree. pt is the pseudo tree of t corresponding to the pseudo path p if and only if:*

$$\mathit{tpaths}(pt) = \phi(p, t, cs)$$

Example 3.7 (Pseudo tree) *Consider Figure 3.1c. With the constraint $\{B, E\}$, we have the following list of feasible pseudo paths for the tree: $\{B\}, \{E\}, \{\}$. The pseudo tree for the pseudo path $\{B\}$ is represented in Figure 3.4. In this pseudo tree, Alt_1 has only B as a child since B is in the pseudo path and thus we are forced to pass through this node*

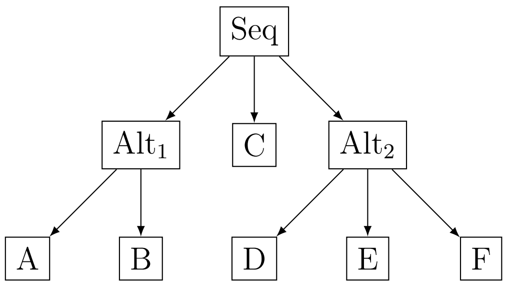


Figure 3.5: A CFT on which the transformations of alternative nodes are easier to detail

to match the pseudo path $\{B\}$. However, E is removed from the pseudo tree because it is in the constraints but not in the pseudo path $\{B\}$, which means that it does not belong to this pseudo path.

3.4.2 Building pseudo trees

Now that we defined the notion of pseudo tree, we explain how such CFTs can be built from a pseudo path. The goal of this step is to derive, from the original CFT, a new CFT that represents only the concrete paths represented by a single pseudo path. In essence, it means that the resulting CFT must be such that:

1. all the paths in the CFT pass through all the basic blocks of the pseudo paths;
2. it does not include any basic block that is a) not in the pseudo path; b) in another pseudo path.

We thus begin with the definition of two preliminary functions that help us with these two requirements.

3.4.2.1 Passing through the basic blocks of the path

The first preliminary function that we define helps for the first requirement. It applies a filter to the set of children trees to an alternative and only keep those that include basic blocks belonging to a given pseudo path:

$$\begin{aligned}&\mathit{findChildren}(\mathit{ppath}, \mathrm{Alt}(t_1, \ldots, t_n), \mathit{cs}) = \\ &\qquad\qquad \{t_k | 1 \leq k \leq n, \exists p \in \mathit{pPaths}(\mathit{cs}, t_k), p \neq \{\} \wedge p \subseteq \mathit{ppath}\}\end{aligned}$$

In essence, *findChildren* finds the children trees that have non empty pseudo paths that are included in a specified pseudo path. In other words, it tries to determine if there are children trees that have at least one path that is included in the pseudo path of interest.

Example 3.8 (*findChildren* **result)** *Consider the tree of Figure 3.5 and in particular Alt_2 on this tree. With the infeasibility constraint $\{A, D\}$, we determine the children of Alt_2 that include basic blocks that must be included in the paths represented by the pseudo path $\{D\}$:*

$$\mathit{findChildren}(\{D\}, Alt(Leaf(D), Leaf(E), Leaf(F)), \{\{A, D\}\}) = \{Leaf(D)\}$$

In essence, this means that we must only keep the child tree Leaf(D) to ensure that all the paths in the built CFT pass through D.

3.4.2.2 Filtering paths that do not belong to a pseudo path

This second preliminary function helps for the second requirement. The trees returned by *findChildren* will be part of the pseudo tree constructed for the considered pseudo path. If the function returns an empty set, we filter sub-trees that contain basic block that belong to other pseudo paths:

$$\mathit{filterAlt}(ppath, \mathrm{Alt}(t_1, \ldots, t_n), cs) = \{t_k | 1 \leq k \leq n, \exists p \in pPaths(cs, t_k), p \subseteq ppath\}$$

Less formally, *filterAlt* returns the children of an alternative tree for which there exists at least one pseudo path that belongs to the specified pseudo path. This function is very similar to the previous one, except that it allows the empty pseudo path.

Example 3.9 (*filterAlt* **result)** *Again, we consider the tree of Figure 3.5 and Alt2 in particular. With the pseudo path $\{A\}$, we have:*

$$\mathit{filterAlt}(\{A\}, Alt(Leaf(D), Leaf(E), Leaf(F)), \{\{A, D\}\}) = \{Leaf(E), Leaf(F)\}$$

The children leaves E and F are not in the infeasibility constraints and thus they can be included in the pseudo tree that corresponds to the pseudo path $\{A\}$, which is included in $\{A\}$. However, the leaf D belong to the pseudo path $\{D\}$ and is not included in the pseudo path $\{A\}$. Thus, we do not include it in the pseudo tree that corresponds the pseudo path $\{A\}$.

3.4.2.3 Pseudo tree construction algorithm

We define below the function *pTree*, which builds the pseudo tree corresponding to a pseudo path. This function takes a pseudo path, the original CFT and the set of constraints, and produces the tree associated with the pseudo path.

$$pTree(ppath, \mathrm{Leaf}(b), cs) = \mathrm{Leaf}(b)$$

$$pTree(ppath, \mathrm{Seq}(t_1, \ldots, t_n), cs) = \mathrm{Seq}(pTree(t_1), \ldots, pTree(t_n))$$

$$pTree(ppath, \mathrm{Alt}(t_1, \ldots, t_n), cs) = \mathrm{Alt}(\bigcup_{1 \le i \le k} pTree(t'_i))$$

$$\text{where } (t'_1, \ldots, t'_n) = \begin{cases} findChildren(ppath, \mathrm{Alt}(t_1, \ldots, t_n), cs) & \text{if } findChildren(ppath, \\ & \mathrm{Alt}(t_1, \ldots, t_n), cs) \neq \{\} \\ filterAlt(ppath, \mathrm{Alt}(t_1, \ldots, t_n), cs) & \text{otherwise} \end{cases}$$

$$pTree(ppath, \mathrm{Loop}(h, t_b, n, t_e), cs) =$$
$$\begin{cases} \mathrm{Loop}(h, pTree(ppath, t_b, cs), n, pTree(ppath, t_e, cs)) & \text{if } \exists p \in pPaths(cs, t_b), \\ & p \subseteq ppath \\ pTree(t_e) & \text{otherwise} \end{cases}$$

Less formally:

- the pseudo tree of a leaf is always the leaf itself;
- the pseudo tree of a sequence is the sequence of the pseudo trees of all its children;
- For the alternatives, two cases can occur:
  - If there is at least a child tree for which there is a non-empty pseudo path that is included in the specified pseudo path, then we build an alternative with these children pseudo trees;
  - Otherwise, it means that there is no basic block that we must traverse in this alternative. Thus, we just ensure that the children trees for which all the pseudo paths include basic blocks that are not included in the pseudo path that we are building the tree of are removed.
- For loops, we simply remove the loop body if all the pseudo paths of this loop body include basic blocks that are not included in the pseudo path we are building the tree of.

3.4.3 Building the feasible control-flow tree

Now that we defined how to build a CFT that corresponds to a feasible pseudo path, the last step is to construct an alternative between all the feasible pseudo trees. This tree is

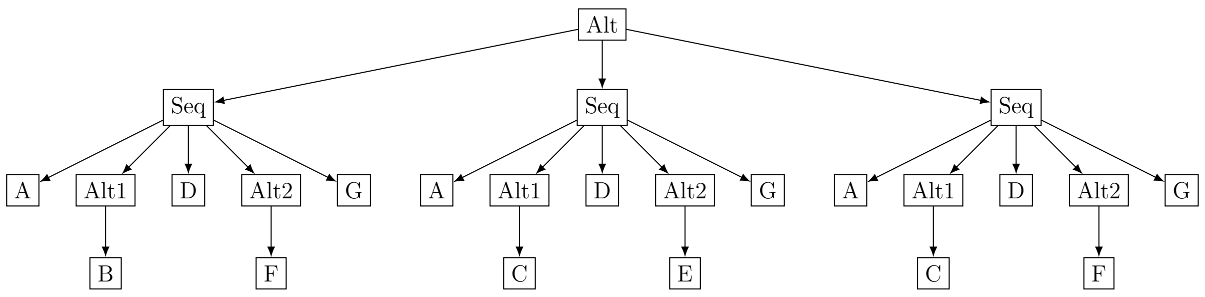


Figure 3.6: The feasible tree of Figure 3.1c

called the *feasible CFT* and is computed as follows:

$$ftree(t, cs) = \text{Alt}\left(\bigcup_{p \in pPaths(cs,t)} pTree(p, t, cs)\right)$$

Example 3.10 (Feasible CFT) *For instance, the tree of Figure 3.6 represents only the feasible paths of the tree of Figure 3.1c, i.e. without the paths passing through B and E. The pseudo trees that correspond to the pseudo paths $\{B\}$, $\{E\}$ and $\{\}$ are the children of a root alternative tree.*

Now we state the validity of our approach:

Theorem 3.1 *Let t be a CFT, cs be a set of constraints and $pt = ftree(t, cs)$. Then:*

$$tpaths(pt) \subseteq tpaths(t) \tag{3.1}$$

$$\nexists p \in tpaths(t) | p \notin tpaths(pt) \wedge \nexists c \in cs, c \sqsubseteq p \tag{3.2}$$

$$\nexists p \in tpaths(pt) | \exists c \in cs, c \sqsubseteq p \tag{3.3}$$

Intuitively, (3.1) expresses that we do not add paths that were not in the original CFT. Equation (3.2) means that all existing feasible paths of the original CFT exist in the feasible CFT. Finally, (3.3) states that no infeasible path remains in the transformed tree. Put together, these properties imply that our approach is *exact* with respect to our infeasible paths abstraction (i.e. infeasibility constraints), in the sense that the resulting CFT corresponds to the original CFT without the semantically infeasible paths.

A proof intuition for this theorem is:

- Regarding property (3.1), when we build the set of pseudo paths, we infer them from the original CFT. Then, we remove from this set the pseudo paths that include an infeasibility constraint, thus we discard some of the paths. Finally, when we build the pseudo trees, we copy the original tree and only discard some sub-trees of this CFT. Thus, the paths in the feasible CFT are definitely included in the initial CFT paths;

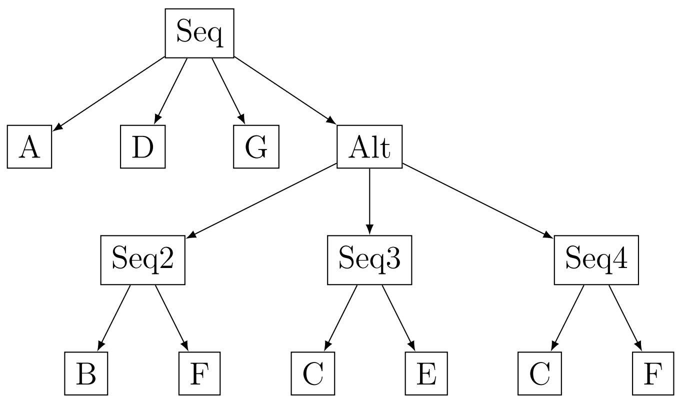


Figure 3.7: Final tree

- Pseudo paths are an abstract representation of the concrete paths in the CFT that pass through certain basic blocks, as denoted by the concretization function. Thus, by removing the pseudo paths that include an infeasibility constraint (and thus are infeasible because of the basic blocks they traverse), we implicitly remove all the concrete paths that were infeasible because they passed through all the basic blocks of the removed pseudo paths, that include at least an infeasibility constraint, which corresponds to property (3.2). Since all the paths that are removed include an infeasibility constraint, we do not remove any feasible path, which corresponds to property (3.3).

3.4.4 Optimization

The tree built by function *ftree* can duplicate many nodes of the original tree. For instance, the tree of Figure 3.6 duplicates the nodes A, D and G. To some extent, we can avoid the duplication of those nodes to get a more compact tree, which still produces the same WCET. We do a simple tree factorization: the nodes that are in all pseudo trees can be taken out of each pseudo tree and put in a parent sequence of the transformed tree.

Example 3.11 (Optimization of feasible CFT) *Consider the CFT in Figure 3.1c. If we have the constraint $\{B, E\}$, the nodes A, D and G have an empty pseudo path $\{\}$. Only the node Alt_1 and the node Alt_2 have non empty pseudo paths in their pseudo paths sets, respectively $\{B\}$ and $\{E\}$. It means that A, D and G exist in all the paths, so we can avoid their duplication by removing them when we transform the tree and putting them back into the final tree. The transformation without factorization returns the tree of Figure 3.6, whereas the tree transformation with factorization is shown in Figure 3.7. As we can see, the nodes A, D and G are not duplicated in each pseudo tree anymore.*

This tree transformation changes the program semantics: it changes the order of the basic blocks. Nevertheless, it is safe regarding WCET analysis if we consider that the

Table 3.1: Benchmarks Summary

Program	**Source(s)**	**Description**
adpcm_dec	TACLeBench	ADPCM decoder
adpcm_enc	TACLeBench	ADPCM encoder
cosf	TACLeBench	Performs calculations of the cosinus function
countnegative	TACLeBench/Mälardalen	Counts signes in a matrix
cover	Mälardalen	Program for testing many paths
expint	Mälardalen	Series expansion for computing an exponential integral function
fft1	Mälardalen	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm
lms	TACLeBench/Mälardalen	LMS adaptive signal enhancement
ludcmp	TACLeBench/Mälardalen	LU decomposition algorithm
md5	TACLeBench	Message digest algorithm
minver	TACLeBench/Mälardalen	Floating point matrix inversion
ndes	TACLeBench/Mälardalen	Complex embedded code
prime	TACLeBench/Mälardalen	Prime number test
qsort-exam	Mälardalen	Non-recursive version of quick sort algorithm
select	Mälardalen	Selects the Nth largest number in a floating point array
statemate	TACLeBench/Mälardalen	Automatically generated code
ud	Mälardalen	Calculation of matrixes

hardware analysis has been performed before the tree transformation. Indeed, as the WCET of a sequence is a commutative pointwise sum of the WCET of its children, changing the order of the children has no impact on the resulting WCET.

3.5 Experiments

In this section, we present the various experiments that we conducted to evaluate our approach. In particular, we evaluate the impact of our technique on the analysis time. We also compare our technique with non-parametric IPET in a parametric context. We begin with the detail of our experimental setup, and then we comment the obtained results.

3.5.1 Experimental setup

We implemented our method with the proposed optimization as an extension of the WSymb [WSy] tool. We tested it with various benchmarks from both the Mälardalen benchmark suite [Gus+10] and TACLeBench [Fal+16]. We selected a subset of these

Table 3.2: Analysis time with and without considering infeasible paths

Program	**SLOC**	**Time w/o ipaths (s)**	**Infeasible paths**	**Time w/ ipaths (s)**
adpcm_dec	397	0.159	8	1.089
adpcm_enc	413	0.173	8	58.356
cosf	631	0.075	122	5.833
countnegative	78	0.029	1	0.037
cover	231	0.018	3	0.026
expint	57	0.024	20	0.041
fft1	136	0.230	2	0.479
lms	146	0.071	2	0.101
ludcmp	71	0.045	784	0.954
md5	352	5.229	3	6.099
minver	136	0.082	405	2.730
ndes	201	0.089	140	435.758
prime	33	0.031	50	0.130
qsort-exam	69	0.039	1168	1.303
select	68	0.038	1178	1.196
statemate	1127	0.449	5468	881.128
ud	61	0.037	57	0.060

benchmarks, presented in Table 3.1, with various source code sizes and complexity. Here, what we mean by complexity is the number of branches in a program.

We used the following hardware setup:

- Processor: no pipeline or cache are used: the execution time of each instruction is set to 5 processor cycles;
- Compilation: each benchmark is compiled as a standalone 32-bit ARM binary file using GCC version 9.2.1 for ARM, with flags -O1 -nostdinc -nostdlib -mtune=cortex-a7 -mfpu=neon-vfpv4 -mfloat-abi=hard -march=armv7ve.
- Analysis time: they are measured using an Intel® Core™ i7-8550U CPU @ 1.80GHz, with 16GB of RAM.

We implemented a tool to generate random infeasibility constraints. This tool is based on a CFG analysis: whenever a node has several successors, we store the identifier of those successors so as to generate constraints between them later. Note that the CFT is obtained from the CFG, so we can generate infeasibility constraints from the CFG in order to compare our method with IPET. For instance, consider the CFG of Figure 3.1b. The nodes A and D have several successors:

- A has B and C as successors;

- D has E and F as successors.

Our tool generates constraints by taking one random node per set of successors. So if we want to generate a constraint of size 2 on this example, we get $\{B, E\}$, $\{B, F\}$, $\{C, E\}$ or $\{C, F\}$. For each benchmark, we generate constraints with two to five basic blocks (the number of basic blocks is chosen randomly). So as to perform experiments that are representative regarding the number of infeasible paths, we used the number of infeasible paths detected by the tool presented in [RCM17] as a reference, which is reported in the *infeasible paths* column in Table 3.2). Our infeasible path generator is still a prototype and has two limitations:

1. It may generate several times the same infeasible paths for small programs because the algorithm generates them randomly.
2. It may generate structurally incorrect infeasible paths. Indeed, a loop header has also several successors but no difference is made between the exit edge and the loop entry edge. In such cases, the incorrect infeasible paths are ignored.

Since when we generate infeasible paths we do not take into account loop bounds, we set the number of iterations of all the loops to 1 to be able to compare with IPET. This is not a problem since the idea here is to compare analysis times and not WCET precision (since our technique is exact).

