PARAMETRIC WCET ANALYSIS

Stefan Bygde

Akademisk avhandling

som för avläggande av filosofie doktorsexamen i datavetenskap vid Akademin för innovation, design och teknik kommer att offentligen försvaras tisdagen den 4 juni 2013, 13.00 i Delta, Mälardalens högskola, Västerås.

Fakultetsopponent: Principal Lecturer Raimund Kirner, University of Hertfordshire
Abstract

In a real-time system, it is crucial to ensure that all tasks of the system hold their deadlines. A missed deadline in a real-time system means that the system has not been able to function correctly. If the system is safety critical, this could potentially lead to disaster. To ensure that all tasks keep their deadlines, the Worst-Case Execution Time (WCET) of these tasks has to be known.

Static analysis analyses a safe model of the hardware together with the source or object code of a program to derive an estimate of the WCET. This estimate is guaranteed to be equal to or greater than the real WCET. This is done by making calculations which in all steps make sure that the time is exactly or conservatively estimated. In many cases, however, the execution time of a task or a program is highly dependent on the given input. Thus, the estimated worst case may correspond to some input or configuration which is rarely (or never) used in practice. For such systems, where execution time is highly input dependent, a more accurate timing analysis which take input into consideration is desired.

In this thesis we present a method based on abstract interpretation and counting of semantic states of a program that gives a WCET in terms of some input to the program. This means that the WCET is expressed as a formula of the input rather than a constant. This means that once the input is known, the actual WCET may be more accurate than the absolute and global WCET. Our research also investigate how this analysis can be safe when arithmetic operations causes integers to wrap-around, where the common assumption in static analysis is that variables can take the value of any integer. Our method has been implemented as a prototype and as a part of a static WCET analysis tool in order to get experience with the method and to evaluate the different aspects. Our method shows that it is possible to obtain very complex and detailed information about the timing of a program, given its input.
Abstract

In a real-time system, it is crucial to ensure that all tasks of the system hold their deadlines. A missed deadline in a real-time system means that the system has not been able to function correctly. If the system is safety critical, this could potentially lead to disaster. To ensure that all tasks keep their deadlines, the Worst-Case Execution Time (WCET) of these tasks has to be known.

Static analysis analyses a safe model of the hardware together with the source or object code of a program to derive an estimate of the WCET. This estimate is guaranteed to be equal to or greater than the real WCET. This is done by making calculations which in all steps make sure that the time is exactly or conservatively estimated. In many cases, however, the execution time of a task or a program is highly dependent on the given input. Thus, the estimated worst case may correspond to some input or configuration which is rarely (or never) used in practice. For such systems, where execution time is highly input dependent, a more accurate timing analysis which take input into consideration is desired.

In this thesis we present a method based on abstract interpretation and counting of semantic states of a program that gives a WCET in terms of some input to the program. This means that the WCET is expressed as a formula of the input rather than a constant. This means that once the input is known, the actual WCET may be more accurate than the absolute and global WCET. Our research also investigate how this analysis can be safe when arithmetic operations causes integers to wrap-around, where the common assumption in static analysis is that variables can take the value of any integer. Our method has been implemented as a prototype and as a part of a static WCET analysis tool in order to get experience with the method and to evaluate the different aspects. Our method shows that it is possible to obtain very complex and detailed information about the timing of a program, given its input.
Abstract

In a real-time system, it is crucial to ensure that all tasks of the system hold their deadlines. A missed deadline in a real-time system means that the system has not been able to function correctly. If the system is safety critical, this could potentially lead to disaster. To ensure that all tasks keep their deadlines, the Worst-Case Execution Time (WCET) of these tasks has to be known.

Static analysis analyses a safe model of the hardware together with the source or object code of a program to derive an estimate of the WCET. This estimate is guaranteed to be equal to or greater than the real WCET. This is done by making calculations which in all steps make sure that the time is exactly or conservatively estimated. In many cases, however, the execution time of a task or a program is highly dependent on the given input. Thus, the estimated worst case may correspond to some input or configuration which is rarely (or never) used in practice. For such systems, where execution time is highly input dependent, a more accurate timing analysis which take input into consideration is desired.

In this thesis we present a method based on abstract interpretation and counting of semantic states of a program that gives a WCET in terms of some input to the program. This means that the WCET is expressed as a formula of the input rather than a constant. This means that once the input is known, the actual WCET may be more accurate than the absolute and global WCET. Our research also investigate how this analysis can be safe when arithmetic operations causes integers to wrap-around, where the common assumption in static analysis is that variables can take the value of any integer. Our method has been implemented as a prototype and as a part of a static WCET analysis tool in order to get experience with the method and to evaluate the different aspects. Our method shows that it is possible to obtain very complex and detailed information about the timing of a program, given its input.
Acknowledgements

This thesis is the result of my research since fall 2006. I like to begin by thanking Björn Lisper, Hans Hansson and Christer Norström for deciding to employ me as a PhD student at Mälardalen University! I am also very grateful to my supervisors Björn Lisper (my main supervisor), Andreas Ermedahl (co-supervisor for the first few years), and Jan Gustafsson (co-supervisor). I also would like to thank Niklas Holsti whom I have been working together with a lot the recent couple of years. Thank you all very much!

I have been meeting a lot of people during these years and have got a lot of new friends and had a lot of good times. I would like to thank all my colleagues, friends and family. I will not attempt to make a list of all people that deserves my gratitude during these years, but if you are looking for your name here it means that you have my thanks! Thank you very much for your support!

My research has been funded by CUGS (the National Graduate School in Computer Science, Sweden), the Swedish Foundation for Strategic Research (SSF) via the strategic research centre P ROGRESS and finally by the Marie Curie IAPP project APARTS.

Stefan Bygde
Västerås, May, 2013
Acknowledgements

This thesis is the result of my research since fall 2006. I like to begin by thanking Björn Lisper, Hans Hansson and Christer Norström for deciding to employ me as a PhD student at Mälardalen University! I am also very grateful to my supervisors Björn Lisper (my main supervisor), Andreas Ermedahl (co-supervisor for the first few years), and Jan Gustafsson (co-supervisor). I also would like to thank Niklas Holsti whom I have been working together with a lot the recent couple of years. Thank you all very much!

I have been meeting a lot of people during these years and have got a lot of new friends and had a lot of good times. I would like to thank all my colleagues, friends and family. I will not attempt to make a list of all people that deserves my gratitude during these years, but if you are looking for your name here it means that you have my thanks! Thank you very much for your support!

My research has been funded by CUGS (the National Graduate School in Computer Science, Sweden), the Swedish Foundation for Strategic Research (SSF) via the strategic research centre PROGRESS and finally by the Marie Curie IAPP project APARTS.