3.5.2 Experimental results

We now present our experimental results. First, we compare the analysis time of symbolic computation with and without considering infeasible paths. Then, we compare this execution time with implicit path enumeration technique.

3.5.2.1 Analysis time

The results of our experiments regarding the analysis time overhead of the infeasible path removal are shown in Table 3.2. The first column shows the name of the benchmark program. The *SLOC* column corresponds to the number of source lines of code (SLOC) measured with the *sloccount* Linux utility. The *Time w/o ipaths (s)* column presents the analysis time (in seconds) of the tool without using our technique (without considering infeasible paths). The *infeasible paths* column shows the number of infeasible paths generated for the analysis. Finally, the *Time w/ ipaths (s)* indicates the average analysis time in seconds with our infeasible paths transformation technique. As we generate random constraints, we ran each benchmark until we reached fifty successful runs. A run is considered to be successful if there is at least one feasible path, i.e. we removed the results

Table 3.3: Comparison of the results with IPET

Program	**IPET analysis time (s)**	**Parametric analysis time (s)**	**Formula instantiations (s)**	**Parametric < IPET**
adpcm_dec	0.050	1.089	0.012	≥ 29
adpcm_enc	0.062	58.356	0.040	≥ 2653
cosf	0.031	5.833	0.008	≥ 254
countnegative	0.019	0.037	0.009	≥ 4
cover	0.014	0.026	0.007	≥ 4
expint	0.014	0.041	0.007	≥ 6
fft1	0.055	0.479	0.009	≥ 11
lms	0.032	0.101	0.008	≥ 5
ludcmp	0.032	0.954	0.007	≥ 39
md5	0.240	6.099	0.010	≥ 27
minver	0.034	2.730	0.008	≥ 105
ndes	0.041	435.758	0.108	–
prime	0.020	0.130	0.007	≥ 10
qsort-exam	0.036	1.303	0.007	≥ 45
select	0.034	1.196	0.007	≥ 45
statemate	0.409	881.128	0.007	≥ 2192
ud	0.019	0.060	0.008	≥ 6

where no feasible paths were found, which may happen since we generate the infeasible paths randomly. All the analysis times presented include:

- The construction of the CFT (and of the feasible CFT for *Time w/ ipaths (s)*);
- The translation of the CFT into a parametric WCET formula;
- The simplification of this formula.

Three programs show the limitations of our technique: *adpcm_enc*, *ndes* and *statemate*. For these three programs the analysis time explodes because of the number of distinct conditional statements involved, which produce many different pseudo paths and thus many different pseudo trees. Note that existing control-flow graph transformation techniques [MS15; Mus+16] exhibit the same problem. Nevertheless, results suggest that our technique is usable for most medium sized programs, whose number of branches is generally limited.

3.5.2.2 Comparison with implicit path enumeration technique

We compare our technique with IPET in a parametric context. The results are presented in Table 3.3. The first column shows the name of the program. The *IPET analysis time (s)* column gives the time needed to compute the WCET using OTAWA [Bal+10] with

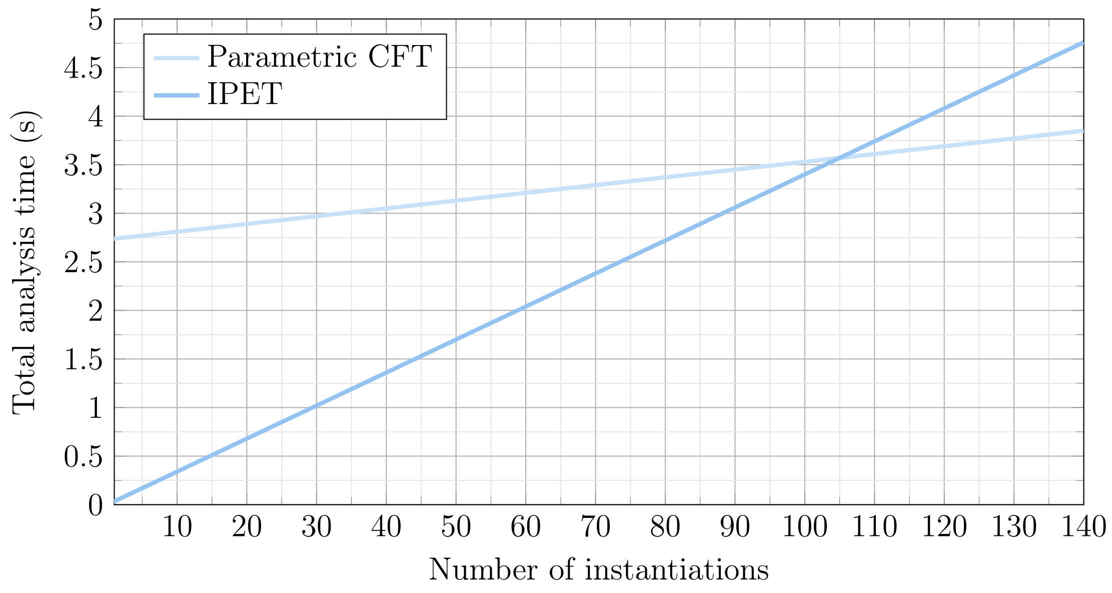


Figure 3.8: Comparison with IPET (linear regression example with minver)

lp_solve [lps] 5, which computes the WCET using IPET. As in most other works using IPET, the infeasible paths are introduced in the analysis by adding linear constraints to the integer linear program computing the WCET [Ray14]. The *Parametric analysis time (s)* column corresponds to the time needed to compute the parametric formula that represents the WCET (it is the same value as the one presented in the last column of Table 3.2). The *Formula instantiations (s)* column is the time needed to compute the numeric WCET bound by replacing parameters with values. Finally, the *Parametric < IPET* column shows the number of instantiations over which the total computation time is lower using our technique. In other words, since IPET is not parametric, we have to run the analysis again each time the parameter values change. This is not the case for the parametric WCET computation technique that we use, which produces a feasible tree once that is then translated into a WCET formula. Then, the formula is instantiated with different values each time the parameter values change.

For all the benchmark programs (except ndes) the instantiation of the formula is faster than the time needed to re-run IPET. It implies that over a certain number of instantiations, the full runtime of our technique is lower than the analysis time of IPET. To better understand the last column of the table and why our approach can be faster than IPET, consider the example of Figure 3.8. It shows two lines corresponding to the time needed to compute the WCET in a parametric context. In the case of IPET, we need to re-run the analysis each time we want to change a parameter value. With our technique, we only need to instantiate the formula each time we want to edit parameters. As shown in Table 3.2, the time needed to compute the parametric formula with infeasible paths is higher than the time needed to compute the WCET with IPET. Thus, in the non parametric case, IPET is more efficient than our technique. Nevertheless, in presence of parameters, Table 3.3 shows that the instantiation of the formula is faster than the WCET computation with IPET. Thus, the total runtime to compute n times the WCET grows slower than the total runtime for IPET.

Results show that, for many benchmarks, our technique is faster than IPET when we need to change parameter values dozens of times. Nevertheless, three benchmark programs show less satisfying results. First, *statemate* is automatically generated code, its code structure contains many occurrences of the same conditional statements, and thus contains many infeasible paths which increases the number of pseudo trees to generate and explains the high analysis time. A human programmer would probably not write code like this. Regarding *ndes* and *adpcm_ enc*, the main problem is that the computed formula is big. Even without our infeasible path technique, the formula simplification is not really efficient since the program contains many loops and thus the output formula is still big after simplification. Adding our infeasible paths representation technique amplifies the problem and thus the final formula instantiation time is higher than when computing the WCET again with IPET. *adpcm_ enc* also has a more complex control-flow than the other programs, which leads to a lot more pseudo paths. The more pseudo paths, the bigger the resulting formula and the longer the analysis.

We did not detail the analysis precision in the results since our infeasible paths representation technique is *exact* (see Theorem 3.1) and thus the WCET gain is the same as with IPET. These results show that our technique can be used to represent infeasible paths in a parametric context, but is not adapted for non parametric usage.

3.6 Conclusion and discussions

In this chapter, we presented a method to take infeasible paths into account in parametric WCET computation. This method consists in transforming the control-flow tree of the program such that semantically infeasible paths become structurally infeasible in the resulting CFT. The technique supports infeasible sets of basic blocks, that is to say a set that contains basic blocks that cannot be all together in the same program path. The technique uses these sets to build an abstract representation of the program paths. This abstract representation of program paths is then used to build a feasible control-flow tree where semantically infeasible paths are also structurally infeasible. The main benefit of this approach is that once the CFT is transformed, we can apply the parametric WCET computation technique proposed in previous works without further modifications. Our experiments show that this method generally scales for medium-sized programs.

Nevertheless, due to the high complexity of eliminating infeasible paths with a CFT unfolding, the analysis takes more time and does not scale as well as the implicit path enumeration technique, which demonstrates a better ability to manage infeasible paths. A future work on this topic could be to find a compromise between the analysis time and the precision of the infeasible paths removal. For instance, it may be possible to use a complexity metric, e.g. the cyclomatic complexity [McC76], as a reference to limit the number of pseudo tree generated. The resulting feasible CFT would be smaller and thus the parametric tree-based WCET computation that account for infeasible paths could scale for more programs.

Chapter 4

Tree-based WCET computation and pipeline modeling

Contents

A well-known limitation of tree-based WCET computation techniques is that they are often not able to model the effect of the hardware components on the produced WCET. In this chapter, we propose to adapt an existing pipeline analysis technique developed on the CFG such that it can be used in tree-based WCET computation.

4.1 Introduction

The computation of the WCET of each basic block of the program is performed during the hardware analysis. One of the hardware components that are modeled to produce a precise WCET is the processor pipeline. Many different approaches model the effect of the processor pipeline on the WCET (see Section 2.3.1). However, these techniques target the control-flow graph (CFG).

In this work, we take advantage of the fact that the CFT is built from the CFG to adapt an existing pipeline analysis to the symbolic WCET computation technique. Essentially, we use the results of the pipeline analysis that stores WCETs on the CFG edges and we report them on the WCETs of the basic blocks of the CFT. Our experimental results show that the WCET computed with the adapted control-flow tree is very close to the WCET produced using IPET.

4.1.1 Motivating example

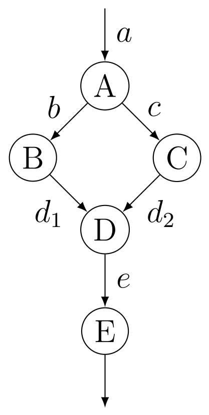
In OTAWA [Bal+10], the *execution graph* approach [RS09] is used to model the processor pipeline. The different WCETs for a basic block are stored on its incoming edges in the CFG. Each of these WCETs represents the WCET of this basic block for a particular *execution context*. An execution context is defined as one or several basic blocks that are executed before the basic block.

Consider Figure 4.1, which represents a simple program with two paths. In the figure, the basic blocks are represented with an upper case letter and their WCET with the corresponding lowercase letter. We distinguish two paths in the CFG: $A.B.D.E$ and $A.C.D.E$. Their WCET is respectively $a + b + d_1 + e$ and $a + c + d_2 + e$. We need to find a way to transfer information from the CFG edges into the CFT.

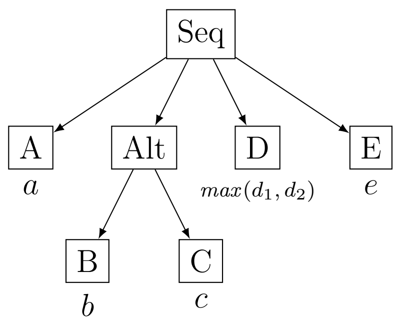
A trivial solution is to set the WCET of D as the maximum between d_1 and d_2, but it could be overly pessimistic. Assume that $a = b = d_2 = e = 10$ and that $c = d_1 = 5$. The WCET of the two possible paths in the CFG is then $10 + 10 + 5 + 10 = 35$ and $10+5+10+10 = 35$. However, with the trivial solution on the CFT, we would then have $10 + max(10, 5) + max(5, 10) + 10 = 40$. This solution is thus not satisfying because each time a node has several predecessors, its WCET can be overestimated, which produces an overly pessimistic result.

```c
int f(int x){
  // A
  if(x > 10)
    // B
  else
    // C
  // D
  // E
}
```

(a) C program



(b) Control-flow graph



(c) Control-fow tree

Figure 4.1: Motivating example

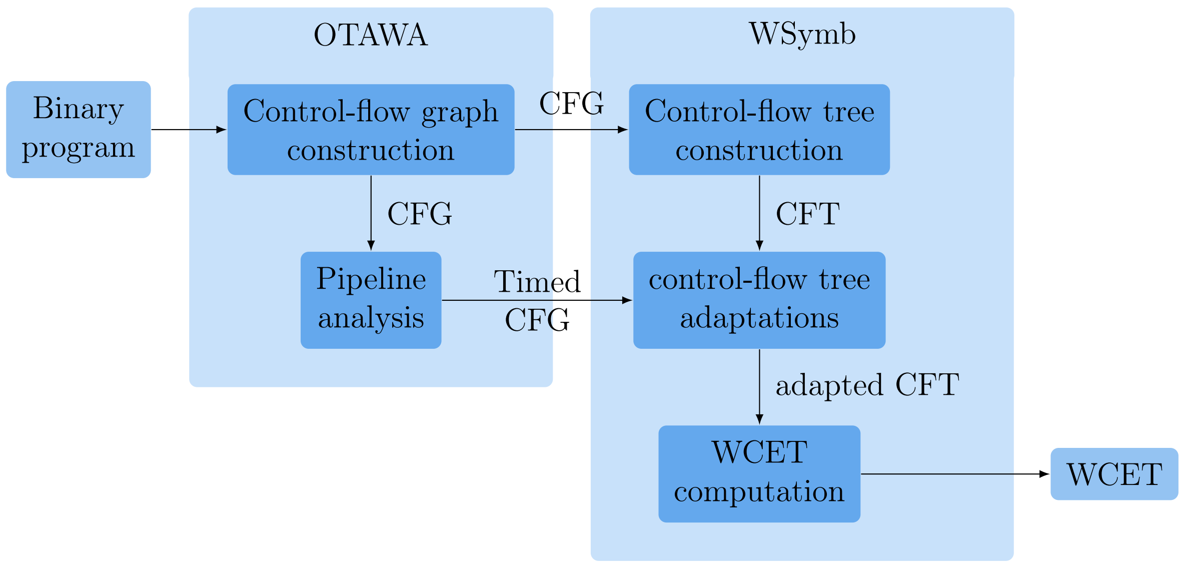

OTAWA
WSymb
Binary program
Control-flow graph construction
CFG
Control-flow tree construction
CFG
CFT
Pipeline analysis
Timed CFG
control-flow tree adaptations
adapted CFT
WCET computation
WCET

Figure 4.2: Workflow of our analysis

4.1.2 Contribution

The goal of this work is to efficiently model the effect of the pipeline in the CFT.

We propose to change the WCET of the basic blocks in the control-flow tree so as to integrate the information provided by execution graph method the different WCETs that a basic block may have. The workflow of our approach is depicted in Figure 4.2:

1. First, OTAWA constructs the control-flow graph from the binary program;

2. Then, the pipeline analysis is performed on this CFG. It computes the WCET of each basic block depending on its execution context and stores the different WCETs for a basic block on its incoming edges;

3. WSymb [WSy], our symbolic WCET computation tool, constructs the control-flow tree from the control-flow graph;

4. Then, we integrate the information from the CFG edges to the CFT basic block WCETs;

5. After that, the WCET is computed normally on the CFT structure.

In the following section, we present the different adaptations applied to the CFT to support the pipeline analysis. Then, we discuss the impact of our technique on the notion of symbol in the symbolic computation in Section 4.3. We present some experimental results in Section 4.4 before concluding in Section 4.5.

4.2 From control-flow graph to control-flow tree

As described in the motivating example, the pipeline modeling implementation in OTAWA stores the different WCET for a basic block on its incoming edges. Thus, in this section, we show various adaptations to the WCET of basic blocks in the control-flow tree that enable us to take the effect of the processor pipeline into account during the WCET computation.

In this section, we use the following notations. Let X be a basic block, $\mathcal{I}(X)$ denotes the set of incoming edges of X. Let e be an edge, $\mathit{source}(e)$ denotes the basic block that is the source of this edge. We denote $\omega_g(e)$ the WCET of the edge e in the CFG (with g for graph). The algorithm relies on two functions:

- $\mathit{loopHeader}(l, e_{\mathit{entry}}, e_{\mathit{back}})$, which determines if the basic block is a loop header of the loop l, with e_{entry} being the entry edge of the loop and e_{back} being the back edge of the loop;

- $\mathit{body}(l)$, which denotes the loop body.

Recall that $\omega(X)$ denotes the WCET of the basic block X in the CFT.

Algorithm 1 details the WCET computation for the basic blocks in the CFT using the WCET of the edges of the CFG. The procedure *ComputeBBWCET* takes the CFT and the CFG as argument and computes the WCET of each basic block in this CFT. Lines 2 to 10 simply call the procedure recursively on the tree structure. Starting from line 11, it computes the WCET of basic blocks. The remainder of this section details how the WCET of these basic blocks is computed.