Stefan Bygde
Västerås, May, 2013
# Contents

## 1 Introduction
1.1 Embedded Systems ........................................ 1  
1.2 Real-Time Systems ........................................ 2  
1.3 Worst-Case Execution Time ................................. 2  
  1.3.1 Determining the WCET of a Program ................. 3  
1.4 WCET Analysis .............................................. 4  
  1.4.1 Dynamic Analysis ..................................... 4  
  1.4.2 Static Analysis ....................................... 5  
  1.4.3 Relation between approaches and the WCET ........ 5  
1.5 Static WCET Analysis ...................................... 6  
  1.5.1 Flow Analysis ....................................... 7  
  1.5.2 Low-Level Analysis .................................. 8  
  1.5.3 Calculation ......................................... 9  
1.6 Parametric WCET Analysis ................................. 10  
  1.6.1 Benefits of Parametric WCET Analysis .............. 11  
  1.6.2 Related Work ....................................... 12  
1.7 Summary .................................................... 14

## 2 Research
2.1 Introduction and Background ................................ 15  
2.2 Research Method .......................................... 16  
2.3 Problem Formulation ...................................... 16  
2.4 Overview of Solutions .................................... 17  
  2.4.1 Approach to Research Questions ..................... 17  
  2.4.2 Contributions ....................................... 18  
  2.4.3 Summary of Publications ............................. 19  
2.5 Thesis Outline ............................................. 20
## 3 Technical Introduction

3.1 Notation and Conventions .................................. 23  
3.1.1 Functions .................................. 23  
3.2 Program Model .................................. 24  
3.2.1 Syntax .................................. 25  
3.2.2 Semantics .................................. 27  
3.2.3 Example .................................. 30  
3.2.4 Program Timing ................................. 31  
3.3 Traces .................................. 31  
3.3.1 Trace Semantics .................................. 33  
3.3.2 Using Trace Semantics to Compute the Execution Time ............. 33  
3.4 Parametric WCET Analysis .................................. 34  
3.4.1 Making it a Counting Problem ......................... 35  
3.5 Collecting Semantics .................................. 39  
3.6 Control Variables .................................. 40  
3.7 Input Parameters .................................. 44  
3.7.1 Representing Programs ............................. 48  
3.8 Summary .................................. 50

## 4 Abstract Interpretation and the Polyhedral Domain

4.1 Domain Theory .................................. 51  
4.1.1 Lattices .................................. 52  
4.1.2 Fixed Point Theory .................................. 53  
4.2 Introduction to Abstract Interpretation .................................. 54  
4.2.1 Abstraction .................................. 54  
4.2.2 Abstract Functions .................................. 56  
4.2.3 Widening and Narrowing .................................. 58  
4.3 The Polyhedral Domain .................................. 59  
4.3.1 Convex Polyhedra .................................. 60  
4.3.2 Representation .................................. 60  
4.3.3 The Lattice of Convex Polyhedra .................................. 61  
4.3.4 Convex Polyhedra as Abstract Domain .................................. 63  
4.4 Approximating Program Semantics .................................. 64  
4.4.1 The Polyhedral Domain as Abstract Environment .................................. 65  
4.4.2 The Abstract Semantic Function .................................. 66  
4.5 Abstract Interpretation Example .................................. 68
5 Parametric WCET Analysis
  5.1 Introduction ........................................... 71
  5.2 Overview ............................................... 71
  5.3 The State Counting Phase ................................. 72
    5.3.1 Preparing the Program .............................. 73
    5.3.2 Relational Abstract Interpretation .................. 74
    5.3.3 State Counting .................................... 74
    5.3.4 Example of the State Counting Phase ................. 77
  5.4 The Parametric Calculation Phase ......................... 79
    5.4.1 Parametric Integer Programming ..................... 80
  5.5 Reducing the Number of Variables ......................... 82
    5.5.1 Concrete Example of Variable Reduction .............. 84
    5.5.2 The Minimum Propagation Algorithm .................. 86
  5.6 The Combination Phase ................................ 87
  5.7 Simplification ....................................... 87
  5.8 Summary ........................................... 89

6 The Minimum Propagation Algorithm .......................... 91
  6.1 Introduction ........................................... 91
  6.2 The Minimum Propagation Algorithm ....................... 91
    6.2.1 The Min-Tree ...................................... 93
    6.2.2 The Algorithm ..................................... 93
    6.2.3 Example of MPA .................................... 96
  6.3 Properties of MPA .................................... 99
    6.3.1 Termination ....................................... 99
    6.3.2 Complexity ....................................... 101
    6.3.3 Correctness of MPA ................................. 101
    6.3.4 Upper Bounds on Tree Depth ......................... 103

7 Fully Bounded Polyhedra .................................. 105
  7.1 Analysis on Low-level Code ............................... 105
  7.2 Representing Integers .................................. 107
  7.3 Introduction and Related Work ......................... 107
  7.4 Simon and King’s Method ................................ 108
    7.4.1 Implicit wrapping .................................. 109
    7.4.2 Explicit Wrapping .................................. 109
  7.5 Fully Bounded Polyhedra ................................. 110
  7.6 Motivation and Illustrating Example ..................... 110
    7.6.1 Unboundedness in Polyhedral Analysis ............... 114
7.6.2 Making Polyhedra Bounded ........................................ 115
7.6.3 Making Widening Bounded ...................................... 116

8 Implementation ......................................................... 119
  8.1 Introduction .......................................................... 119
  8.2 First Prototype ....................................................... 119
    8.2.1 Input Language .................................................. 119
    8.2.2 Implemented Analyses ......................................... 120
    8.2.3 Discoveries and Experience ................................... 121
  8.3 SWEET ................................................................. 122
    8.3.1 Implemented Analyses ......................................... 122
    8.3.2 Discoveries and Experiences ................................... 123

9 Evaluation ............................................................... 125
  9.1 Introduction .......................................................... 125
    9.1.1 Benchmarks ...................................................... 125
    9.1.2 A Note on Input Parameters .................................. 128
    9.1.3 Experimental set-up ............................................ 128
  9.2 Evaluating Parametric Calculation .............................. 128
    9.2.1 Evaluation of Precision ....................................... 130
    9.2.2 Evaluation of Upper Bounds on Min-Tree Depth .......... 130
    9.2.3 Scaling Properties .............................................. 133
  9.3 The Reason for Over-Estimation .................................. 135
  9.4 Evaluation of Bounded Polyhedra .................................. 137
    9.4.1 Comparison between SK and BD .............................. 138
    9.4.2 Computing the Set of Integers of a Polyhedron .......... 138
    9.4.3 The Results ..................................................... 139
    9.4.4 On Efficiency ................................................... 141
  9.5 Demonstration of Parametric WCET Analysis .................... 142

10 Conclusions ........................................................... 145
  10.1 Introduction .......................................................... 145
  10.2 Research Process ..................................................... 145
  10.3 Main Results .......................................................... 146
    10.3.1 Parametric WCET Analysis .................................... 146
    10.3.2 Parametric Calculation ........................................ 147
    10.3.3 Analysis of Fixed Size Integers ............................ 148
  10.4 Future Work .......................................................... 149
    10.4.1 Evaluation ...................................................... 149

Bibliography ............................................................. 151
Chapter 1

Introduction

In this thesis, we present research with the aim of obtaining safe, input-sensitive estimations of the worst-case execution time of real-time tasks in embedded systems. This introductory chapter explains what this means and puts it into context.