Algorithm 1 Computing the WCET of CFT basic blocks

```
procedure ComputeBBWCET($t, g$)
    if $t$ is Seq($T$) then                                   ▷ Sequences recursive calls
        for all $t_c \in T$ do
            ComputeBBWCET($t_c$)
    else if $t$ is Alt($T$) then                              ▷ Alternatives recursive calls
        for all $t_c \in T$ do
            ComputeBBWCET($t_c$)
    else if $t$ is Loop($l, t_b, n, t_e$) then                ▷ Loops recursive calls
        ComputeBBWCET($t_b$)
        ComputeBBWCET($t_e$)
    else if $t$ is Leaf($b$) then                             ▷ Basic blocks WCETs
        if $\mathcal{I}(b) = \{i\}$ then                      ▷ Single predecessor
            $\omega(b) \leftarrow \omega_g(i)$
        else if $b$ is $loopHeader(l, e_{entry}, e_{back})$ then   ▷ Loop header WCETs
            if $b \in body(l)$ then                           ▷ Header in the body
                $\omega(b) \leftarrow \omega_g(e_{back})$
            else                                              ▷ Header in the exit tree
                $\omega(b) \leftarrow \omega_g(e_{entry})$
        else                                                  ▷ Successor of alternative
            for all $i \in \mathcal{I}(b)$ do
                if $\exists e \in \mathcal{I}(b) \setminus \{i\} : source(i) \in preds(source(e))$ then   ▷ Artificial leaf
                    $\omega(artificialPred(b)) \leftarrow \omega_g(i))$
                else                                          ▷ Update the predecessor WCET
                    $\omega(source(i)) \leftarrow \omega(source(i)) + \omega_g(i)$
            $\omega(b) \leftarrow 0$
```

4.2.1 Trivial case: basic blocks with a single incoming edge

We begin with the trivial case, which corresponds to basic blocks that have only a single incoming edge. In this case, we simply define the WCET of the basic block (i.e. the leaf in the CFT) to be the WCET stored on its incoming edge in the CFG, which corresponds to lines 12 and 13 in Algorithm 1.

4.2.2 Basic blocks with several incoming edges

Whenever a basic block has several incoming edges, the way the WCET of each basic block is adapted depends on the context. We detail how we adapt the execution times in the CFT for each of these contexts. Two kinds of basic blocks can have several incoming edges in the CFT:

1. Successors to alternative CFTs, that have a different incoming edge for each alternative path in the CFG;

2. Loop headers, that have two predecessors: the loop entry edge and the loop back edge (the edge that mark the end of a loop iteration).

Note that the changes we present in the remainder of this section actually modify the program semantics. However, all these changes preserve the WCET of the whole program.

4.2.2.1 Loops

We now concentrate on the adaptations made for loops. Note that for this part, we focus on loops with a single entry, which represent most of the loops generated by intermediate and high level programming languages. For programs that use multiple entries loops, it is possible to transform the CFG to make sure that all the loops have a single entry using *node splitting* [JC97].

When a loop has a single entry, it also has the following properties:

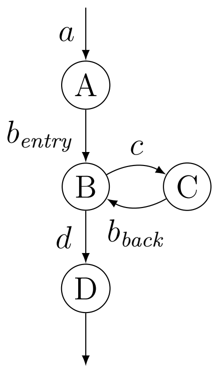
- The loop header has exactly two incoming edges: one when entering the loop (entry edge of the loop) and one when iterating inside the loop (back edge of the loop);
- The entry edge is executed once each time we enter the loop and the back edge is executed one time per iteration over the loop body;
- The CFT contains two times the loop header: one time in the loop body tree, that is executed at each iteration, and another time in the loop exit tree, which is executed a single time.

With these three properties, we can deduce that the WCET of the loop header in the loop body should be the WCET of the back edge because it is executed at each iteration, as in the CFG, and the WCET of the header in the exit tree should be the WCET of the entry edge because it is executed once each time we enter the loop. The way to compute the WCET of the loop header corresponds to lines 14 to 18 in Algorithm 1. In particular, line 16 computes the WCET of the loop header located in the body of the loop and line 18 computes the WCET of the loop header located in the loop exit tree.

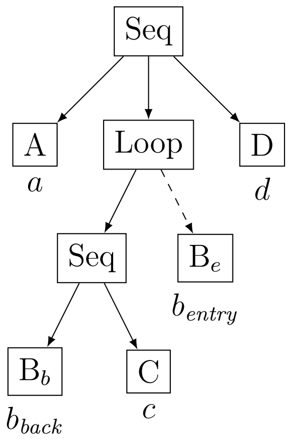
Example 4.1 (Loop header WCET) *Consider Figure 4.3. In this figure, a simple program with a loop is presented. On the CFG, the loop is represented by the basic blocks B and C. The entry edge of the loop from A to B is executed once each time we enter the loop and the back edge of the loop from C to B is executed at each iteration. On the CFT, it is thus correct to set the WCET of B in the exit tree (denoted B_b) to the WCET of the loop entry edge since it is executed once each time we enter the loop. Similarly, we can set the WCET of B in the loop body (denoted B_e) to be the WCET of the back edge*

```c
int f(int x){
  // A
  for(int i=0;
      i<x;
      i++) // B
    // C
  // D
}
```

(a) C program with a loop



(b) Control-flow graph



(c) Control-flow tree

Figure 4.3: Program with loop

of the loop since it is executed at each loop iteration. We thus have:

$$\omega(B_b) = b_{back}$$
$$\omega(B_e) = b_{entry}$$

4.2.2.2 Alternatives

In the CFG, when we have the choice between several paths, it results in an alternative in the CFT. For instance, Figure 4.1 show that in the CFG, we can pass though either B or C between the basic blocks A and D. In such cases, the basic block D, where the two paths converge, can have a different WCET depending on which basic blocks were executed before it. In Algorithm 1, this corresponds to lines 20 to 25.

Alternative between several paths In the case of n paths that converge into a single path at a basic block X, we must consider that the basic block X can have n different WCETs, depending on which basic blocks are executed before it. Instead of setting the WCET of the basic block to the maximum between the WCET stored on incoming edges, we propose to add the WCET stored on incoming edges to the WCET of the source basic block of the edge. This corresponds to line 24 in the algorithm. Then, since the WCET of the basic block has been added to the WCET of its predecessors, its WCET is set to θ, the neutral WCET.

Example 4.2 (Alternative between several paths) *Consider again the CFG of Figure 4.1b. We obtain the CFT of Figure 4.4. Three basic block WCETs were modified: the WCET of the last basic block in each alternative path (B and C here) and the WCET of*

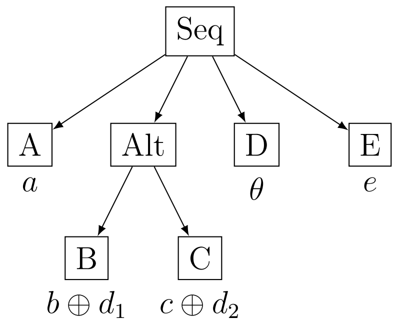
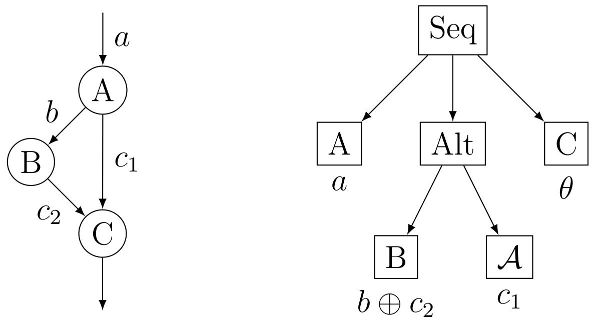


Figure 4.4: Control-fow tree of Figure 4.1c with correct WCETs

```c
int f(int x){
  // A
  if(x > 10)
    // B
  // C
}
```

(a) C program



(b) Control-flow graph (c) Control-fow tree

Figure 4.5: Alternative with a null leaf

the basic block D, where the paths converge. Their WCET is expressed as follows[1]*:*

$$\omega(B) = b \oplus d_1 \tag{4.1}$$

$$\omega(C) = c \oplus d_2 \tag{4.2}$$

$$\omega(D) = \theta \tag{4.3}$$

The case of if-then without else In this case, an edge goes from the basic block before the control-flow split to the basic block after the control-flow converges, the pipelining effect must still be considered. For instance, this can be the case when an *if-then-else* statement has no "else" path. When this happens, an *artificial leaf*, that is to say a leaf with no basic block that has a neutral WCET, is added in the CFT for the else case. Thus, we take the WCET of the edge for the else case and use it as the WCET of the artificial leaf of the alternative tree. This corresponds to line 22 in the algorithm.

Example 4.3 (Optional path in an alternative) *Consider Figure 4.5. In this figure, there is no else statement in the code. As a result, there is a path in the CFG between A*

[1]Recall that the operator $\oplus$ is the WCET sum and that θ represents the neutral WCET, as defined in section 2.7.2

and C without any basic block. In the CFT, this path is represented by an artificial leaf, denoted $\mathcal{A}$. We have:

$$\omega(\mathcal{A}) = c_1$$

4.2.3 Limitation

Until now, we explained how we can handle alternatives and loops to avoid pessimism in the computed WCET. Nevertheless, there is still a limitation in case of a loop with multiple exit edges (due to structure breaking instructions). During the CFT construction, these exit edges are merged (see Algorithm 1 in [BFL17]). When this occurs, we take the maximum of the WCET of these exit edges and set this as the WCET of the basic block after the loop.

4.3 Symbols with pipeline analysis

The introduction of the pipeline support in the control-flow tree brings up the question of symbols. Regarding symbolic loop bounds, nothing changes. However, for symbolic WCETs, which correspond to call to external functions (for which we may not have the code), the impact is non-negligible.

When we have a symbolic WCET for a block of the CFT, we cannot compute statically the impact of the previously executed basic blocks on the symbolic part of the code or the impact of the symbolic part of the code on the following basic blocks. Thus, our solution is to set the WCET of the basic block without considering the pipelining effect. Also, for the basic blocks executed after the symbolic block and that have this symbolic block in their context, we propose to compute their WCET without considering the inter block pipelining effect. The resulting WCET is then overestimated, but still safe.

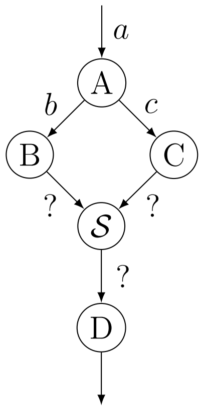
Example 4.4 (Symbolic WCET) *Consider Figure 4.6. In this figure, we consider an execution context of 1 basic block for the pipeline analysis. The code in $\mathcal{S}$ has an impact on the WCET of the basic block D, thus we set the WCETs as follows:*

$$\begin{aligned}\omega(\mathcal{S}) &= \chi \\ \omega(D) &= WCET(D)\end{aligned}$$

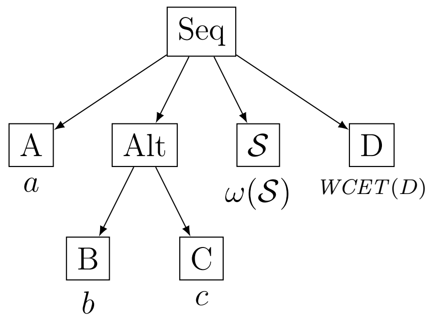
where χ is a symbolic value and $WCET(b)$ gives the WCET without considering the inter block pipelining effect for the basic block b.

```
int f(int x){
  // A
  if(x > 10)
    // B
  else
    // C
  // Symbolic WCET
  // D
}
```

(a) A function with a symbolic WCET



(b) Control-flow graph



(c) Control-flow tree

Figure 4.6: Symbolic WCET in presence of pipeline analysis

4.4 Experiments

In this section, we present the experiments that we conducted to evaluate our adaptations of the pipeline modeling to the control-flow tree representation. We first introduce our experimental setup. Then, we detail our benchmark selection process and finally we discuss the results regarding analysis time and precision.

4.4.1 Experimental setup

We implemented our method into the WSymb [WSy] tool. 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;
- Since the instruction cache implementation differs in WSymb and in Otawa, we disabled it in order to provide a comparison only for the pipeline modeling itself;
- 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: they are measured on an Intel® Core™ i7-8550U CPU @ 1.80GHz × 8 with 16 GB of RAM.

4.4.2 Benchmark selection process

We used the benchmark programs from TACLeBench [Fal+16], which is the reference regarding WCET analysis. Among the 54 programs, 15 could not be analyzed for the following reasons:

- 2 programs are not supported because of recursion: *fac* and *recursion*. This has to do with OTAWA;
- 9 programs because of the incomplete support for division instructions (*adpcm_ dec*, *adpcm_ enc*, *ammunition*, *cjpeg_ transupp*, *epic*, *h264_ dec*, *huff_ enc*, *quicksort* and *susan*). This also has to do with OTAWA;
- *bitonic* and *prime* are not supported by OTAWA: the analysis simply crashes with an error related respectively to the integer linear program that is unbounded and to the execution graph implementation;
- *dijkstra* and *lms* are not supported by WSymb, respectively because of a shortcoming in the control-flow tree construction implementation and a shortcoming in the simplification procedure.

We run our prototype on all the other programs, summarized in Table 4.1.

4.4.3 Experimental results

We compared our adaptations with the original approach, which uses the CFG and IPET, implemented in OTAWA. Table 4.1 summarizes the results regarding the whole WCET analysis. The *Program* column indicates the name of the analyzed program. For each of these programs, the analysis used the main function as entry point. The *analysis time (s)* column give the execution times in seconds for both the original implementation with implicit path enumeration technique (*IPET*) and *WSymb* with our adaptations. The *WCET* column reports the computed WCET, in processor cycles, of *IPET* and *WSymb*. The *Diff (%)* column corresponds to the pessimism of the WSymb WCET over the IPET WCET. Note that "< 0.1" in this column means that the rounded difference between the two WCET would give 0.0%, but we use this notation to indicate the presence of a very slight pessimism. Since IPET does not support parametric WCET computation, both the techniques were used without parameters to ensure a fair comparison between the two of them.

Analysis time Regarding the analysis time, both techniques show very close results for most programs because they are quite small. For the most complex programs to analyze,

Table 4.1: Experimental results

Program	**Analysis time (s)**		**WCET**		
	IPET	**WSymb**	**IPET**	**WSymb**	**Diff (%)**
audiobeam	1.068	0.800	13112533	13112579	< 0.1
binarysearch	0.090	0.068	12888	12888	0.0
bitcount	0.087	0.090	191072	195196	2.2
bsort	0.060	0.069	5570507	5571894	< 0.1
cjpeg_wrbmp	0.357	0.315	2480339	2480433	< 0.1
complex_updates	0.105	0.788	30117	30117	0.0
cosf	0.299	0.291	389919	389919	0.0
countnegative	0.093	0.081	268426	268426	0.0
cubic	23.463	10.314	54215426	54215426	0.0
deg2rad	0.075	0.059	93091	93091	0.0
fft	0.192	0.116	1429513437	1429513437	0.0
filterbank	0.184	0.135	149892341	149892341	0.0
fir2dim	0.214	0.147	116397	116397	0.0
fmref	2.641	1.646	14412706	14412734	< 0.1
g723_enc	0.901	0.675	19895553	19895553	0.0
gsm_dec	1.286	1.058	91539472	91613467	0.1
gsm_enc	2.712	2.371	118766285	118828185	0.1
huff_dec	0.367	0.373	8488352	8498324	0.1
iir	0.087	0.078	29348	29348	0.0
insertsort	0.068	0.077	38929	38929	0.0
isqrt	0.079	0.100	13307772	13307772	0.0
jfdctint	0.152	0.192	67717	67717	0.0
lift	0.271	0.245	17080513	17114303	0.2
ludcmp	0.151	0.210	158166	158556	0.2
matrix1	0.077	0.052	219691	219691	0.0
md5	5.337	4.959	807971215	807971236	< 0.1
minver	0.260	0.312	80819	83295	3.1
mpeg2	–	51.791	–	222513057311	–
ndes	0.347	0.321	972769	972769	0.0
petrinet	0.491	0.526	37153	37191	0.1
pm	0.824	0.887	178532565	178532565	0.0
powerwindow	4.447	2.396	1233059	1233059	0.0
rad2deg	0.037	0.045	92835	92835	0.0
rijndael_dec	1.481	1.679	67532368	67536867	< 0.1
rijndael_enc	1.488	1.744	65091507	65096007	< 0.1
sha	0.525	0.505	60055513	60162012	0.2
st	0.212	0.246	2650931	2650931	0.0
statemate	0.750	0.769	1233059	1368959	11.0
test3	–	–	–	–	–

i.e. *cubic*, *md5* and *powerwindow*, the analysis performed by WSymb is faster. This has to do with the WCET computation technique: IPET rely on integer linear programming, which has an exponential complexity, while WSymb rely on tree-based computation, which has a lower complexity that depends on the size of the control-flow tree.

We considered that the analysis times out if it lasts more than 2 hours. This is the case for *test3* for both IPET and WSymb. Nevertheless, *test3* is an artificial program that was created only to stress the WCET analysis tools. It is thus not very representative of real-life programs. Regarding *mpeg2*, the analysis time exceeded 2 hours only for IPET while WSymb computed the WCET in less than one minute. This brings to light the difference in terms of complexity between integer linear programming and tree-based arithmetic computations.

These results are not very surprising since our method consists only into moving the WCET of certain basic blocks to add them to the WCET of other basic blocks, which does not seem to be complex computations. However, it shows that it is possible to model the effect of the pipeline in a tree-based control-flow representation and still benefits from the low complexity of the tree-based WCET computation.