1.1 Embedded Systems

Computer systems which are designed for a specific purpose and which are not operated using a mouse and keyboard like a regular PC are usually referred to as embedded systems. Embedded systems are in fact the most common type of computer systems we use today since they are integrated into mobile phones, cars, trains, aeroplanes, toys, industrial robots etc. Since these systems are specialised in performing specific tasks, and are part of the internal electronics of devices or vehicles, they typically have different requirements than PCs. Often they are small, battery driven and may have strict requirements on safety, performance, energy consumption or timing. In general, embedded systems are more resource constrained than PCs. Most PCs today run on 32-bit or 64-bit processors, while a large portion of embedded systems still uses 8-bit or 16-bit processors.
Chapter 1

Introduction

In this thesis, we present research with the aim of obtaining safe, input-sensitive estimations of the worst-case execution time of real-time tasks in embedded systems. This introductory chapter explains what this means and puts it into context.

1.1 Embedded Systems

Computer systems which are designed for a specific purpose and which are not operated using a mouse and keyboard like a regular PC are usually referred to as embedded systems. Embedded systems are in fact the most common type of computer systems we use today since they are integrated into mobile phones, cars, trains, aeroplanes, toys, industrial robots etc. Since these systems are specialised in performing specific tasks, and are part of the internal electronics of devices or vehicles, they typically have different requirements than PCs. Often they are small, battery driven and may have strict requirements on safety, performance, energy consumption or timing. In general, embedded systems are more resource constrained than PCs. Most PCs today run on 32-bit or 64-bit processors, while a large portion of embedded systems still uses 8-bit or 16-bit processors.
1.2 Real-Time Systems

A very common characteristic of embedded systems is that they need to be predictable in their timing. This is especially the case in safety critical systems. When the timing is so important in a system that it is not considered to operate correctly if it does not perform its computation within a given time frame, we refer to it as a real-time system or RTS. An RTS has requirements to deliver results within a given time frame as part of its functional specification. Examples of real-time systems are control systems in cars or aeroplanes, in which the timing is essential and in which case a timing failure actually could result in a catastrophe. This thesis focuses on important problems and challenges concerning real-time embedded systems.

Since timing is such a central role in an RTS, a real-time operating system must make sure that all timing constraints are fulfilled. An RTS manages a set of software tasks. A task is a piece of software designed to perform a particular action, or sequence of actions, at a particular time or times. Tasks are commonly executed periodically on the system (i.e., with a constant inter-arrival time). Some tasks, however, are executed aperiodically, often as a result of an external event. To make sure that all tasks, both periodic and aperiodic, respect their timing constraints they have to be scheduled.

To schedule a set of tasks, each task is associated with a set of properties and requirements, such as inter-arrival time (if periodic), execution time, priority and deadline. The real-time system is equipped with a scheduler that attempts to make sure that every task can execute, and to make sure it finishes its execution before its deadline (which comes from system requirements) while minimising the consumption of system resources. Some schedulers allow tasks to be interrupted. In such schedulers a task may be suspended to let another task with higher priority execute for a while, as long as the original deadline is still respected. More information about real-time systems and scheduling can be found in, for example [20].

1.3 Worst-Case Execution Time

To make a feasible schedule and make sure all deadlines are met, it is important for the scheduler to know the execution time of a task. The execution time is defined as the time passed between a task starts its execution until it finishes, assuming it is uninterrupted.

The execution time of a task running on a fixed platform will typically vary.
The execution time is affected by the hardware state as well as the input to the task. To make sure that enough time is allocated for a task to execute to its completion, it is essential to know the longest possible execution time a task can have, known as the Worst-Case Execution Time (WCET).

While the terms execution time and WCET are mostly used in the context of tasks in real-time systems, the concepts are applicable to any code segment or program. In the rest of the thesis, we will refer to the WCET and execution times of programs in general, rather than using the specific term task.

1.3.1 Determining the WCET of a Program

Determining the WCET of a program is far from trivial. The worst-case execution time in practice corresponds to a single execution taking the longest time due to the input to the program and the conditions of the hardware. These exact conditions are hard to determine due to the complex interaction of software and hardware. Even small embedded systems may be equipped with advanced and unpredictable (timing wise) hardware features such as caches, pipelines or branch predictors. These features affect the execution times depending on their internal states and interacting with each other in a very complex manner. However, the software has even more impact on the execution time. A non-trivial program typically has a very large number of possible execution paths. The number of paths comes from loops and branches in a program. Finding the single path that gives the longest possible execution time is not easy. Moreover, which path is executed depends, often in non-trivial ways, on the input given to the program. In summary, finding the execution that takes the longest to execute on a particular platform is not practically possible in the general case, let alone finding the execution time of it (as a consequence of Turing’s Halting problem).

The practical approach to determine the execution time of a program is to estimate the WCET as close as possible. To be absolutely sure that the deadline cannot be missed, this estimate has to be safe, meaning that it has to be at least as large as the actual WCET. However, if the allocated execution time of a program is much longer than the actual WCET, it means that the scheduler is making an unnecessarily inefficient schedule. In the worst case, a too long execution time estimate may cause the program set not to be schedulable at all! This means that the estimation should be as precise as possible while remaining safe. This is very challenging, and as we will see, there are several approaches to make such an estimate.
1.4 WCET Analysis

WCET analysis is a systematic process dedicated to obtain an estimate of the WCET of a program. Under certain conditions, such an estimate can be guaranteed to be safe. Figure 1.1 outlines the relationship between execution times and safe estimates. The figure also displays the sometimes important concept of Best-Case Execution Time (BCET), which is defined exactly as WCET but refers to the shortest possible execution time of a program.

Thus, any number greater than or equal to WCET is a safe estimate, but to avoid impossible or impractical schedules this estimate should naturally be as small as possible while being safe. Quite a lot of research has been dedicated to the field of WCET analysis. A good introduction can be found in [119]. There are several approaches to obtaining WCET estimates, but they can in general be classified as dynamic, static or hybrid approaches.

1.4.1 Dynamic Analysis

Dynamic (or measurement based) approaches, which have long been the industrial standard, are methods that rely on running or simulating a program to find information about the execution time. In order to find the WCET then, the program would have to be run on its worst-case input and its worst-case hardware state. However, it is typically very difficult to know which is the worst-case input and hardware state. Thus, the dynamic approach has to execute a program end-to-end a large number of times with a large number of inputs and observe
the longest execution time. However, unless it is a very simple program running on very simple hardware, the measured executions will constitute a subset of the actual set of possible executions. Thus, there are no guarantees that the worst-case has been observed. This means that results from measurement based approaches typically cannot be considered as safe estimates. To compensate for this, sometimes a safety margin is added to increase the likelihood of safety [103]. However, it should be noted that the longest observed execution time constitutes a lower bound for the WCET, which may be a useful indicator of the accuracy of a safe estimate obtained by another method.

Methods for measuring the execution time are outlined in [108]. The simplest way is to simply measure end-to-end execution time using something as simple as a stop-watch (for very coarse approximations) or the time command in Unix. More sophisticated approaches may include the use of profiling or dedicated software analysers.

1.4.2 Static Analysis

Static analysis methods attempt to obtain a WCET estimate without executing or simulating a program. Instead, an estimation is made by statically analysing the software together with the hardware. Static analysis most often aims to derive safe estimates, in contrast to dynamic analyses. However, due to the complexity of software and hardware, and due to the requirement on safety, a static safe static analysis occasionally has to introduce safe approximations resulting in a less precise estimate. Often such approximations arise from simplifications, with the result that the analysis considers potential executions which actually never occur in the program.

1.4.3 Relation between approaches and the WCET

As can be seen, the dynamic and static approaches to WCET analysis do not deliver the same kind of result. Although both aim to derive precise estimates of the WCET, they provide lower and upper bounds of the actual WCET. Figure 1.2 illustrates this concept in a Venn diagram. While this is a bit simplified, it illustrates the basic relationship between the approaches.

There are also hybrid approaches combining static and dynamic analysis. Typically this is done by extracting safe flow information from static WCET analysis and combine it with measured times from traces in some way. Such approaches can generally not guarantee that the result is a lower or upper bound on the WCET, but may provide accurate estimates. Kirner et. al. [69] does this
Chapter 1. Introduction

Figure 1.2: Relation between WCET Analysis approaches. LB and UB stands for lower bound and upper bound respectively.

by automatically deriving appropriate test-data for which to make measurements on. A probabilistic approach is used in [16] which attempts to make a very accurate bound on the WCET with a high probabilistic guarantee of safety.

1.5 Static WCET Analysis

Static WCET Analysis is an emerging technique in industry and there are some commercial static analysis tools on the market such as aiT\(^1\) and Bound-T\(^2\). In addition to commercial WCET tools there are also WCET tools developed in academia, such as SWEET\(^3\), TuBound\(^4\) and OTAWA\(^5\).

The main benefits of using static analysis are:

- It can provide a safe upper bound of the WCET.
- Does not require measuring devices or a controlled environment.
- Typically shorter analysis times compared to dynamic analysis.

---

\(^1\)http://www.absint.com/ait/
\(^2\)http://www.bound-t.com/
\(^3\)http://www.mrtc.mdh.se/projects/wcet/sweet/index.html
\(^4\)http://costa.tuwien.ac.at/tubound.html
\(^5\)http://www.otawa.fr/
- Typically simpler set-up and easier configuration than dynamic analysis.
- Can be made partly or fully automated.

In addition, static analysis has a lot of interesting research challenges, such as making the analysis as accurate, fast and automatic as possible. For these reasons the research presented in this thesis is focused entirely on static analysis.

1.5 Static WCET Analysis

Most static analyses are using the same underlying principles, and are essentially divided into three independent phases. To put it simply, the flow-analysis phase analyses the software, the low-level analysis phase analyses the hardware and the calculation phase combines the analysis results to calculate an estimation of the WCET. This estimation is in most cases expressed as clock cycles, milliseconds or microseconds. Figure 1.3 shows how the different analysis phases relate.

Our research is concerned mainly with the flow analysis and calculation phases. A low-level analysis is important, but in this thesis, the assumption is that a low-level analysis is available and that it can give timing estimates of atomic parts of a program.

1.5.1 Flow Analysis

Flow analysis (or high-level analysis) analyses the source or object code of a program. The goal of this process is to find constraints on the program that originates from the structure and semantics of the program. Below are given some examples of what type of constraints that a flow-analysis can derive.

Structural Constraints. The structure of a program imposes restrictions on the order and frequency of the execution of different parts of a program. By analysing the graph structure of the program, such constraints can be
extracted. When analysing low-level code, this graph structure has to be reconstructed by analysis.

**Value Constraints.** Value analysis can be used to find information about the set of values that variables can attain during execution. This valuable information can be used to find loop bounds, to find out memory addresses or pointer sets. Perhaps the most popular way of doing value analysis is to use abstract interpretation [34].

**Infeasible Paths.** The graph structure of a program often has many possible paths through it. In many cases, however, it is not possible for a program to execute all different paths through the graph. A simple example is an if-statement that is followed by an if-statement with the opposite condition making it impossible to take both if-branches. Detecting such restrictions on the program flow can help to make better estimations of the WCET. Infeasible path detection has been of great interest in literature; Alternbernd [5] explores it by path enumeration, path pruning and symbolic evaluation, in [51] by using abstract execution. Other relevant literature include [109, 71, 61, 4].

**Loop Bounds.** In order to get a finite upper bound on the WCET, an upper bound on every loop of the analysed program has to be known. Because of this a loop bound analysis is crucial for any WCET analysis, and if a loop bound cannot automatically be derived, it has to be provided by the user. Some approaches on finding loop bounds are identifying loop counters [58, 81, 64] or by abstract interpretation to discover the iteration-space and count the number of points in it [78, 42].

### 1.5.2 Low-Level Analysis

The low-level analysis analyses a mathematical model of the hardware platform. The model should be as detailed as possible, but it has to be conservative, that is, it has to over-estimate rather than under-estimate the possible execution times and hardware states. The purpose of the low-level analysis is to derive worst-case execution times for atomic parts of the program. The atomic parts are usually the instructions or basic blocks of a program.

For a low-level analysis to be accurate, it needs to be taking the timing effects of the advanced features of the hardware into consideration. This may include cache memories, pipelines, branch-predictors etc. This makes low-
level analysis quite difficult and ultimately the worst-case timing might depend on the entire execution history.

**Cache Analysis.** An accurate low-level analysis has to account for the effect that a cache has on the WCET. An accurate model should be able to conservatively predict cache misses. Some approaches to cache analysis are found in [44, 40].

**Pipeline Analysis.** Most modern processors use a pipeline mechanism to make it possible for instructions to execute in parallel. However, due to pipeline hazards it cannot be assumed that a pipeline is fully utilised at all times. A good low-level analysis must be able to analyse the worst-case effect of the timing that the pipeline can have. Pipeline analysis has been investigated in for example [39, 102, 40].

**Branch Predictors.** Branch predictors are used to fetch and execute code before a branch is taken. This is so that the pipeline can be utilised even before a conditional branch has been evaluated. A branch predictor may speed up the execution time in average, but a miss-prediction means that the pipeline has to be flushed and to fetch the correct instructions. A good low-level analysis should be able to estimate the effect of this. Approaches to branch predictor analysis are suggested in [32, 83, 84, 39].