WCET precision As for the execution times, the two analysis show very similar results regarding the computed WCET, except that the WCET computed by WSymb is never lower than the WCET computed by IPET. Still, our method and the CFT construction can introduce pessimism.

On average, WSymb produces a WCET 0.47% greater than the WCET produced by IPET. Still, some other programs exhibit the limitation that WSymb is subject to: the CFT construction algorithm that may add some paths in the CFT that do not exist in the CFG and the inability to identify a predecessor when a loop has several exit edges. The highest pessimism that we observed is on *statemate*. This program contains many structure breaking instructions both inside and outside of loops, which explains the quite high pessimism. Note also that this program was generated by a statechart code generator and thus may not be considered as a well-written program.

When comparing the WCET produced by IPET and WSymb, our technique shows promising results and demonstrates that it is possible to efficiently take into account the effect of the processor pipeline in a tree-based WCET computation technique.

4.5 Conclusion and discussions

In this chapter, we presented a technique that enables to take into account the effect of the pipeline in a tree-based WCET computation technique. First, the pipeline analysis, which computes the execution times of basic blocks depending on the basic blocks that are executed before it, is performed on the control-flow graph. Then, we showed that it is possible to use the resulting WCET information, stored on the edges of the control-flow graph, to adapt the WCET of the basic blocks in the control-flow tree so as to take into account the pipelining effect.

The experiments that we conducted to evaluate our method showed great results. In particular, they demonstrate that it is possible, with a slight pessimism, to take into account the effect of the pipeline on the WCET computation in a tree-based WCET analysis.

The main benefit of this technique is that it enables to use tree-based WCET computation techniques as a realistic alternative to IPET. Indeed, at the cost of a slight pessimism, the complexity of the WCET computation is reduced, which enables to analyze more complex programs. Furthermore, contrary to IPET, the control-flow tree representation is well-suited for symbolic WCET computation and thus this technique supports parametric WCET computation without introducing too much pessimism.

Chapter 5

Procedure arguments as parameters

Contents

Our last contribution focuses on the parametric aspect of the analysis. In this chapter, we propose an approach that use the program input, i.e. the program arguments, as parameters. The presented technique automatically detects and characterizes the impact of the procedure argument values on the WCET of this procedure.

5.1 Introduction

Many different parametric approaches to the WCET problem produce a formula that depends on parameters. Most of these techniques simply put parameters that represent the maximum number of times loops may iterate. Altmeyer et al.[Alt+08] proposed to automatically infer the parameters from the program. In their approach, a parameter is defined as a value to which the program reads before it writes to. The problem of both these techniques is that it requires knowledge of the program from the user:

- In the first type of approaches, authors assume that the user has knowledge about all the loops in the program;
- In the technique presented by Altmeyer et al., the user should know to which value the parameter correspond, which also require knowledge about the program.

The goal of our work is to eliminate the need for the user to have knowledge about the program to analyze: we propose to use the arguments passed to the program, that the user must know to run this program, as parameters. Our technique automatically infer the impact of these arguments on the control-flow of the program. Then, this information is used to produce a parametric formula that can be instantiated to determine the WCET of a program depending on its argument values.

5.1.1 Motivating example

We motivate our work with the example of Figure 5.1. This procedure is part of an implementation of the G.723 speech encoding standard, taken verbatim from TACLe-Bench [Fal+16].

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

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

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

Figure 5.1: Speech encoding, reconstructing the difference signal

step `y` (line 5), the reconstructed difference is set to 0 (line 6[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 5 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.

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 instantiation function takes almost as much time as the potential maximum variability (see line *g723_enc_reconstruct* in Table 5.4 in Section 5.7.3.3). However, other more complex functions show a much larger variability and computing the formula on-line may make sense for those functions (see Section 5.7 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.

[2]`dql` is signed, in two's complement, which explains the test at line 6.

5.1.2 Contribution

Our technique is based on two previous works on symbolic WCET computation [BFL17] and abstract interpretation [Bal+19]. Although our methodology relies on foundations presented in these two papers, many novel contributions and extensions were necessary to make it work in a coherent and automatic way. In this chapter, we present these extensions as well as some use cases of our approach, which include modular WCET analysis.

First, we devise an analysis that infers *input conditionals*, which are predicates on procedure arguments that serve as branch conditions, either in conditional statements or in loops. This analysis extends the relational abstract interpretation on binary code of Ballabriga et al. [Bal+19] and is presented in Section 5.4.

Second , we extend the symbolic WCET computation technique to support these input conditionals:

1. We introduce a new kind of tree into the CFT model to represent conditional branches subject to input conditionals in Section 5.5.1;
2. Section 5.5.2 extends the symbolic WCET computation to support formulae with input conditionals;
3. New extensive simplification procedures to reduce the size of the resulting formulas are presented in Section 5.5.3;
4. A new compiler compiles formulas to optimized C code that has a low and easy-to-compute WCET, to evaluate the formula on-line (Section 5.5.4).

Third, we introduce an extension to this approach to perform modular WCET analysis, in Section 5.6, that extends both the abstract interpretation and the WCET computation parts.

We demonstrate with our experiments on TACLeBench that our approach is simultaneously **adaptive**, **embeddable** and **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.

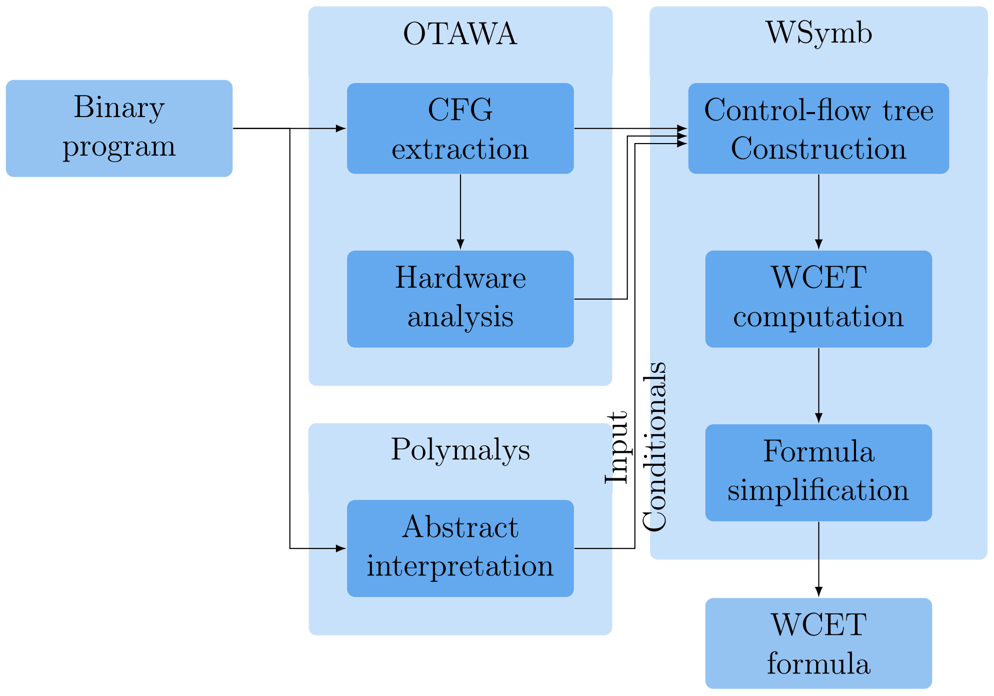


Figure 5.2: Workflow of our approach

5.2 Overview

The workflow of our approach is depicted in Figure 5.2. We illustrate this workflow on the program of Figure 5.3. Starting from the binary code of function f, the analysis consists of the following steps.

CFG extraction The binary code is translated into a control-flow graph, where nodes are basic blocks and edges represent the control-flow of the program. We obtain a CFG with basic blocks A to G. We rely on OTAWA [Bal+10] for this step.

Hardware analysis The hardware analysis infers the WCET of each basic block. We 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 respectively $\omega(B)$, $\omega(D)$). We rely on the method presented in Chapter 4 for this step.

Inferring input conditionals The abstract interpreter identifies the value stored in r_0 (a.k.a. n) as an argument of procedure f at line 1 (as per function call conventions). At line 7, it infers $r_0 \geq 11$ as the *input conditional* for branching to label *L2* (a.k.a. block B) and $r_0 \leq 10$ if we do not branch. Similarly, the input conditionals $r_0 \geq 0$ and $r_0 \leq -1$ are inferred at line 16. We extend the abstract interpretation analysis of [Bal+19] to infer input conditionals on conditional branches and loops which depend on function arguments

```
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 */
  @ ...               @  }
```

(a) Procedure code

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
B \arrow[dr] & & C \arrow[dl] \\
 & D \arrow[dl] \arrow[dr] & \\
E \arrow[dr] & & F \arrow[dl] \\
 & G &
\end{tikzcd}$$

(b) Control-flow graph

Figure 5.3: A procedure with an argument

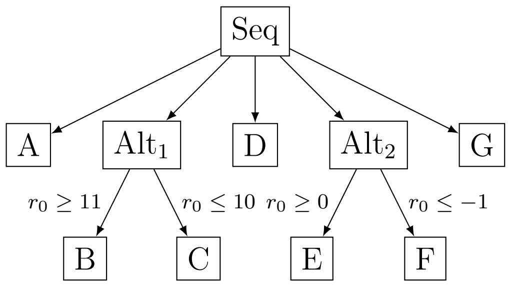


Figure 5.4: Control-Flow Tree for function f of Figure 5.3

(see Section 5.4).

CFT with symbolic input conditionals The CFG is translated into the Control-Flow Tree (CFT) depicted in Figure 5.4. It consists of a sequence (the root node *Seq*) of basic blocks (A, D, G) and of alternatives (*Alt₁*, *Alt₂*) between two sub-trees (B or C, resp. E or F). Output edges of alternative nodes are annotated with the input conditionals inferred by the abstract interpreter. We extend the CFT of [BFL17] with a new type of alternative node to model conditional branches with input conditionals (see Section 5.5.1).

WCET formula The CFT is translated into a WCET formula. The WCET of each alternative sub-tree 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:

$$10 \oplus (((r_0 \geq 11) \circledast \omega(B)) \uplus ((r_0 \leq 10) \circledast 5)) \oplus \\ \omega(D) \oplus (((r_0 \geq 0) \circledast 10) \uplus ((r_0 \leq -1) \circledast 10)) \oplus 5$$

The new operator $\circledast$ is introduced in Section 5.5.2.

Formula simplification With the introduction of the new $\circledast$ operator, new simplification rules are added to the existing rules. We obtain:

$$25 \oplus (((r_0 \geq 11) \circledast \omega(B)) \uplus ((r_0 \leq 10) \circledast 5)) \oplus \omega(D)$$

First, we simplified $(((r_0 \geq 0) \circledast 10) \uplus ((r_0 \leq -1) \circledast 10))$ to 10, since $r_0 \geq 0$ and $r_0 \leq -1$ are mutually exclusive and multiply the same value (10). Then, we used commutativity to gather and reduce constant values ($10 + 5 + 10 = 25$).

It is important to underline that, for the sake of clarity, in this example we show a simplified version of the formula with integers (recall that we use a more complex representation with lists of values, as presented in Section 2.7.2). Unfortunately, we could not reuse classical simplification procedures for integer formulae since WCETs are represented by lists. Instead, we establish and prove the correctness of our own simplification rules. This work is described in Section 5.5.3.

Formula instantiation The formula is instantiated when symbolic values become known. For instance, assuming $n = 0$ (i.e. $r_0 = 0$), $\omega(B) = \omega(D) = 8$, 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 5.5.4, 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.

5.3 Abstract interpretation of binary code

In this section, we recall the main concepts of the abstract interpretation procedure presented in [Bal+19]. *Abstract interpretation* [CC77] 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 [Bal+19] because we want to inject the inferred invariants into our WCET analysis, which is applied to binary code. We summarize the main features of this interpretation procedure below.

5.3.1 Polyhedra

We begin with a quick reminder about the definition of a 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*. 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.

One important operation is the ability to do a *projection* of a polyhedron p on a subset $\mathcal{V}' = \{x_0, \ldots, x_n\}$ 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 this 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 5.4 to explain how we treat conditionals for building a parametric formula.

Example 5.1 (Projection operation) *This example illustrates the projection operation:*

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

5.3.2 Abstract state

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

Example 5.2 (Abstract state) *In the following abstract state, register r_0 contains a positive value and address 7872 contains a value greater than that of r_0:*

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

5.3.3 Interpretation procedure

The procedure proceeds by forward abstract interpretation [CC77] applied to ARM A32 programs. A program P is represented as a sequence of labeled instructions $l_0 : I_0, l_1 : I_1, \ldots, l_n : \mathit{END}$, where I_k is the instruction at label l_k $(0 \leq k \leq n)$ and END terminates the program. The result $M = \mathit{interpret}(P)$ maps each label to the abstract state immediately *before* the execution of the corresponding instruction. An important specificity of this interpretation procedure is that the mapping between dimensions and data-locations can change as the interpretation progresses.

Example 5.3 (Register mapping) *Table 5.1 details the abstract states at several program points of the program of Figure 5.3. We assume that the value of n is not modified*

Table 5.1: Abstract states at several program points in Figure 5.3

line	**Polyhedron**	**Registers**	**Memory**
2	$p_2 = \langle\rangle$	$\mathcal{R}^\sharp_2 = \{r_0 : x_0, fp : x_1\}$	
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	$\mathcal{R}^\sharp_6 = \{r_0 : x_4, r_3 : x_2, fp : x_1\}$	$*^\sharp_4$
8	$p_8 = p_4 \sqcap_\diamond \langle x_2 \leq 10\rangle$	$\mathcal{R}^\sharp_6$	$*^\sharp_4$
11	$p_{11} = p_4 \sqcap_\diamond \langle x_2 > 10\rangle$	$\mathcal{R}^\sharp_6$	$*^\sharp_4$

in the program. At line 4, the register r_0 contains the value of the integer n, which corresponds to the dimension x_0 in the polyhedron. However, at line 6, r_0 is mapped to x_4, which means that the register r_0 has a new value, which is different from the value n.

A *transition function* $sem(I) : \mathcal{A} \to \mathcal{A}$ updates the abstract state of the analysis to represent the effect of an instruction I on the data locations of the program. When a branching occurs, the branches are analyzed separately, and a *join* operation $\mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is executed when the branches converge in a single program point (e.g. after an if-then-else for instance). Since program may contain loops, a *widening* operator ensures that the analysis reaches a fixpoint.

5.4 Infering input conditionals

In this section, we extend this abstract interpretation 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.

5.4.1 Identifying procedure arguments

By convention [Arm23], 32-bit ARM programs pass the first four arguments of a procedure call through registers r_0, r_1, r_2 and r_3. 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 polyhedron 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 5.4 (Identifying procedure arguments) *In the program of Figure 5.3, we identify the polyhedron dimensions to which r_0 is associated in the abstract state at line 1, that is to say x_0. Now assume that line 4 changes the value of r_0 to perform some computations. Thus, r_0 is mapped to another dimension: x_4. As a result, 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.*

5.4.2 From polyhedra to input conditionals

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

5.4.2.1 Conditionals statements

When the interpreter analyzes 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 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 5.5 (Inferring branch conditions) *In figure 5.3, in the abstract state at line 8 of Table 5.1, the register r_3 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 conditional, we project the polyhedron over the variable x_0:*

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

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

- 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 to an unconditional branch, i.e. the input conditional is set to *true*.
- If the projected polyhedron has unsatisfiable constraints (empty polyhedron), the branch target is dead code, thus the input conditional is set to *false*.

5.4.2.2 Loop bounds

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

Let p denote the polyhedron of the abstract state at the loop header, obtained after the abstract interpretation has reached a fixpoint. Let *args* denote the set of procedure argument variables, and g denote the variable mapped to the ghost register. The function $\mathit{lbound}(p, \mathit{args}, g)$ computes the loop bound as follows. First, it computes $p' = \mathit{proj}(p, \mathit{args} \cup \{g\})$. From there, two cases can occur:

- p' contains exactly two inequalities where the ghost register variable appears: one of them indicates that the ghost register variable is positive (a loop bound is always positive) and the other one is the loop bound.

- Otherwise, we are not able to compute a loop bound and it must be provided by the user.