Ideally, a low-level analysis should account for several of these effects at the same time, examples of integrated approaches to low-level analysis can be found in [40, 60, 59, 107].

### 1.5.3 Calculation

When flow facts have been derived from the flow analysis and atomic worst-case execution times have been calculated by the low-level analysis, the results can be combined to obtain a concrete bound of the WCET. This is done in the calculation phase. There are three main approaches to calculation, described below.

**Path Based.** Path based calculation involves explicitly modeling each path through a loop in order to determine the most costly one. Since each path is analysed it makes it easy to take hardware dependencies between instructions into consideration. Path based calculation is used in [59, 107, 98, 40].
**Tree Based.** This type of calculation uses the syntax-tree of a program as model for the timing. A timing is determined for each node of the tree bottom-up and different rules depending on what the structure that the node represent are used to calculate the timing of the parent node. The first tree-based approach is known as a “timing schema” [90]. Newer approaches to tree-based calculation include [75, 93, 31].

**Implicit Path Enumeration.** The Implicit Path Enumeration Technique (IPET) was introduced in [73, 74] and a similar method in [97]. The idea here is to represent all path implicitly in contrast to path based calculation that consider paths implicitly. This is done by modeling the execution times for each atomic part of the program and model it as an optimisation problem, in particular a linear integer programming problem that is optimised subject to the flow-constraints. This problem is the solved by a linear integer programming solver. This approach has been used in [40, 97, 89]. A parametric IPET method is investigated in Section 5.4.

### 1.6 Parametric WCET Analysis

It is very common for a program’s execution time to vary according to the given input, often in a complex and unpredictable way. The main reason for this is that the program executes different paths depending on the given input. The worst-case path of a program may correspond to an input that is unlikely or never given to the program, and may not be useful in practice. Thus, if the input would be known, then the worst-case execution time for that particular input would be safe and more precise than the global WCET.

We define a *Parametric WCET Analysis* to be a WCET analysis which results in a formula rather than a constant. This formula has some selected input as parameters and the formula can be instantiated with concrete values once the input is known. The usefulness of this concept is illustrated in Figure 1.4. The X-axis represents different input combinations that result in different execution times of a certain program. The red line represents the WCET of the program. The orange line represents a safe upper bound of the WCET which has been obtained by static analysis. A parametric WCET analysis will result in a formula in terms of input. The blue line represents the parametric WCET estimate. As can be seen, the parametric formula gives a safe upper bound of the execution times based on the input. This means that if the input is known, a much smaller (i.e., better!) estimation can be used. As can be seen, the
blue line can even dip below the actual WCET of the program. This is safe, however, if the input is known.

![Graph showing WCET, Estimated WCET, and Parametric Estimate](image)

Figure 1.4: Relation between the WCET, a fixed WCET estimation and a parametric WCET estimation

### 1.6.1 Benefits of Parametric WCET Analysis

Static WCET analysis is a complex and often time consuming process. If a program would be analysed for every input combination to give a more precise estimate, this would be a very long and tedious process. The benefit of having a formula is that instantiating is much faster than re-analysing a program. A parametric analysis only needs to be done once, and then the formula can be instantiated once the input is known. If the formula is very quick to instantiate, the results could potentially be used in an on-line scheduler where the input is known in advance.

Another benefit of parametric WCET analysis is that it relaxes the requirement of statically determining concrete upper bounds for each loop in a program. A concrete WCET estimate is a finite number which depends on upper loop bounds, however, a parametric formula does not need to have a maximum value. If a loop bound is dependent only on parameters, the loop will be bound once the formula has been instantiated. However, if a loop bound is dependent on non-parameters only, the loop bound still has to be statically known.
Having a mathematical formula as estimate can also be beneficial for several reasons: formulae can be algebraically manipulated, mathematically analysed to find minima and maxima (under certain conditions), evaluated partially, finding timing sensitivity in input \textit{etc.}

This thesis investigates the concept of parametric WCET analysis and the research aims to make advances in this particular field. Parametric WCET analysis is naturally more complex than classical static WCET analysis and probably cannot be used on large systems with millions of lines of code; rather, the parametric estimation is most efficiently used on code segments, like small programs or functions, which have input-dependent execution times. Interesting applications would include \textit{disable interrupt sections}, which are code sections which may not be interrupted and are therefore naturally interesting objects for WCET analysis. These sections typically need to be small and are interesting candidates for a parametric WCET [28]. Another important application of parametric WCET analysis would be in component based software development [36, 62]. In component based software development, reusable components designed to interact with each other in different contexts can be analysed in isolation. Since components are designed to function in different contexts (which would imply different input), a context-dependent WCET estimate is desired. Component models designed for embedded systems (such as saveCCM [57] or Rubus \textsuperscript{6}) typically use quite small components which makes parametric WCET analysis interesting.

1.6.2 Related Work

This section presents previous work that is related to parametric WCET analysis in general. One approach to parametric WCET analysis is to analyse programs parametrised in loop bounds rather than input-values. Vivancos et. al. presents such an analysis in [116], later extended by Coffman et. al [30] for computing polynomial expressions for nested loops. While the method in [116] aims for an integrated cache-analysis which works on low-level code rather than source level code, the flow analysis only takes structural constraints and loop-bound constraints into consideration.

Previous parametric methods has mainly been using path-based or tree-based calculation methods which typically requires that the analysed program has structured program flow [31, 65]. Colin and Bernat [31] uses a tree-based technique with the additional possibility to express constraints such as infeasible paths.

\textsuperscript{6}http://www.arcticus-systems.com/
Some of these approaches [31, 15] relies on using external software like Matlab or Mathematica to simplify and evaluate the parametric formulae. While this is an interesting approach, this is not something that we have explored, but would potentially be interesting.

Many approaches require or recommends user annotations [15, 31] to fully utilise the analysis, while we are aiming to do a fully automatic analysis, ideally without interaction from the user. Some methods [65, 116] are aiming to obtain as simple formulae as possible in order to be able to evaluate them quickly at run-time. This means that these methods intentionally aims to derive a simple (and therefore potentially less precise) formula than possible.

Marref [80] introduces a hybrid approach to parametric analysis based on genetic algorithms. Since part of the method is based on measurements, it cannot be guarantee safety.

In [112], van Engelen et. al. analyses the WCET of loops parametrically. It does so by counting points symbolically over the iteration-space of (possibly nested loops) where each iteration of any loop is allowed to have different worst-case timing. However, it assumes that loops are on the form for $i = a$ to $b$ step $s$, although some of these symbols may be parametric, it still requires the iteration space has a known form.

Altmeyer et. al. [6] presents a method which is similar to ours: it gives a parametric result in terms of input to the program, uses abstract interpretation to determine values and uses parametric integer programming as calculation method. This method is integrated with cache analysis in order to give precise results for complex architectures. However, this method uses a simpler value analysis that does not take complex linear dependencies between variables into account.