Example 5.6 (Inferring loop bounds) *The code of a function f consisting of a simple loop is detailed in Figure 5.5a. The abstract states at the entry of the first and the second iterations of the loop are detailed in Figure 5.5b. The ghost variable gr is added to the register mapping and its value is set to 0 at the entry of the loop. The additional constraints of p_2 show the evolution of the abstract state after one iteration. The value of the ghost register is incremented at each iteration, as shown in the abstract state at the second loop iteration. At the end of the loop interpretation, the state of the loop contains a bound to the value of the ghost register. Thus, assuming that x_0 has been identified as a procedure argument, $\mathit{proj}(s, \{x_0, \mathcal{R}^\sharp(gr)\}) = \langle \mathcal{R}^\sharp(gr) \geq 0, \mathcal{R}^\sharp(gr) \leq x_0 \rangle$, where s is the state of the loop header after the interpretation of loop ended. We thus keep $\mathcal{R}^\sharp(gr) \leq x_0$ as the loop bound and we replace $\mathcal{R}^\sharp(gr)$ such that we have $lb \leq x_0$.*

5.5 Symbolic WCET with input conditionals

In this section, we detail how we extend the symbolic WCET computation approach from [BFL17] to support input conditionals.

```
 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]     @        // when entering the loop
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                 @  }
```

(a) Assembly and C code of a loop

(b) Loop input conditionals

#it	**Polyhedron**	**Registers**	**memory**
1st	$p_1 = \langle x_3 = x_1 - 16, x_5 = x_1 - 8, x_8 = x_1 - 12, x_2 = x_0, x_4 = 1, x_6 = 0, x_7 = x_6, x_9 = 0\rangle$	$\mathcal{R}^\sharp_1 = \{r_0 : x_0, fp : x_1, r_3 : x_6, gr : x_9\}$	$*^\sharp_1 = \{x_3 : x_2, x_5 : x_4, x_8 : x_7\}$
2nd	$p_2 = p_1 \sqcap_\diamond \langle x_{10} = 2 \times x_4, x_{11} = x_{10}, x_{12} = x_7 + 1, x_{13} = x_{12}, x_{14} = x_{13}, x_{15} = x_2, x_{16} \geq 0, x_{16} \leq x_9 + 1, x_{14} < x_{15}\rangle$	$\mathcal{R}^\sharp_2 = \{r_0 : x_0, fp : x_1, r_2 : x_{14}, r_3 : x_{15}, gr : x_{16}\}$	$*^\sharp = \{x_3 : x_2, x_5 : x_{11}, x_8 : x_{13}\}$

Figure 5.5: Abstract states at the beginning of the loop (line 9)

5.5.1 Control-flow tree with input conditionals

We extend the previous definition of alternative trees so that an input conditional is associated to each alternative. This input conditional indicates whether the alternative is feasible or not, depending on its valuation and thus on the procedure argument values.

Definition 5.1 (Deterministic alternative) *Let* $(t_1, \dots, t_n)$ *be a set of control-flow trees,* $(e_1, \dots, e_n)$ *be a set of input conditionals and* $1 \leq k \leq n$*. The* deterministic alternative[3] *tree* $Alt(e_1 \to t_1, \dots, e_n \to t_n)$ *represents an alternative between the execution of one tree among* $(t_1, \dots, t_n)$*, such that the tree* t_k *can be executed only if* e_k *is* true*.*

Example 5.7 (Deterministic alternative) *Figure 5.4 depicts the CFT obtained for the program of Figure 5.3. We can see that the input conditional* $r_0 \geq 11$*, whose inference was detailed in Example 5.5, appears as an input conditional to execute* B *in the deterministic alternative tree* Alt_1*. It means that the alternative* B *can be executed only if* $r_0 \geq 11$ *is* true*.*

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

Example 5.8 (Loop with linear expression as bound) *The loop* $Loop(l, t_1, 4 \times r_0 + r_1, t_2)$ *represents a loop identified by* l*, that executes* $4 \times r_0 + r_1$ *times the tree* t_1 *and exits by executing the tree* t_2*.*

5.5.2 WCET formulas with input conditionals

We define a new operator $\circledast$ that multiplies a WCET by an input conditional. It has a higher precedence than $\oplus$ and $\uplus$ operators, but a lesser precedence than the other operators. It is used to compute the WCET of a deterministic alternative tree:

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

Definition 5.2 ($\circledast$ operator) *Let* e *be an input conditional and* w *be an abstract WCET.*

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

Example 5.9 ($\circledast$ operator) *The sub-tree* Alt_1 *of Figure 5.4 is translated into the formula* $(r_0 \geq 11) \circledast \omega(B) \uplus (r_0 \leq 10) \circledast (\top, \{5\})$*. This corresponds to* $\omega(B)$ *if* $r_0 \geq 11$*, or to* $(\top, \{5\})$ *otherwise.*

[3]Note that input conditionals might not be mutually exclusive, thus we use the term "deterministic" in the sense that input conditionals add a kind of determinism to the alternative CFT

Commutativity

$$(e_k \wedge e_l) \circledast w_1 \mapsto (e_l \wedge e_k) \circledast w_1 \qquad \text{if } e_l \lhd e_k \tag{5.1}$$

$$e_k \circledast w_1 \oplus e_l \circledast w_2 \mapsto e_l \circledast w_2 \oplus e_k \circledast w_1 \qquad \text{if } e_l \lhd e_k \tag{5.2}$$

$$e_k \circledast w_1 \uplus e_l \circledast w_2 \mapsto e_l \circledast w_2 \uplus e_k \circledast w_2 \qquad \text{if } e_l \lhd e_k \tag{5.3}$$

Factorization

$$e_k \circledast w_1 \oplus e_l \circledast w_1 \mapsto w_1 \qquad \text{if } e_l \Leftrightarrow \neg e_k \tag{5.4}$$

$$e_k \circledast w_1 \uplus e_l \circledast w_1 \mapsto w_1 \qquad \text{if } e_l \Leftrightarrow \neg e_k \tag{5.5}$$

$$e_k \circledast w_1 \oplus e_l \circledast w_2 \mapsto e_k \circledast (w_1 \oplus w_2) \qquad \text{if } e_k \Leftrightarrow e_l \tag{5.6}$$

$$e_k \circledast w_1 \uplus e_l \circledast w_2 \mapsto e_k \circledast (w_1 \uplus w_2) \qquad \text{if } e_k \Leftrightarrow e_l \tag{5.7}$$

$$e_k \circledast w_1 \oplus (e_k \wedge e_l) \circledast w_2 \mapsto e_k \circledast (w_1 \oplus e_l \circledast w_2) \tag{5.8}$$

$$e_k \circledast w_1 \uplus (e_k \wedge e_l) \circledast w_2 \mapsto e_k \circledast (w_1 \uplus e_l \circledast w_2) \tag{5.9}$$

Multiplication

$$e_k \circledast \mathit{0} \mapsto \mathit{0} \tag{5.10}$$

$$e_k \circledast w_1 \mapsto \mathit{0} \qquad \text{if } e_k \Leftrightarrow \mathit{false} \tag{5.11}$$

$$e_k \circledast w_1 \mapsto w_1 \qquad \text{if } e_k \Leftrightarrow \mathit{true} \tag{5.12}$$

$$e_k \circledast (e_l \circledast w_1) \mapsto e_k \circledast w_1 \qquad \text{if } e_k \Leftrightarrow e_l \tag{5.13}$$

Loops

$$(e_k \circledast w_1)^{it,l} \mapsto e_k \circledast (w_1)^{it,l} \tag{5.14}$$

Figure 5.6: Rewriting rules with input conditionals

5.5.3 Simplifying WCET formula

The size of the resulting formula is linear in the size of the CFT. To reduce the size of this formula, we apply a simplification procedure that follows the classic integer arithmetic simplification strategy described in [Coh03].

5.5.3.1 Simplification rules

The new simplification rules for WCET formulae that contain input conditionals are detailed in Figure 5.6. e_k and e_l are input conditionals, 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 [BFL17]. For each rule of the form $l \mapsto r$ we must prove that $l = r$. We illustrate the general proof principle for rule (5.8) below. The equivalence proofs of l and r for all these rules can be found in Appendix A.

This proof is a 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*). We then demonstrate that $l = r$ for all the possible values of e_k and e_l.

Proof of rule (5.8). 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: $e_k = 0$

$$\begin{aligned}
&0 \circledast w_1 \oplus (0 \wedge e_l) \circledast w_2 = \theta \oplus 0 \circledast w_2 = \theta \\
&0 \circledast (w_1 \oplus e_l \circledast w_2) = \theta
\end{aligned}$$

2. Case: $e_l = 0$

$$\begin{aligned}
&e_k \circledast w_1 \oplus (e_k \wedge 0) \circledast w_2 = e_k \circledast w_1 \oplus 0 \circledast w_2 = e_k \circledast w_1 \\
&e_k \circledast (w_1 \oplus 0 \circledast w_2) = e_k \circledast (w_1 \oplus \theta) = e_k \circledast w_1
\end{aligned}$$

3. Case: $e_k = e_l = 1$

$$\begin{aligned}
&1 \circledast w_1 \oplus (1 \wedge 1) \circledast w_2 = w_1 \oplus 1 \circledast w_2 = w_1 \oplus w_2 \\
&1 \circledast (w_1 \oplus 1 \circledast w_2) = 1 \circledast (w_1 \oplus w_2) = w_1 \oplus w_2
\end{aligned}$$

□

Factorization rules require to test the equivalence of input conditionals. For distributivity rules, we rely on an order relation $\lhd$ on input conditionals 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 $\circledast$.

5.5.3.2 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 a *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 5.10 (Normal form of input conditionals) *The normal form of the input conditional* $10 \geq 15 + r_1 + r_0$ *is* $0 \leq -r_0 - r_1 - 5$.

5.5.3.3 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:

$$\begin{aligned} e_k \lhd e_l \Leftrightarrow &(\mathrm{lid}(e_k) < \mathrm{lid}(e_l)) \vee \\ &(\mathrm{lid}(e_k) = \mathrm{lid}(e_l) \wedge \mathrm{size}(e_k) < \mathrm{size}(e_l)) \vee \\ &((\mathrm{conj}(e_k) = \mathit{false} \wedge \mathrm{conj}(e_l) = \mathit{false}) \wedge \\ &(\mathrm{lid}(e_k) = \mathrm{lid}(e_l)) \wedge (\mathrm{size}(e_k) = \mathrm{size}(e_l)) \wedge \\ &(\mathrm{linconst}(e_k) < \mathrm{linconst}(e_l))) \end{aligned} \tag{5.15}$$

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 5.11 (Order relation functions) *Consider the input conditional* $0 \leq r_0 + r_1 + 10 \wedge 0 \leq r_2$. *We have:*

$$\begin{aligned} \mathit{lid}(0 \leq r_0 + r_1 + 10 \wedge 0 \leq r_2) &= 0 \\ \mathit{size}(0 \leq r_0 + r_1 + 10 \wedge 0 \leq r_2) &= 6 \\ \mathit{linconst}(0 \leq r_0 + r_1 + 10) &= 10 \\ \mathit{conj}(0 \leq r_0 + r_1 + 10 \wedge 0 \leq r_2) &= \mathit{true} \end{aligned}$$

5.5.4 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 compiled and 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 use, we ensure that the C compiler optimizations can 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.

These rules ensure that we can easily compute the WCET of the formula evaluation, and that this WCET is very low. An example of instantiation code produced for the procedure *g723_enc_reconstruct* is shown in Appendix B.

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.

5.6 Towards modular WCET analysis

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.

5.6.1 Modular abstract interpretation

In our previous abstract interpretation analysis, procedure calls were inlined. Instead, in this section we detail a modular abstract interpretation analysis, which relies on the extensions previously presented in this chapter. Each procedure is analyzed only once per program analysis instead of analyzing it each time it is called, which decreases the overall analysis complexity. The 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 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.

We make a few additional definitions before detailing the analysis. First, 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. A procedure $p \in \mathcal{P}$ is defined as a sequence of labeled instructions $l_0 : I_0, \ldots, l_n : END$.

Algorithm 2 Summary construction

```
1: function ConstructSummary($P$)
2:     $\mathcal{A}^\sharp \leftarrow \{r_0 : x_0, r_1 : x_1, r_2 : x_2, r_3 : x_3\}$
3:     $\mathcal{A}^\sharp_1 \leftarrow \mathcal{A}^\sharp$
4:     $s \leftarrow (\top, \mathcal{A}^\sharp_1, \emptyset)$
5:     $(p_P, \mathcal{R}^\sharp_P, *^\sharp_P) \leftarrow interpret(s, P)$
6:     $p_s \leftarrow proj(p_P, \mathcal{A}^\sharp \cup \{\mathcal{R}^\sharp_P(r_0)\})$
7:     return $(p_s, \mathcal{A}^\sharp, \mathcal{R}^\sharp_P)$
```

5.6.1.1 Procedure summary

In the 32-bit ARM convention [Arm23], the value returned by a procedure is stored in register r_0. The summary of a procedure is defined as a tuple $(p, \mathcal{A}^\sharp, \mathcal{R}^\sharp)$, where:

- p is a polyhedron that represents the abstract state of the analysis at the end of the procedure interpretation;
- $\mathcal{A}^\sharp$ is an argument mapping, that associates a variable in p to each procedure argument stored in a register before the execution of f;
- $\mathcal{R}^\sharp$ is a register mapping.

Algorithm 2 details the summary construction. Line 2 constructs a register mapping that associate to each procedure argument (stored in register r_0 to r_3) a polyhedron variable. Line 4 builds an abstract state, that specifies the initial state of the analysis before the interpretation of the procedure. Line 5 run the interpretation on the procedure P with, as initial state, the abstract state constructed at line 4. Since the only value that can change during a pure procedure execution is its return value, and that this value only depends on the procedure arguments, we perform a projection at line 6 to only keep the constraints that express the return value depending on the procedure argument values. Finally, line 7 creates and returns the procedure summary.

Example 5.12 (Procedure summary) *Consider procedure* add_nozero *in Figure 5.7, which is a pure function. Its return value depends on its two input arguments. Note that to ease understanding the assembly code is simplified compared to what the compiler would actually produce. The procedure is summarized as:*

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

In other words, $arg_1 + arg_2 \leq return\_value \leq arg_1 + arg_2 + 1$.

```
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	$\mathcal{R}_1^\sharp = \mathcal{A}^\sharp = \{r_0 : x_0, r_1 : x_1\}$
3	$p_3 = \langle x_2 = x_0 + x_1 \rangle$	$\mathcal{R}_3^\sharp = \mathcal{R}_1^\sharp[r_2 : x_2]$
5	$p_5 = p_3 \sqcap_\diamond \langle x_2 = 0 \rangle$	$\mathcal{R}_5^\sharp = \mathcal{R}_3^\sharp$
6	$p_6 = p_5 \sqcap_\diamond \langle x_3 = x_2 + 1 \rangle$	$\mathcal{R}_6^\sharp = \mathcal{R}_3^\sharp[r_2 : x_3]$
7	$p_7 = \langle x_0 + x_1 \leq x_2 \leq x_0 + x_1 + 1 \rangle$	$\mathcal{R}_7^\sharp = \mathcal{R}_3^\sharp$
8	$p_8 = p_7 \sqcap_\diamond \langle x_4 = x_2 \rangle$	$\mathcal{R}_8^\sharp = \mathcal{R}_7^\sharp[r_0 : x_4]$

Figure 5.7: 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	$p_{1'}$	$\mathcal{R}_{1'}^\sharp = \{r_0 : x_5, r_1 : x_6, r_2 : x_7\}$
3	$p_{3'} = p_{1'} \sqcap_\diamond \langle x_8 = x_5 + x_6 \rangle$	$\mathcal{R}_{3'}^\sharp = \mathcal{R}_{1'}^\sharp[r_3 : x_8]$
5	$p_{5'} = p_{3'} \sqcap_\diamond \langle x_9 = x_7, x_{10} = x_8 \rangle$	$\mathcal{R}_{5'}^\sharp = \mathcal{R}_{3'}^\sharp[r_0 : x_{10}, r_1 : x_9]$

Figure 5.8: A procedure that calls *add_nozero*

Algorithm 3 Summary instantiation