Other interesting approaches of representing properties of programs parametrically include [3] where the resource cost of executing a program is modelled as a recurrence relation and evaluated into closed-form. Gulwani investigates how to derive symbolic complexity bounds of a program in [47].

Common for the previous approaches is that they uses mostly constraints coming from program structure, infeasible path information, user annotations, loop bounds (either automatically derived or as information from the user) and except in [65, 6] no automatic value analysis is used to obtain a parametric WCET. Huber et. al. [65] uses SWEET to extract flow facts which uses a non-relational value analysis to derive the constraints.

What differs our method from the previously mentioned methods is that our method do not require structured flow, it does not need to identify loops or other structures explicitly, it parametrises in input-values and is fully auto-
matic and does not require user annotations. None of the previous methods uses a relational value analysis to derive complex constraints on program flow originating from intricate relations between the values of variables.

1.7 Summary

The research in this thesis is in the field of real-time embedded systems. More specifically the research aims to find precise, yet safe, estimates of the worst-case execution time of real-time programs, in order to facilitate scheduling. In particular, these estimates are parametric and are based on automatic static analysis. The research is concerned with the flow analysis of source code and calculation phase.
Chapter 2

Research

2.1 Introduction and Background

The previous section described the context of the research presented in this thesis. The primary problem we are concerned with is to find good and useful estimates for the WCET of tasks executing in real-time embedded systems. The focus on parametric WCET analysis comes from the observation of parametric timing behaviour on industrial code [27, 101, 28] which demonstrated the usefulness of a parametric estimate. Some of the previous approaches to parametric WCET analysis have required manual annotations [31] or have been parametric in loop-bounds rather than input [116, 30].

Lisper introduces an advanced method for parametric WCET analysis in [76], further detailed in [77]. This method is based on well known analysis techniques and is able to express potentially very complex relationships between variables and input parameters. The analysis is more expressive than previously suggested methods, and has the potential to derive more information from programs. Other approaches to WCET analysis (both parametric and non-parametric) such as [6, 42, 78] have been inspired by this method.

For the above reasons, Lisper’s method serves as basis for our research. Note however that our research has not investigated all of the aspects originally presented in [77] (more specifically the parts concerning infeasible path detection).
2.2 Research Method

Our research is mainly motivated by industrial problems and would therefore be classified mainly as applied research. However, many of our solutions are general and contribute to the field of static analysis and computer science in general.

Our basic approach has been to repeat the following steps for several cycles:

1. Literature study about current research problems.
2. Formulate a solution based on academic body of knowledge, deductive methods and discussions.
3. Testing by construction and implementation and discover new research problems.
4. Verification and Validation by experiment.

This process has been iterative since new problems and challenges were discovered in phase 3 and 4. Overall, several techniques have been implemented and tested. While the individual techniques have been developed to solve different problems, the techniques work well together as tools for our research problems.

2.3 Problem Formulation

As described in our research method, our research has been conducted in a series of iterations of progressive research problems. The initial research problem can be stated as:

**Q1** Can Lisper’s method [76, 77] be used to efficiently find accurate parametric WCET estimates on program segments? What are the practical issues with the method, and does it need to be adjusted?

After a prototype implementation and initial results, the calculation phase (see Section 1.5) of the method turned out to be a huge bottleneck, leading to the follow-up question.

**Q2** Is it possible to use Lisper’s method more efficiently by replacing the parametric calculation with something more practical?
This has lead to the development of a new parametric WCET algorithm which is more efficient, at the cost of some lost precision and flexibility.

Our final research question is concerned with the standard flow analysis method used in [76], which is not perfectly safe in presence of integers with wrap-around behaviour. Wrap-around semantics is the most common strategy to deal with overflows, and this is especially prevalent in embedded systems using 8-bit or 16-bit processors.

Q3 Can the flow analysis used in [76] be adapted or changed so that it is safe to use with integers with wrap-around behaviour without losing precision?

2.4 Overview of Solutions

In this section we briefly look at how the different research questions have been approached and our suggested solutions. The solutions have led to a number of scientific publications and a set of concrete contributions in our field.

2.4.1 Approach to Research Questions

Q1

To answer Q1, a simple prototype as proof of concept for Lisper’s method was implemented. The prototype was implemented in Haskell and C++, and analyses programs written in a small subset of C. The method was tested on a set of small programs. From this prototype, the following conclusions could be drawn:

- Lisper’s method can be used to obtain very detailed parametric formulae of small programs.
- The resulting formulae can be very long and complex, even for small input programs.
- The initial approach to parametric calculation is very inefficient and does not scale well.

Later, after considering Q2 and Q3, this question was re-issued leading to an implementation of Lisper’s method into the static WCET analysis tool SWEET. This has resulted in more experience of the method as well as opening up possibilities for further research.
Q2

The approach to the second question was to first study literature, searching for a possible solution. After surveying, a new algorithm was designed to deal with the particular issues that made the previous approach non-scalable. This algorithm was then implemented and tested on a larger set of programs. This led to the following conclusions:

- It is possible to use a quite scalable method for parametric WCET calculation in Lisper’s method.
- The new algorithm is more efficient and more scalable than the previous approach, at the cost of some precision and flexibility.

Q3

The final question was approached by an extensive literature study. Through this study, a previous solution was found [106]. However, the solution had some flaws with imprecision which led to the construction of an extension of this method with the goal of reclaiming some of the lost precision. The previously suggested method and our own improved method were both implemented into SWEET and compared to each other. After the literature study, design, implementation and comparison, we can make the following statements.

- It is possible to use the polyhedral flow analysis suggested in [76] without too much changes and still makes it safe for integers with wrap-around semantics.
- The previous approach had problems with imprecision.
- Initial experiments with our new extended method indicate that it is strictly better than the previous approach.

2.4.2 Contributions

To formalise our solutions, we state them as five concrete contributions.

C1 A thorough formalisation of Lisper’s method along with simplifications, generalisations, improvements and clarifications w.r.t. the original publication. This contribution is in response to Q1.
C2 A prototype implementation of the method to check its feasibility, along with initial experiences of the method. This contribution is in response to Q1.

C3 An efficient algorithm for parametric WCET calculation, along with an evaluation and comparison to a previous approach. This contribution is in response to Q2.

C4 An improved flow analysis which is safe for programs using integers with wrap-around semantics including implementation and comparison to a previous approach. This contribution is in response to Q3.

C5 An implementation of Lisper’s method into the static WCET analysis tool SWEET. This contribution is in response to Q1.

2.4.3 Summary of Publications

Parts of the thesis are based on the following scientific publications:

Paper A


This paper presents a prototype implementation of the parametric WCET analysis introduced in [76]. The paper presents necessary workarounds to make a functioning implementation as well as some simplifications that can be done to reduce complexity. It contains parts of C2. I was the main author and contributor of this paper, the implementation and most of the work arounds were my contributions.