```
1: function InstantiateSummary($(p_s, \mathcal{A}_s^\sharp, \mathcal{R}_s^\sharp), (p, \mathcal{R}^\sharp, *^\sharp)$)
2:     $(p_t, \mathcal{A}_t^\sharp, \mathcal{R}_t^\sharp) \leftarrow \mathit{fresh}((p_s, \mathcal{A}_s^\sharp, \mathcal{R}_s^\sharp))$
3:     $p'_t \leftarrow p_t$
4:     for all $a \in \mathit{Dom}(\mathcal{A}^\sharp)$ do
5:         $p'_t \leftarrow p'_t[\mathcal{R}^\sharp(a)/\mathcal{A}_t^\sharp(a)]$
6:     $p' \leftarrow p \sqcap_\diamond p'_t$
7:     $\mathcal{R}_1^\sharp \leftarrow \mathcal{R}^\sharp$
8:     for all $r \in \mathit{Dom}(\mathcal{R}_t^\sharp)$ do
9:         if $\mathcal{R}_t^\sharp(r) \in p'$ then
10:            $\mathcal{R}_1^\sharp \leftarrow \mathcal{R}_1^\sharp[r : \mathcal{R}_t^\sharp(r)]$
11:    return $(p', \mathcal{R}_1^\sharp, *^\sharp)$
```

5.6.1.2 Summary instantiation

The created summary can then be instantiated into an abstract state. We denote $p[x_i/x_j]$ the polyhedron resulting from the substitution of variable x_j by x_i in p. It substitutes x_j by x_i in the polyhedron constraints (or does nothing when $x_i = x_j$).

The instantiation of a procedure summary into an abstract state is detailed in Algorithm 3, which takes the procedure summary and the abstract state at the procedure call as arguments. First, we create a *fresh* summary at line 2, that is to say that we create a copy of the polyhedron and the mappings, where all the variables are substituted by new variables. From line 3 to line 5, we substitute the summary polyhedron variables that represent procedure arguments by the actual argument variables at the call site. At line 6, we perform an intersection between the previous polyhedron and the polyhedron at the call site. The last step of the algorithm, from line 7 to line 10 creates an updated version of the abstract state register mapping, where the registers are mapped to the new variables that represent their value after the procedure interpretation. Line 11 creates and returns a new abstract state that take the procedure interpretation into account. It uses the new polyhedron as well as the new register mapping. Since we work on pure functions that do not have any side effect, the memory mapping is not updated.

Example 5.13 (Summary instantiation) *The procedure* caller *of Figure 5.8 calls the procedure* add_nozero *at label 5. By instantiating the summary obtained in Example 5.12, we obtain the abstract state* $(p_{6'}, \mathcal{R}_{6'}^\sharp, *_{6'}^\sharp)$ *at label 6 of* caller, *with:*

$$p_{6'} = p_{5'} \sqcap_\diamond (\langle x'_0 + x'_1 \leq x'_4 \leq x'_0 + x'_1 + 1\rangle[x_{10}/x'_0, x_9/x'_1])$$
$$\mathcal{R}_{6'}^\sharp = \mathcal{R}_8^\sharp[r_0 : x'_4, r_1 : x_9]$$
$$*_{6'}^\sharp = \{\}$$

Note that for any n, x'_n denotes the fresh variable corresponding to x_n in the summary.

5.6.1.3 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 about how this argument passed to the callee depends on the arguments of the caller.

Definition 5.3 (Call predicate) *Let f be a procedure that contains a* call *instruction that calls a procedure g at label l_i. Let $M = interpret(f)$, $(p, \mathcal{R}^\sharp, *^\sharp) = M[l_i]$. We denote A_f the set of dimensions identified as f procedure arguments. Let $args_{g_i}$ be such that $args_{g_i}(j)$ represents the value of the $(j+1)^{th}$ argument passed to g at call site l_i*[4]*. The* call predicate *$cpred_{g_i}(j)$ of $args_{g_i}(j)$ at l_i is defined as:*

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

where export substitutes $A_f(k)$ by the identifier f_k for all the k arguments of f, $args_{g_i}(j)$ by the identifier g_i_j, and lists the set of constraints of the polyhedron.

Example 5.14 (Call predicate) *Consider the procedure* caller *in Figure 5.8. 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 $cpred_{add\_nozero}(1) = \{add\_nozero_1 = caller_2\}$

5.6.2 Modular WCET analysis

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

5.6.2.1 Procedure calls and control-flow trees

In [BFL17], a call to another procedure inlines the CFT of the callee in the CFT of the caller. Instead, for our modular WCET analysis, we introduce a new kind of tree to represent a procedure call.

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

Definition 5.4 (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 $m_k = cpred_{\mathcal{f}}(k)$ for $1 \leq k \leq n$.*

The abstract WCET of a call is defined as:

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

where $\mathcal{f}$ identifies the WCET formula of f.

Example 5.15 (Call CFTs WCET formula) *Consider the* caller *procedure in Figure 5.8, which calls procedure* add_nozero*. The WCET of* caller *is expressed as:*

$$w_1 \oplus \mathcal{add\_nozero}(add\_nozero_0 = caller_0 + caller_1, add\_nozero_1 = caller_2) \oplus w_2$$

where w_1 and w_2 are the WCET of the instructions before and after the procedure call. $\mathcal{add\_nozero}(add\_nozero_0 = caller_0 + caller_1, add\_nozero_1 = caller_2)$ is a reference to the WCET formula of function add_nozero *with $caller_0 + caller_1$ as first argument, i.e. the sum of the two first arguments passed to the* caller *formula, and $caller_2$ as second argument, i.e. the third argument passed to the* caller *formula.*

5.6.2.2 Simplification of formulas and instantiation

We instantiate sub-formulas during formula simplification. To do so, we update input conditionals for all the loops and all the alternatives so that they depend on arguments of the caller rather than on arguments of the callee. More formally: let n be the number of arguments. We introduce the following simplification rule:

$$\mathcal{f}(m_1, \ldots, m_n) \mapsto inst(\mathcal{f}, p, Dom(p))$$

where[5]:

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

Example 5.16 (Sub-formula instantiation) *Consider the two procedures* caller *and* add_nozero *of Figure 5.7 and Figure 5.8. Assume that the WCET of* add_nozero *is:* $w_3 \oplus ((\mathit{add\_nozero}_1 + \mathit{add\_nozero}_2 = 0) \circledast w_4) \oplus w_5$.

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

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

5.7 Evaluation

In this section we present the experiments that 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.

5.7.1 Experimental setup

We implemented our approach as an extension to WSymb [WSy]. 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;

- L1 instruction cache: 64KB, LRU replacement policy, 1-way. The miss penalty is 10 cycles;

[5]Recall that function *lbound* was defined in Section 5.4.2.2

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

5.7.2 Benchmark selection

We run our experiments on the TACLeBench benchmarks suite [Fal+16]. 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 other programs;

- OTAWA does not handle well procedures with switch-cases, thus we do not use such procedures;

- 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 leads to an explosion of the number of dimensions in the polyhedron.

Among the remaining procedures, we present only 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 contain 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.

5.7.3 Procedure arguments as parameters

We begin by the evaluation of our technique without the modular extension[6].

We compare our results with those of non-parametric IPET (from OTAWA [Bal+10]) as a reference. We were unable to reproduce the results of the related works on parametric WCET analysis because the prototypes are unavailable, and their results are insufficiently detailed to enable a proper comparison. Also, these related works can only analyze parametric loop bounds. Our work is the first to consider parametric conditional statements.

5.7.3.1 WCET adaptivity

Table 5.2 summarizes our results regarding WCET adaptivity. 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 in percentage (in the *Diff* column).

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 procedures 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 procedures 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.

[6]An artifact can be used to obtain our result, it is available at https://gitlab.cristal.univ-lille.fr/sgrebant/rtns_2023_artifact.

Table 5.2: WCET adaptivity (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

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 [BFL17] 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).

5.7.3.2 Analysis time

The analysis times of IPET and our technique are presented in Table 5.3. The *IPET* column exhibit the analysis time with IPET. The *CFT* sub-columns indicate the analysis time for our technique: *Polyhedra* indicates the time spent in abstract interpretation, while *Symbolic WCET* indicates the time spent in 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 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.

5.7.3.3 Embeddability

The size of the initial and simplified formulae are reported in Figure 5.9. 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 5.3: 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

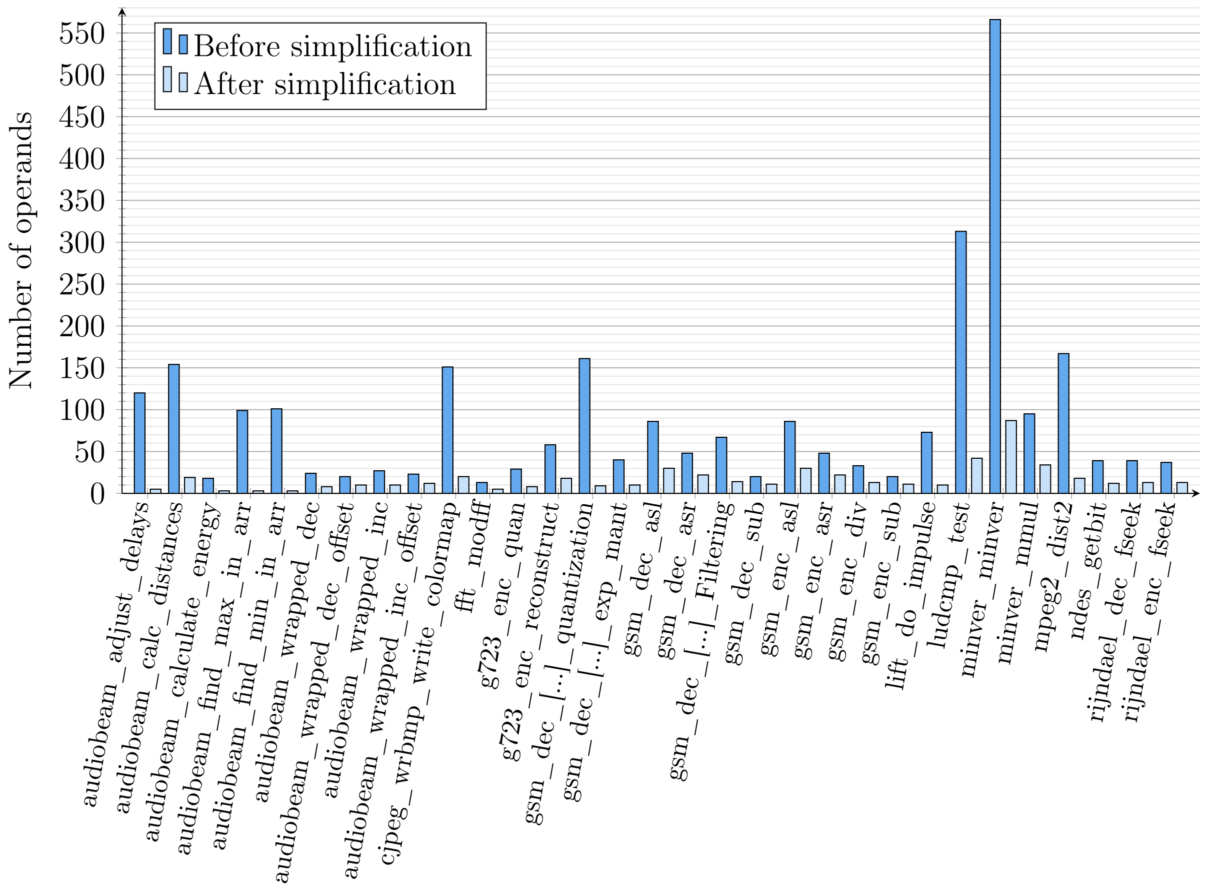


Figure 5.9: Parametric WCET formula size before and after simplification

Table 5.4: Instantiation times (in cycles)

Procedure	**Instantiation**	**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

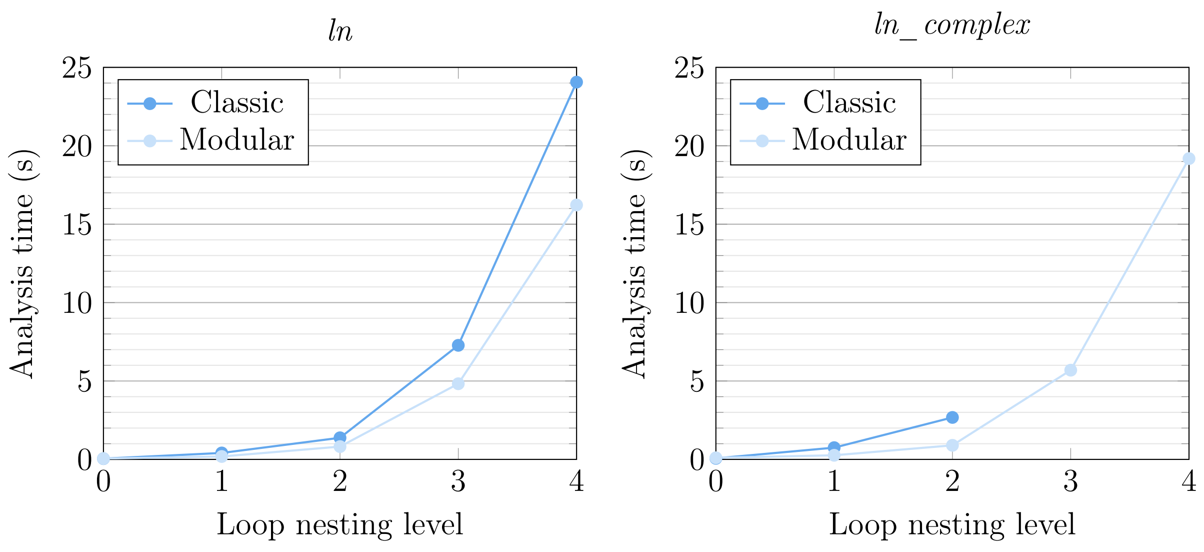


Figure 5.10: Comparison between classic and modular analysis time (in seconds)

Table 5.4 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 5.2. *Op* reports the number of operands in the formula, from Figure 5.9.

On-line instantiation can be considered only when *Max gain* is significantly larger than *Instantiation*. This is the case for most procedures of Table 5.2, 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.

5.7.4 Modular WCET analysis

We use two artificial programs[7], *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 only 4 additions, while *ln_complex* calls a procedure that has a more complex control-flow with conditional statements.

In abstract interpretation, the code located inside of a loop is analyzed repeatedly until a fixpoint is reached. Thus, increasing the loop nest level stresses the analysis. The number of times the procedure is analyzed is exponential in the number of nested loops

[7]The source code of these programs can be found at https://gitlab.cristal.univ-lille.fr/sgrebant/artificial-benchmarks.

(even though widening is applied to speedup convergence).

Figure 5.10 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 slower. This is due to the overhead for computing the procedure summary and instantiating it, which is not performed in the non-modular approach. 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.

5.8 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 focus on practical implementation aspects.

5.8.1 Semi-clairvoyant mixed-criticality scheduling

Recently, adaptive scheduling has gained interest following work on semi-clairvoyant scheduling for mixed-criticality systems [ABB19]. The system model is based on the *dual-criticality* model of Vestal [Ves07], where a system has two distinct criticality levels, LO (for *low*) and HI (for *high*). The workload consists of a set of tasks defined as $\{\tau_i(\chi_i, [C_i^L, C_i^H]), T_i\}_{0 \leq i < n}$, 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 WCETs 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 *jobs*, of the task[8].

[8][ABB19] 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)

```
1 void schedule() {
2   saveContext();
3
4   if(goBackToLo())
5     resumeAllLo();
6   selectNextTask();
7
8   restoreContext();
9 }
```

(b) Scheduler code

Figure 5.11: Implementing semi-clairvoyant mixed-criticality scheduling

In *semi-clairvoyant scheduling*, the WCET of a job is estimated at its release. This estimate $\gamma_{i,j}$ is less or equal to either C_i^L or C_i^H. The system starts in LO-criticality mode, where every job 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 5.11 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-criticality 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 resumes 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.

There is a slight difference between the implementation proposed in Figure 5.11 and the theoretical semi-clairvoyant model: in Figure 5.11, 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

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

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

Figure 5.12: Implementing an adaptive control task

function and into the callback function of the periodic timer of the task. This timer is the actual time triggered for new job releases. Its callback is usually triggered by interruptions and is thus not delayed by the scheduler. The pros and cons of both options (at release time or at start time) should be explored in future works.

5.8.2 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 5.1. The parameter-space is often large, making control law computation very intensive. Implementing such adaptive control in real-time systems induces a trade-off between control precision and computation time.

Figure 5.12 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.

5.9 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 conditionals using abstract interpretation. Then, based on these input conditionals, the analysis produces a parametric WCET formula that depends on the procedure argument values. We also detailed a modular version of the analysis, that supports pure procedures. Experiments show that our automatic approach is adaptive and embeddable. We also illustrated how this approach can be used to implement adaptive real-time systems.

There is still room for improvement. First, the complexity of the polyhedra analysis is quite high and thus is not well-suited to analyze bigger procedures or programs. A first future work is to reduce the complexity of this analysis, or to explore the other solutions to trade precision for analysis time. For instance, by changing the abstract domain used for a less expressive abstract domain might result in a better trade-off between analysis time and the amount of input conditionals that the abstract interpretation infer. A second future work is to extend the modular abstract interpretation so as to add support for non-pure functions. The main challenge lies in summarizing functions with side effects at the binary level.

Chapter 6

Conclusion and perspectives

Contents

In this chapter, we summarize the different contributions presented in this thesis. We also present research perspectives that can improve our contributions.

6.1 Summary

In this thesis, we formalized, implemented and evaluated three extensions to parametric tree-based WCET computation. In particular, we first introduced a technique to remove infeasible paths from a tree-based program representation. Then, we presented a method to adapt an existing pipeline analysis technique to tree-based WCET computation. Finally, we proposed an approach to automatically take into account the effect of input values on the WCET of a program.

6.1.1 Infeasible paths elimination

In chapter 3, we presented a technique that starts from the control-flow tree representation and removes all the semantically infeasible paths, that are structurally feasible, in order to produce a feasible control-flow tree. First, pseudo paths abstract the paths of the control-flow tree such that the paths are grouped by the basic blocks that they traverse that belong to infeasible paths. Then, the infeasible paths are removed from the set of pseudo paths of the program, which implicitly remove all the infeasible paths. For each remaining pseudo path, a pseudo tree, which represents the concrete paths that correspond to this pseudo paths, is built. A control-flow tree without infeasible paths is then built using an alternative between all these pseudo trees.

We implemented a prototype, which showed that this approach remains tractable for medium-size programs. We also showed that in a parametric context, this analysis can be more efficient that IPET for many of these programs.

6.1.2 Pipeline effect modeling

In chapter 4, we presented various adaptations to an existing pipeline analysis, such that this analysis can be used with tree-based symbolic WCET computation. This pipeline analysis initially stores the WCET of the basic blocks on their incoming edges in the CFG. Our technique transfers these WCETs from the CFG to the CFT thus computing the WCET of the basic blocks in the CFT.

We implemented this technique in our prototype and compared the results with the original approach, implemented with IPET. Our results showed that the resulting WCETs are very close to those computed with IPET.

6.1.3 Procedure arguments as parameters

In chapter 5, we studied the impact of procedure arguments on the WCET of programs. First, we use abstract interpretation to infer input conditionals, i.e. predicates that depend on the value of procedure arguments that represent either conditional statement branching conditions or loop bounds. Then, we extended the control-flow tree representation in order to support these input conditionals. A new product operator enables to integrate the input conditionals into the parametric WCET formula. This formula is then simplified with the new rules that we introduced. Finally, the formula is compiled to C code. Instantiating this formula produces the WCET for the given parameter values.

Our experiments showed that the WCET of programs vary most of the time from 30% to 70% depending on the procedure argument values. They also demonstrate that the produced formula can be used for on-line adaptive scheduling techniques since the instantiation time of the WCET formula is very small compared to the execution time of most of the procedures.

6.2 Perspectives

Although this thesis proposes contributions that enhance parametric WCET computation, is also brings to light several interesting problems.

6.2.1 Tree transformation complexity

A first problem is related to the representation of the program that is most of the time a graph or a tree. The various contributions that focus on transforming the program representation, including our work on infeasible paths, reveal a high complexity. In the case of infeasible paths, it is due to the explosion of the size of the program representation, which slows down the WCET computation. A starting point to limit this explosion would be to find a trade-off between the analysis time and the precision. For instance, we could decide whether or not a tree should be transformed based on a complexity metric like the cyclomatic complexity. This would limit the size of the program representation as well as the resulting WCET precision.

6.2.2 Abstract domains

As demonstrated in our experiments in chapter 5, the polyhedra analysis is quite complex. In particular, the analysis time of procedures with many memory accesses is often very high. We also noticed that some procedures did not rely on predicates that required the polyhedral domain. As a consequence, it could be interesting to study other abstract domains with a lower complexity such as octagons [Min06]. With octagons, we may not be able to detect as much predicates but we could run the analysis on bigger programs that contain more memory accesses.

6.2.3 Modular abstract interpretation

Our modular WCET analysis, presented in chapter 5 has shown that it is possible to perform a modular abstract interpretation, i.e. to analyze each function separately, for pure procedures. Nevertheless, we are currently not able to analyze a non-pure function and to summarize it. The difficult part of this work is that we must take into account different kinds of elements that can safely be ignored when performing the analysis on pure functions.

In particular, to support more than just pure functions, it is necessary to deal with side-effects and more specifically pointers.

Appendix A

Rewriting rules: equivalence proofs

In the following proofs, 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. For the sake of readability, *true* and *false* values are replaced respectively by 1 and 0. The proofs of all the rules of Figure 5.6 are presented, except for rules (5.10), (5.11) and (5.12) since those are direct consequences of the application of the $\circledast$ operator semantics 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 (5.1). Property: $(e_k \wedge e_l) \circledast w_1 = (e_l \wedge e_k) \circledast w_1$

1. Case $e_k = 0$

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

2. Case $e_l = 0$

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

3. Case $e_k = e_l = 1$

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

□

Proof of rule (5.2). 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 (5.3). 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 (5.4). Property: $e_k \circledast w_1 \oplus e_l \circledast w_1 = w_1 \qquad \text{if } e_l \Leftrightarrow \neg e_k$

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 (5.5). Property: $e_k \circledast w_1 \uplus e_l \circledast w_1 = w_1 \qquad \text{if } e_l \Leftrightarrow \neg e_k$

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 (5.6). Property: $e_k \circledast w_1 \oplus e_l \circledast w_2 = e_k \circledast (w_1 \oplus w_2) \qquad \text{if } e_k \Leftrightarrow e_l$

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 $e_k = e_l = 1$

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

$\square$

Proof of rule (5.7). Property: $e_k \circledast w_1 \uplus e_l \circledast w_2 = e_k \circledast (w_1 \uplus w_2) \qquad \text{if } e_k \Leftrightarrow e_l$

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 $e_k = e_l = 1$

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

$\square$

Proof of rule (5.8). 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 $e_k = e_l = 1$

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

$\square$

Proof of rule (5.9). 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 (5.13). 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 (5.14). 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 $e_k = 1$

$$(1 \circledast w_1)^{it,l} = (w_1)^{it,l}$$
$$1 \circledast (w_1)^{it,l} = (w_1)^{it,l}$$

□

Appendix B

g723_enc_reconstruct WCET instantiation