Paper B


This paper introduces the new parametric calculation algorithm. The paper presents the algorithm and evaluates it on a set of benchmarks. It contains C3. I was the main author and inventor of the presented algorithm.
20 Chapter 2. Research

Paper C

Fully Bounded Polyhedral Analysis of Integers with Wrapping. Stefan Bygde, Björn Lisper, Niklas Holsti. Presented at NSAD’11 [26].
This paper describes a new improved flow analysis which is safe to use in programs with integer wrapping behaviour, which is common in embedded system software, especially in processors with short word length. This paper presents the ideas and the approach but does not contain an evaluation. However, this thesis contains an evaluation of the approach. Paper C contains parts of C4. I was the main author of this paper and worked out the details of the method.

Paper X


Although this report is not formally contained within this thesis, it contains the results of the initial literature study made in response to Q3, and is therefore related.

2.5 Thesis Outline

Figure 2.1 summarises the relation between the chapters of this thesis and the research questions, publications and contributions.

![Figure 2.1: Relation between research questions, chapters, publications and contributions](image)

The rest of the thesis is outlined as follows:

Chapter 3 gives an introduction to static analysis and formalises our basic approach to static analysis.
Chapter 4 gives an introduction to abstract interpretation and the polyhedral domain.

Chapter 5 introduces our parametric WCET analysis.

Chapter 6 describes our new algorithm for parametric WCET calculation.

Chapter 7 presents the improved flow analysis which is safe for integers with wrap-around behaviour.

Chapter 9 shows experimental results for the different methods presented in this thesis.

Chapter 10 concludes the thesis with summary, conclusions and suggestions for future work.
Chapter 3
Technical Introduction

This chapter introduces the theoretical foundations of our approach to static analysis. Our method is highly inspired by, and to a large extent equivalent to, Lisper's method [77, 76]. However, our method is developed in a systematic and progressive manner over trace semantics, along with proofs of correctness.

3.1 Notation and Conventions

In this section we present all conventions and notation used throughout this thesis. Many concepts are described as tuples, where the tuple has a name $T$ and each component of the tuple is subscripted with the tuple name to identify it, i.e., $T = \langle A, B, C \rangle$, these subscripts are included when needed for emphasis, but are generally omitted when obvious from the context.

3.1.1 Functions

Functions are very important for the development of static analysis theory, so functions will generally be treated as first-order elements of a set. The notation $A \rightarrow B$ denotes the set of total functions with domain $A$ and range $B$. The notation $f: A \rightarrow B$ means that $f$ is a function mapping an element in $A$ to an element in $B$. The set $A \rightarrow B$ denotes the set of partial functions, where $f(a)$ may be undefined (written as $f(a) = \perp$) for some $f \in A \rightarrow B$. This means that $(A \rightarrow B) \subseteq (A \rightarrow B)$.

In addition, we often use higher order functions, i.e., functions which may have functions as domain or range. As an example $g \in A \rightarrow (B \rightarrow C)$ is a...
Chapter 3

Technical Introduction

This chapter introduces the theoretical foundations of our approach to static analysis. Our method is highly inspired by, and to a large extent equivalent to, Lisper’s method [77, 76]. However, our method is developed in a systematic and progressive manner over trace semantics, along with proofs of correctness.

3.1 Notation and Conventions

In this section we present all conventions and notation used throughout this thesis. Many concepts are described as tuples, where the tuple has a name $T$ and each component of the tuple is subscripted with the tuple name to identify it, i.e., $T = (A_T, B_T, C_T)$, these subscripts are included when needed for emphasis, but are generally omitted when obvious from the context.

3.1.1 Functions

Functions are very important for the development of static analysis theory, so functions will generally be treated as first-order elements of a set. The notation $A \to B$ denotes the set of total functions with domain $A$ and range $B$. The notation $f : A \to B$ means that $f$ is a function mapping an element in $A$ to an element in $B$. The set $A \leftarrow B$ denotes the set of partial functions, where $f(a)$ may be undefined (written as $f(a) = \bot$) for some $f \in A$. This means that $(A \to B) \subseteq (A \leftarrow B)$.

In addition, we often use higher order functions, i.e., functions which may have functions as domain or range. As an example $g \in A \to (B \to C)$ is a
function mapping an element in $A$ to a function. Function application is written with parenthesis and is left associative, so if $a \in A$, $b \in B$ then $g(a)(b) \in C$.

Furthermore, to treat functions as elements we often use lambda abstraction (see for example [104]) to define functions. The expression $z = \lambda x.y$ where $x$ is a symbol and $y$ is an expression is a function with one argument $x$. We say that $x$ is bound in $z$ and that all other variables occurring in $y$ are free. Since $z$ is a function, we can perform function application and compute $z(u) = (\lambda x.y)(u) = y'$, where $y'$ equals $y$ in which all occurrences of $x$ have been replaced by $u$. So, for example $(\lambda x.x+2)(3) = 3+2 = 5$. Some common functions defined using lambda abstraction are $\lambda x.x$ as the identity function, since $(\lambda x.x)(y) = y$, or a constant function $\lambda x.y$ (where $y$ is a variable) since $(\lambda x.y)(z) = y$, or the completely undefined function $\lambda x.\bot$. Lambda abstraction $\lambda x$ has lower precedence than any other operator, meaning that is $y$ is any expression, $\lambda x.y$ should be understood as $\lambda x.(y)$.

Finally, we define some common operations of functions: $f \circ g$ denotes function composition, so $(f \circ g)(x) = f(g(x))$. A function is $f \in A \rightarrow B$ injective if $f(a) = f(b) \Rightarrow a = b$ and surjective if $\forall b. \exists a : f(a) = b$. A function which is both injective and surjective is bijective. Any bijective function $f$ has an inverse $f^{-1} : B \rightarrow A$, where $f^{-1}(b) = a$ iff $f(a) = b$.

The notation $f[x \mapsto y]$ denotes

$$f[x \mapsto y](z) = \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise.} \end{cases}$$

The latter notation allows us to define functions compositionally. Consider for example $(\lambda x.\bot)[a \mapsto 1][b \mapsto 2]$ completely defines a function with domain $\{a, b\}$. However, to avoid clutter, we generally just write $[a \mapsto 1][b \mapsto 2]$.

### 3.2 Program Model

Static analysis analyses the source or object code of a program. In order to do this, programs need to be represented in a convenient manner. To keep the presentation of our method as simple and straightforward as possible, we present a very simple intra-procedural model for the programs to be analysed in this section. To give a quick overview of assumptions used in the model, these assumptions have been clearly marked. Note that some of the assumptions are there mainly to simplify the presentation and can easily be relaxed, while other assumptions are specific and necessary to the method.
3.2 Program Model

3.2.1 Syntax

Definition 1. A program $P$ is represented by a flow chart of nodes $V_P$ and arcs $Q_P \subseteq V_P \times V_P$. The arcs will also be referred to as program points, because analysis results are often associated with the arcs in the program graph.

![Flow Chart Nodes](image)

Figure 3.1: Flow Chart Nodes (from left to right): The start node, the basic block node, the join node, the conditional node and the exit node.