```
/*
 * WCET evaluation function
 * @param param_0  1 th procedure argument
 * @param param_1  2 th procedure argument
 * @param param_2  3 th procedure argument
 * @return The WCET of the procedure depending on its
 * arguments
 */
int eval(int param_0, int param_1, int param_2) {
    int cst_1_loop_id;
    int cst_1_eta_0;
    int cst_5_loop_id;
    int cst_5_eta_0;
    int cst_7_loop_id;
    int cst_7_eta_0;
    int alt_4_eta_0;
    int cst_10_loop_id;
    int cst_10_eta_0;
    int cst_12_loop_id;
    int cst_12_eta_0;
    int alt_9_eta_0;
    int alt_2_eta_0;
    int seq_0_loop_id;
    int seq_0_eta_0;

    cst_1_loop_id = 0;
    cst_1_eta_0 = 335;
    cst_5_loop_id = 0;
    cst_5_eta_0 = 367;
    cst_7_loop_id = 0;
    cst_7_eta_0 = 366;
    cst_7_eta_0 = (0 == ((1)*(param_0))) ? cst_7_eta_0 : 0;
    if (cst_7_eta_0 > cst_5_eta_0) {
        alt_4_eta_0 = cst_7_eta_0;
    } else {
        alt_4_eta_0 = cst_5_eta_0;
    }
    alt_4_eta_0 = (0 <= ((4)*(param_1)) + ((1)*(param_2))) ?
      alt_4_eta_0 : 0;
    cst_10_loop_id = 0;
    cst_10_eta_0 = 94;
    cst_12_loop_id = 0;
    cst_12_eta_0 = 102;
    cst_12_eta_0 = (0 == ((1)*(param_0))) ? cst_12_eta_0 : 0;
    if (cst_12_eta_0 > cst_10_eta_0) {
        alt_9_eta_0 = cst_12_eta_0;
    } else {
        alt_9_eta_0 = cst_10_eta_0;
    }
    alt_9_eta_0 = (0 <= ((-4)*(param_1)) + ((-1)*(param_2)) +
      ((-1)*(4))) ? alt_9_eta_0 : 0;
    if (alt_9_eta_0 > alt_4_eta_0) {
        alt_2_eta_0 = alt_9_eta_0;
    } else {
        alt_2_eta_0 = alt_4_eta_0;
    }
    seq_0_loop_id = 0;
    seq_0_eta_0 = 0 + cst_1_eta_0 + alt_2_eta_0;
    return seq_0_eta_0;
}
```

List of Figures

List of Examples

Bibliography

[AAN11a] Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. “Precise and efficient parametric path analysis”. In: *SIGPLAN Not.* 46.5 (Apr. 2011), pp. 141–150. ISSN: 0362-1340. DOI: 10.1145/2016603.1967697. URL: http://doi.org/10.1145/2016603.1967697.

[AAN11b] Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. “Symbolic Worst Case Execution Times”. en. In: *Theoretical Aspects of Computing – ICTAC 2011*. Ed. by Antonio Cerone and Pekka Pihlajasaari. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 25–44. ISBN: 978-3-642-23283-1. DOI: 10.1007/978-3-642-23283-1_5.

[ABB19] Kunal Agrawal, Sanjoy Baruah, and Alan Burns. “Semi-Clairvoyance in Mixed-Criticality Scheduling”. In: *2019 IEEE Real-Time Systems Symposium (RTSS)*. Hong Kong, China: IEEE, Dec. 2019, pp. 458–468. DOI: 10.1109/RTSS46320.2019.00047.

[All70] Frances E. Allen. “Control flow analysis”. In: *ACM SIGPLAN Notices* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479. URL: http://doi.org/10.1145/390013.808479.

[Alt+08] Sebastian Altmeyer, Christian Hümbert, Björn Lisper, and Reinhard Wilhelm. “Parametric Timing Analysis for Complex Architectures”. In: *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Kaohsiung, Taiwan: IEEE, Aug. 2008, pp. 367–376. DOI: 10.1109/RTCSA.2008.7.

[Alt+96] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. “Cache behavior prediction by abstract interpretation”. en. In: *Static Analysis*. Ed. by Radhia Cousot and David A. Schmidt. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 52–66. ISBN: 978-3-540-70674-8. DOI: 10.1007/3-540-61739-6_33.

[Alt96] P. Altenbernd. “On the false path problem in hard real-time programs”. In: *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*. June 1996, pp. 102–107. DOI: 10.1109/EMWRTS.1996.557827.

[Arm23] Arm. *Procedure Call Standard for the Arm® Architecture*. en. 2023. URL: https://developer.arm.com/Additional%20Resources/ABI-Procedure%20Call%20Standard%20for%20the%20Arm%20Architecture.

[Bal+10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. “OTAWA: An Open Toolbox for Adaptive WCET Analysis”. en. In: *Software Technologies for Embedded and Ubiquitous Systems*. Ed. by Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 35–46. ISBN: 978-3-642-16256-5. DOI: 10.1007/978-3-642-16256-5_6.

[Bal+19] Clément Ballabriga, Julien Forget, Laure Gonnord, Giuseppe Lipari, and Jordy Ruiz. “Static Analysis of Binary Code with Memory Indirections Using Polyhedra”. en. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Constantin Enea and Ruzica Piskac. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 114–135. ISBN: 978-3-030-11245-5. DOI: 10.1007/978-3-030-11245-5_6.

[Bal+23] Clément Ballabriga, Julien Forget, Sandro Grebant, and Giuseppe Lipari. “New challenges in adaptive real-time systems with parametric WCET”. In: *12th International Real-Time Scheduling Open Problems Seminar*. Vienne, Austria, July 2023. URL: https://hal.science/hal-04197411.

[Bar+06] J. Barre, C. Landet, C. Rochange, and P. Sainrat. “Modeling Instruction-Level Parallelism for WCET Evaluation”. In: *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*. Aug. 2006, pp. 61–67. DOI: 10.1109/RTCSA.2006.44.

[BB06] Adam Betts and Guillem Bernat. “Tree-based WCET analysis on instrumentation point graphs”. In: May 2006, 8 pp. ISBN: 978-0-7695-2561-7. DOI: 10.1109/ISORC.2006.75.

[BDB08] Enrico Bini, Marco Di Natale, and Giorgio Buttazzo. “Sensitivity analysis for fixed-priority real-time systems”. en. In: *Real-Time Syst* 39.1 (Aug. 2008), pp. 5–30. ISSN: 1573-1383. DOI: 10.1007/s11241-006-9010-1. URL: https://doi.org/10.1007/s11241-006-9010-1.

[Bec+19] Martin Becker, Ravindra Metta, R. Venkatesh, and Samarjit Chakraborty. "Scalable and precise estimation and debugging of the worst-case execution time for analysis-friendly processors: a comeback of model checking". en. In: *International Journal on Software Tools for Technology Transfer* 21.5 (Oct. 2019), pp. 515–543. ISSN: 1433-2787. DOI: 10.1007/s10009-018-0497-2. URL: https://doi.org/10.1007/s10009-018-0497-2.

[BEL11] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. "An efficient algorithm for parametric WCET calculation". en. In: *Journal of Systems Architecture.* Design and Optimization for Embedded and Real-Time Computing Systems and Applications 57.6 (June 2011), pp. 614–624. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2010.06.009. URL: https://www.sciencedirect.com/science/article/pii/S1383762110000676.

[BFL17] Clément Ballabriga, Julien Forget, and Giuseppe Lipari. "Symbolic WCET Computation". en. In: *ACM Transactions on Embedded Computing Systems* 17.2 (2017), pp. 1–26. ISSN: 1539-9087, 1558-3465. DOI: 10.1145/3147413. URL: https://dl.acm.org/doi/10.1145/3147413.

[BL08] Stefan Bygde and Björn Lisper. "Towards an Automatic Parametric WCET Analysis". In: *8th International Workshop on Worst-Case Execution Time Analysis (WCET'08).* Ed. by Raimund Kirner. Vol. 8. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008. ISBN: 978-3-939897-10-1. DOI: 10.4230/OASIcs.WCET.2008.1659. URL: http://drops.dagstuhl.de/opus/volltexte/2008/1659.

[BLH14] 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).* Berlin, Germany: IEEE, Apr. 2014, pp. 169–178. DOI: 10.1109/RTAS.2014.6926000.

[CB02] 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.* Vienna, Austria: IEEE, June 2002, pp. 50–59. DOI: 10.1109/EMRTS.2002.1019185.

[CBM] CBMC. *The CBMC Homepage.* URL: https://www.cprover.org/cbmc/.

[CBS00] M. Caccamo, G. Buttazzo, and Lui Sha. "Capacity sharing for overrun control". In: *Proceedings 21st IEEE Real-Time Systems Symposium.* Orlando, FL, USA: IEEE, Nov. 2000, pp. 295–304. DOI: 10.1109/REAL.2000.896018.

[CC12] Patrick Cousot and Radhia Cousot. “An abstract interpretation framework for termination”. In: *ACM SIGPLAN Notices* 47.1 (Jan. 2012), pp. 245–258. ISSN: 0362-1340. DOI: 10.1145/2103621.2103687. URL: http://doi.org/10.1145/2103621.2103687.

[CC77] Patrick Cousot and Radhia 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. New York, NY, USA: Association for Computing Machinery, Jan. 1977, pp. 238–252. ISBN: 978-1-4503-7350-0. DOI: 10.1145/512950.512973. URL: http://doi.org/10.1145/512950.512973.

[Čer+15] Pavol Černý, Thomas A. Henzinger, Laura Kovács, Arjun Radhakrishna, and Jakob Zwirchmayr. “Segment Abstraction for Worst-Case Execution Time Analysis”. en. In: *Programming Languages and Systems*. Ed. by Jan Vitek. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2015, pp. 105–131. ISBN: 978-3-662-46669-8. DOI: 10.1007/978-3-662-46669-8_5.

[Che+07] Ting Chen, Tulika Mitra, Abhik Roychoudhury, and Vivy Suhendra. “Exploiting Branch Constraints without Exhaustive Path Enumeration”. In: *5th International Workshop on Worst-Case Execution Time Analysis (WCET’05)*. Ed. by Reinhard Wilhelm. Vol. 1. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007, pp. 46–49. ISBN: 978-3-939897-24-8. DOI: 10.4230/OASIcs.WCET.2005.816. URL: http://doi.org/10.4230/OASIcs.WCET.2005.816.

[CHR13] Pavol Cerny, Thomas A. Henzinger, and Arjun Radhakrishna. “Quantitative abstraction refinement”. In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’13. New York, NY, USA: Association for Computing Machinery, Jan. 2013, pp. 115–128. ISBN: 978-1-4503-1832-7. DOI: 10.1145/2429069.2429085. URL: https://dl.acm.org/doi/10.1145/2429069.2429085.

[Cof+07] Joel Coffman, Christopher Healy, Frank Mueller, and David Whalley. “Generalizing parametric timing analysis”. In: *SIGPLAN Not.* 42.7 (June 2007), pp. 152–154. ISSN: 0362-1340. DOI: 10.1145/1273444.1254795. URL: http://doi.org/10.1145/1273444.1254795.

[Coh03] Joel S. Cohen. *Computer alegebra and symbolic computation: mathematical methods*. en. Natick, Mass: AK Peters, 2003. ISBN: 978-1-56881-159-8.

[CP00] Antoine Colin and Isabelle Puaut. “Worst Case Execution Time Analysis for a Processor with Branch Prediction”. en. In: *Real-Time Systems* 18.2 (May 2000), pp. 249–274. ISSN: 1573-1383. DOI: 10.1023/A:1008149332687. URL: https://doi.org/10.1023/A:1008149332687.

[CP01] A. Colin and I. Puaut. “A modular and retargetable framework for tree-based WCET analysis”. In: *Proceedings 13th Euromicro Conference on Real-Time Systems*. June 2001, pp. 37–44. DOI: 10.1109/EMRTS.2001.933995.

[Dij59] E. W. Dijkstra. “A note on two problems in connexion with graphs”. en. In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0945-3245. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390.

[DT13] Sun Ding and Hee Beng Kuan Tan. “Detection of Infeasible Paths: Approaches and Challenges”. en. In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by Leszek A. Maciaszek and Joaquim Filipe. Communications in Computer and Information Science. Berlin, Heidelberg: Springer, 2013, pp. 64–78. ISBN: 978-3-642-45422-6. DOI: 10.1007/978-3-642-45422-6_5.

[EE00] J. Engblom and A. Ermedahl. “Modeling complex flows for worst-case execution time analysis”. In: *Proceedings 21st IEEE Real-Time Systems Symposium*. Nov. 2000, pp. 163–174. DOI: 10.1109/REAL.2000.896006.

[EES00] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. “Comparing Different Worst-Case Execution Time Analysis Methods”. en. In: *The Work-in-Progress session of the 21st IEEE Real-Time Systems Symposium (RTSS 2000)*. Orlando, Florida, 2000, p. 4.

[Eng02] Jakob Engblom. “Processor Pipelines and Static Worst-Case Execution Time Analysis”. eng. In: (2002). URL: http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-1832.

[Fal+16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. “TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research”. In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Ed. by Martin Schoeberl. Vol. 55. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 2:1–2:10. ISBN: 978-3-95977-025-5. DOI: 10.4230/OASIcs.WCET.2016.2. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6895.

[Fea88] Paul Feautrier. “Parametric integer programming”. en. In: *RAIRO - Operations Research* 22.3 (1988), pp. 243–268. ISSN: 0399-0559, 1290-3868. DOI: 10.1051/ro/1988220302431. URL: http://www.rairo-ro.org/10.1051/ro/1988220302431.

[FGG18] Joachim Fellmuth, Thomas Göthel, and Sabine Glesner. “Instruction Caches in Static WCET Analysis of Artificially Diversified Software”. In: *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Ed. by Sebastian Altmeyer. Vol. 106. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 21:1–21:23. ISBN: 978-3-95977-075-0. DOI: 10.4230/LIPIcs.ECRTS.2018.21. URL: http://drops.dagstuhl.de/opus/volltexte/2018/8982.

[FW99] Christian Ferdinand and Reinhard Wilhelm. “Efficient and Precise Cache Behavior Prediction for Real-Time Systems”. en. In: *Real-Time Systems* 17.2 (Nov. 1999), pp. 131–181. ISSN: 1573-1383. DOI: 10.1023/A:1008186323068. URL: https://doi.org/10.1023/A:1008186323068.

[GBF21] Sandro Grebant, Clément Ballabriga, and Julien Forget. “Efficient tree-based symbolic WCET computation”. en. In: *Compas’21 :Conférence francophone d’informatique en Parallélisme, Architecture et Système*. Lyon, France, July 2021. URL: https://hal.science/hal-03428961.

[GC11] Laurent George and Pierre Courbin. “Reconfiguration of Uniprocessor Sporadic Real-Time Systems: The Sensitivity Approach”. en. In: *Reconfigurable Embedded Control Systems: Applications for Flexibility and Agility*. Hershey, PA: IGI Global, 2011. DOI: 10.4018/978-1-60960-086-0.ch007. URL: https://www.igi-global.com/chapter/reconfigurable-embedded-control-systems/50429.

[GEL06] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. “Algorithms for Infeasible Path Calculation”. In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET’06)*. Ed. by Frank Mueller. Vol. 4. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006, pp. 1–6. ISBN: 978-3-939897-03-3. DOI: 10.4230/OASIcs.WCET.2006.667. URL: http://doi.org/10.4230/OASIcs.WCET.2006.667.

[Gre+23] Sandro Grebant, Clément Ballabriga, Julien Forget, and Giuseppe Lipari. “WCET analysis with procedure arguments as parameters”. In: *Proceedings of the 31st International Conference on Real-Time Networks and Systems*. RTNS ’23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 11–22. ISBN: 978-1-4503-9983-8. DOI: 10.1145/3575757.3593655. URL: https://doi.org/10.1145/3575757.3593655.

[Gus+06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn 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)*. Rio de Janeiro, Brazil: IEEE, Dec. 2006, pp. 57–66. DOI: 10.1109/RTSS.2006.12.

[Gus+10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. "The Mälardalen WCET Benchmarks: Past, Present And Future". In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Ed. by Björn Lisper. Vol. 15. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 136–146. ISBN: 978-3-939897-21-7. DOI: 10.4230/OASIcs.WCET.2010.136. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2833.

[Gus00] Jan Gustafsson. "Analyzing execution-time of object-oriented programs using abstract interpretation". eng. PhD thesis. Uppsala University, May 2000. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-1174.

[Hea+98] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. "Bounding loop iterations for timing analysis". In: *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No.98TB100245)*. June 1998, pp. 12–21. DOI: 10.1109/RTTAS.1998.683183.

[Hea+99] C.A. Healy, R.D. Arnold, F. Mueller, D.B. Whalley, and M.G. Harmon. "Bounding pipeline and instruction cache performance". In: *IEEE Transactions on Computers* 48.1 (Jan. 1999), pp. 53–70. ISSN: 1557-9956. DOI: 10.1109/12.743411.

[Hen+14] Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maïza. "How to Compute Worst-Case Execution Time by Optimization modulo Theory and a Clever Encoding of Program Semantics". In: *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 43–52. ISBN: 9781450328777. DOI: 10.1145/2597809.2597817. URL: https://doi.org/10.1145/2597809.2597817.

[HJR11] B. K. Huynh, L. Ju, and A. Roychoudhury. "Scope-Aware Data Cache Analysis for WCET Estimation". In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. Apr. 2011, pp. 203–212. DOI: 10.1109/RTAS.2011.27.

[HRW15] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. “Toward Compact Abstractions for Processor Pipelines”. en. In: *Correct System Design: Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015, Proceedings*. Ed. by Roland Meyer, André Platzer, and Heike Wehrheim. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 205–220. ISBN: 978-3-319-23506-6. DOI: 10.1007/978-3-319-23506-6_14. URL: https://doi.org/10.1007/978-3-319-23506-6_14.

[HW02] C.A. Healy and D.B. Whalley. “Automatic detection and exploitation of branch constraints for timing analysis”. In: *IEEE Transactions on Software Engineering* 28.8 (Aug. 2002), pp. 763–781. ISSN: 1939-3520. DOI: 10.1109/TSE.2002.1027799.

[HWH95] C.A. Healy, D.B. Whalley, and M.G. Harmon. “Integrating the timing analysis of pipelining and instruction caching”. en. In: *Proceedings 16th IEEE Real-Time Systems Symposium*. Pisa, Italy: IEEE Comput. Soc. Press, 1995, pp. 288–297. ISBN: 978-0-8186-7337-5. DOI: 10.1109/REAL.1995.495218. URL: http://ieeexplore.ieee.org/document/495218/.

[JC97] Johan Janssen and Henk Corporaal. “Making graphs reducible with controlled node splitting”. In: *ACM Transactions on Programming Languages and Systems* 19.6 (Nov. 1997), pp. 1031–1052. ISSN: 0164-0925. DOI: 10.1145/267959.269971. URL: https://dl.acm.org/doi/10.1145/267959.269971.

[KBC10] Tai Hyo Kim, Ho Jung Bang, and Sung Deok Cha. “A systematic representation of path constraints for implicit path enumeration technique”. en. In: *Software Testing, Verification and Reliability* 20.1 (2010), pp. 39–61. ISSN: 1099-1689. DOI: https://doi.org/10.1002/stvr.406. URL: http://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.406.

[Keb06] D. Kebbal. “Automatic flow analysis using symbolic execution and path enumeration”. In: *2006 International Conference on Parallel Processing Workshops (ICPPW’06)*. Aug. 2006, 8 pp.–404. DOI: 10.1109/ICPPW.2006.26.

[Kir+11] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. “Beyond loop bounds: comparing annotation languages for worst-case execution time analysis”. en. In: *Software & Systems Modeling* 10.3 (July 2011), pp. 411–437. ISSN: 1619-1374. DOI: 10.1007/s10270-010-0161-0. URL: https://doi.org/10.1007/s10270-010-0161-0.

[KKZ12] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. "Symbolic Loop Bound Computation for WCET Analysis". en. In: *Perspectives of Systems Informatics*. Ed. by Edmund Clarke, Irina Virbitskaite, and Andrei Voronkov. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 227–242. ISBN: 978-3-642-29709-0. DOI: 10.1007/978-3-642-29709-0_20.

[LB00] 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*. Stockholm, Sweden: IEEE, June 2000, pp. 193–200. DOI: 10.1109/EMRTS.2000.854007.

[Lim+95] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. "An accurate worst case timing analysis for RISC processors". In: *IEEE Transactions on Software Engineering* 21.7 (July 1995), pp. 593–604. ISSN: 1939-3520. DOI: 10.1109/32.392980.

[Lis03] Björn Lisper. "Fully Automatic, Parametric Worst-Case Execution Time Analysis". In: Jan. 2003, pp. 99–102.

[LM95] Yau-Tsun Steven Li and Sharad 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. New York, NY, USA: Association for Computing Machinery, Nov. 1995, pp. 88–98. ISBN: 978-1-4503-7308-1. DOI: 10.1145/216636.216666. URL: http://doi.org/10.1145/216636.216666.

[LMW95] Y.-T.S. Li, S. Malik, and A. Wolfe. "Efficient microarchitecture modeling and path analysis for real-time software". In: *Proceedings 16th IEEE Real-Time Systems Symposium*. Dec. 1995, pp. 298–307. DOI: 10.1109/REAL.1995.495219.

[LMW96] Y.-T. S. Li, S. Malik, and A. Wolfe. "Cache modeling for real-time software: beyond direct mapped instruction caches". In: *17th IEEE Real-Time Systems Symposium*. Dec. 1996, pp. 254–263. DOI: 10.1109/REAL.1996.563722.

[lps] lpsolve. *lpsolve*. URL: https://sourceforge.net/projects/lpsolve/.

[LRM04] Xianfeng Li, A. Roychoudhury, and T. Mitra. "Modeling out-of-order processors for software timing analysis". In: *25th IEEE International Real-Time Systems Symposium*. Dec. 2004, pp. 92–103. DOI: 10.1109/REAL.2004.33.

[LRM06] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. “Modeling out-of-order processors for WCET analysis”. en. In: *Real-Time Systems* 34.3 (Nov. 2006), pp. 195–227. ISSN: 1573-1383. DOI: 10.1007/s11241-006-9205-5. URL: https://doi.org/10.1007/s11241-006-9205-5.

[LS08] Mark H. Liffiton and Karem A. Sakallah. “Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints”. en. In: *Journal of Automated Reasoning* 40.1 (Jan. 2008), pp. 1–33. ISSN: 1573-0670. DOI: 10.1007/s10817-007-9084-z. URL: https://doi.org/10.1007/s10817-007-9084-z.

[LS99] Thomas Lundqvist and Per Stenström. “An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution”. en. In: *Real-Time Systems* 17.2 (Nov. 1999), pp. 183–207. ISSN: 1573-1383. DOI: 10.1023/A:1008138407139. URL: https://doi.org/10.1023/A:1008138407139.

[LTH02] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. “Pipeline Modeling for Timing Analysis”. en. In: *Static Analysis*. Ed. by Manuel V. Hermenegildo and Germán Puebla. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, pp. 294–309. ISBN: 978-3-540-45789-3. DOI: 10.1007/3-540-45789-5_22.

[McC76] T.J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. ISSN: 1939-3520. DOI: 10.1109/TSE.1976.233837.

[Met+16] Ravindra Metta, Martin Becker, Prasad Bokil, Samarjit Chakraborty, and R Venkatesh. “TIC: a scalable model checking based approach to WCET estimation”. In: *ACM SIGPLAN Notices* 51.5 (June 2016), pp. 72–81. ISSN: 0362-1340. DOI: 10.1145/2980930.2907961. URL: http://doi.org/10.1145/2980930.2907961.

[Min06] Antoine Miné. “The octagon abstract domain”. en. In: *Higher-Order and Symbolic Computation* 19.1 (Mar. 2006), pp. 31–100. ISSN: 1573-0557. DOI: 10.1007/s10990-006-8609-1. URL: https://doi.org/10.1007/s10990-006-8609-1.

[Moh+05] 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)*. Miami, FL, USA: IEEE, Dec. 2005, 10 pp.–242. DOI: 10.1109/RTSS.2005.33.

[Moh+11] Sibin Mohan, Frank Mueller, Michael Root, William Hawkins, Christopher Healy, David Whalley, and Emilio Vivancos. “Parametric timing analysis and its application to dynamic voltage scaling”. In: *ACM Trans. Embed. Comput. Syst.* 10.2 (Jan. 2011), 25:1–25:34. ISSN: 1539-9087. DOI: 10.1145/1880050.1880061. URL: http://doi.org/10.1145/1880050.1880061.

[MS15] Vincent Mussot and Pascal Sotin. “Improving WCET Analysis Precision through Automata Product”. In: *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2015, pp. 207–216. DOI: 10.1109/RTCSA.2015.11.

[Mus+16] Vincent Mussot, Jordy Ruiz, Pascal Sotin, Marianne De Michiel, and Hugues Cassé. “Expressing and exploiting path conflicts in WCET analysis”. In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016) in conjunction with ECRTS*. Vol. 55. Toulouse, France, July 2016, pp. 1–11. URL: https://hal.archives-ouvertes.fr/hal-01682967.

[MW95] F. Mueller and D. B. Whalley. “Fast instruction cache analysis via static cache simulation”. In: *Proceedings of Simulation Symposium*. Apr. 1995, pp. 105–114. DOI: 10.1109/SIMSYM.1995.393589.

[Pal+08] Luigi Palopoli, Luca Abeni, Tommaso Cucinotta, Giuseppe Lipari, and Sanjoy K. Baruah. “Weighted feedback reclaiming for multimedia applications”. In: *2008 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*. Atlanta, GA, USA: IEEE, Oct. 2008, pp. 121–126. DOI: 10.1109/ESTMED.2008.4697009.

[PK89] P. Puschner and Ch. Koza. “Calculating the maximum execution time of real-time programs”. en. In: *Real-Time Systems* 1.2 (Sept. 1989), pp. 159–176. ISSN: 1573-1383. DOI: 10.1007/BF00571421. URL: https://doi.org/10.1007/BF00571421.

[Ray14] P. Raymond. “A general approach for expressing infeasibility in Implicit Path Enumeration Technique”. In: *2014 International Conference on Embedded Software (EMSOFT)*. Oct. 2014, pp. 1–9. DOI: 10.1145/2656045.2656046.

[RC15] Jordy Ruiz and Hugues Cassé. “Using SMT Solving for the Lookup of Infeasible Paths in Binary Programs”. In: *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*. Ed. by Francisco J. Cazorla. Vol. 47. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 95–104. ISBN: 978-3-939897-95-8. DOI: 10.4230/OASIcs.WCET.2015.95. URL: http://doi.org/10.4230/OASIcs.WCET.2015.95.

[RCM17] Jordy Ruiz, Hugues Cassé, and Marianne 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)*. Shanghai, China: IEEE, Sept. 2017, pp. 1–10. DOI: 10.1109/SCAM.2017.13.

[RS09] Christine Rochange and Pascal Sainrat. “A Context-Parameterized Model for Static Analysis of Execution Times”. en. In: *Transactions on High-Performance Embedded Architectures and Compilers II*. Ed. by Per Stenström. Vol. 5470. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 222–241. ISBN: 978-3-642-00904-4. DOI: 10.1007/978-3-642-00904-4_12. URL: http://link.springer.com/10.1007/978-3-642-00904-4_12.

[SF99] Jörn Schneider and Christian Ferdinand. “Pipeline behavior prediction for superscalar processors by abstract interpretation”. In: *ACM SIGPLAN Notices* 34.7 (1999), pp. 35–44. ISSN: 0362-1340. DOI: 10.1145/315253.314432. URL: https://dl.acm.org/doi/10.1145/315253.314432.

[SR10] T. Sondag and H. Rajan. “A More Precise Abstract Domain for Multi-level Caches for Tighter WCET Analysis”. In: *2010 31st IEEE Real-Time Systems Symposium*. Nov. 2010, pp. 395–404. DOI: 10.1109/RTSS.2010.8.

[SS07] Rathijit Sen and Y. N. Srikant. “WCET estimation for executables in the presence of data caches”. In: *Proceedings of the 7th ACM & IEEE international conference on Embedded software*. EMSOFT ’07. New York, NY, USA: Association for Computing Machinery, Sept. 2007, pp. 203–212. ISBN: 978-1-59593-825-1. DOI: 10.1145/1289927.1289960. URL: http://doi.org/10.1145/1289927.1289960.

[Suh+06] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. “Efficient detection and exploitation of infeasible paths for software timing analysis”. In: *Proceedings of the 43rd annual Design Automation Conference*. DAC ’06. New York, NY, USA: Association for Computing Machinery, July 2006, pp. 358–363. ISBN: 978-1-59593-381-2. DOI: 10.1145/1146909.1147002. URL: http://doi.org/10.1145/1146909.1147002.

[Sun+98] Sung-Soo Lim, Jung Hee Han, Jihong Kim, and Sang Lyul Min. “A worst case timing analysis technique for multiple-issue machines”. In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*. Dec. 1998, pp. 334–345. DOI: 10.1109/REAL.1998.739765.

[TFW00] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. "Fast and Precise WCET Prediction by Separated Cache and Path Analyses". en. In: *Real-Time Systems* 18.2 (May 2000), pp. 157–179. ISSN: 1573-1383. DOI: 10.1023/A:1008141130870. URL: https://doi.org/10.1023/A:1008141130870.

[Tou+19] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. "Fast and exact analysis for LRU caches". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 54:1–54:29. DOI: 10.1145/3290367. URL: https://doi.org/10.1145/3290367.

[Ves07] Steve Vestal. "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. Dec. 2007, pp. 239–243. DOI: 10.1109/RTSS.2007.47.

[Viv+01] Emilio Vivancos, Christopher Healy, Frank Mueller, and David Whalley. "Parametric Timing Analysis". In: *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*. OM '01. New York, NY, USA: Association for Computing Machinery, Aug. 2001, pp. 88–93. ISBN: 978-1-58113-426-1. DOI: 10.1145/384198.384230. URL: http://doi.org/10.1145/384198.384230.

[Wil+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. "The worst-case execution-time problem—overview of methods and survey of tools". en. In: *ACM Transactions on Embedded Computing Systems* 7.3 (Apr. 2008), pp. 1–53. ISSN: 1539-9087, 1558-3465. DOI: 10.1145/1347375.1347389. URL: https://dl.acm.org/doi/10.1145/1347375.1347389.

[WSy] WSymb. *WSymb*. URL: https://gitlab.cristal.univ-lille.fr/otawa-plugins/WSymb.

[ZBN93] N. Zhang, A. Burns, and M. Nicholson. "Pipelined processors and worst case execution times". en. In: *Real-Time Systems* 5.4 (Oct. 1993), pp. 319–343. ISSN: 1573-1383. DOI: 10.1007/BF01088834. URL: https://doi.org/10.1007/BF01088834.