Each node $n \in V_P$ of a flow chart has a type. Below, the node types seen in Figure 3.1 are explained from left to right.

**Start node** Each program has exactly one start node. The start has one outgoing arc and no incoming arcs. The outgoing arc of the start node is referred to as the *initial program point* and is generally denoted $q_0$.

**Basic block node** A basic block node has one incoming and one outgoing arc. Each basic block node is associated with an assignment list explained below. We assume that two basic blocks are never connected directly without any other node type in between.

**Join Node** The join node has two incoming arcs and one outgoing arc. If one of the incoming arcs is a back arc\(^1\) of the graph, then the join node is classified as a *loop join*.

**Conditional Node** The conditional node has one incoming arc and two outgoing arcs, one of the outgoing arcs is labeled $\text{true}$ and the other outgoing arc is labeled $\text{false}$. This node is associated with a $b\text{exp}$ to explained below.

**Exit Node** Each program has exactly one exit node. The exit node has one incoming arc and no outgoing arcs. The program point associated with the incoming arc is referred to as the *final program point*.

\(^1\)An arc coming from a descendant in the graph.
Assumption 1. All programs are assumed to have single entry and exit points.

Each basic block node is associated with a string representing a list of assignments, assignmentlist. This category of strings is represented by the following grammar:

\[
\text{assignmentlist} \rightarrow \epsilon \mid \text{assignment}, \text{assignmentlist} \\
\text{assignment} \rightarrow \text{var} := \text{aexp} \\
\text{aexp} \rightarrow \text{num} \mid \text{var} \mid \text{op}_A^1 \text{aexp} \mid \text{aexp} \text{op}_A^2 \text{aexp} \mid (\text{aexp})
\]

where \(\epsilon\) represents the empty string, \(\text{var}\) is a set of symbols representing program variables, \(\text{num}\) is an integer and \(\text{op}_A^1\) and \(\text{op}_A^2\) represent unary and binary arithmetic operations.

The precedence between arithmetic operators is not reflected in the grammar, it is assumed that the natural precedence of operators are used or that it is resolved using parentheses. Using integers for the category \(\text{num}\) implies that we only consider integer values in our programs. This is for several reasons: it makes the presentation simpler, it can be generalised fairly simply and most analyses we explore are concerned only with integers.

Assumption 2. All variables and expressions are integer valued.

Each conditional node is associated with a bexp, which is a Boolean expression specified by the following grammar:

\[
\text{bexp} \rightarrow \text{true} \mid \text{false} \mid \text{not} \text{bexp} \mid \text{aexp} \text{op}_R^2 \text{aexp} \mid \text{bexp} \text{op}_B^2 \text{bexp} \mid (\text{bexp})
\]

For a program \(P\), we define the set \(\text{VAR}_P\) as the set of all variables occurring in any string in the associated basic block nodes and conditional nodes, in other words, the set of variables occurring in \(P\). We define the functions from, to : \(Q_P \rightarrow V_P\) as

\[
\text{from}((n_1, n_2)) = n_1 \\
\text{to}((n_1, n_2)) = n_2
\]

to easily be able to refer to the nodes associated with a program point.

An example program \(L = (V_L, \{q_0, q_1, q_2, q_3, q_4, q_5\})\) is depicted in Figure 3.2. This program is used as a running example of the analysis techniques throughout the thesis.
3.2 Program Model

3.2.2 Semantics

The previous section defined how to represent programs. In this section we attach a formal meaning to the flow chart nodes in order to define what happens during execution of a program.

**Definition 2.** An environment of a program $P$ is a mapping $\sigma : \text{VAR}_P \rightarrow \mathbb{Z}$. In other words, an environment is an assignment for every variable in the program to an integer. The set of all environments of a program $P$ is denoted $\text{ENV}_P$.

Before program flow is introduced in the semantics, we define how flow chart nodes affect the environment. To do this, we introduce the semantics of nodes as a function $S_{\text{node}} : V_P \rightarrow (\text{ENV}_P \rightarrow \text{ENV}_P)$.

Let $n \in V_P$, then if $n$ not a basic block node:

$$S_{\text{node}}(n) = \lambda \sigma. \sigma$$

and if $n$ is a basic block node associated with the assignment list $l$:

$$S_{\text{node}}(n) = S_{\text{assignment list}}(l)$$

where $S_{\text{assignment list}}$ is defined as follows

$$S_{\text{assignment list}}(\epsilon) = \lambda \sigma. \sigma$$

$$S_{\text{assignment list}}(a, l) = S_{\text{assignment list}}(l) \circ S_{\text{assignment}}(a)$$

$$S_{\text{assignment}}(v := e) = \lambda \sigma. \sigma[v \mapsto A(e)(\sigma)]$$
The function $A : aexp \to (\Sigma_P \to \mathbb{Z})$, evaluates an arithmetic expression in terms of a given environment. Formally:

$$A(n) = \lambda \sigma. n \quad \text{if } n \text{ is a number}$$
$$A(v) = \lambda \sigma. v \quad \text{if } v \text{ is a variable}$$
$$A(\sim a) = \lambda \sigma. (\sim A(a)(\sigma)) \quad \text{if } \sim \in \text{op}_A^1$$
$$A(a_1 * a_2) = \lambda \sigma. (A(a_1)(\sigma) * A(a_2)(\sigma)) \quad \text{if } * \in \text{op}_A^2$$

That is, given an arithmetic expression, all variables $v$ in the expression are substituted for $\sigma(v)$ resulting in a concrete number. The * operator denotes the semantic equivalent of the syntactical operator *. Again, this definition ignores operator precedence, but it would be simple to extend the grammar to accommodate it.

**Assumption 3.** We assume that undefined behaviour such as division by zero does not occur in our programs.

Consider the basic block node in Figure 3.2 associated with the assignment $i := i + 1$. Call this node $n_0$, and assume that $\sigma_0 = [i \mapsto 0][n \mapsto 2]$, then:

$$S_{\text{node}}(n_0)(\sigma_0) = S_{\text{assignment list}}(i := i + 1, \epsilon)(\sigma_0)$$
$$= (S_{\text{assignment list}}(\epsilon) \circ S_{\text{assignment}}(i := i + 1))(\sigma_0)$$
$$= S_{\text{assignment}}(i := i + 1)(\sigma_0) = \sigma_0[i \mapsto A(i + 1)(\sigma_0)]$$
$$= \sigma_0[i \mapsto \{A(i)(\sigma_0) + A(1)(\sigma_0)] = \sigma_0[i \mapsto \sigma_0(i) + 1]$$
$$= \sigma_0[i \mapsto 1] = [i \mapsto 1][n \mapsto 2]$$

as expected. Having defined the semantics of the nodes, we introduce the semantics concerning the program flow.

**Definition 3.** A state $\langle q, \sigma \rangle \in \text{STATE}_P = Q_P \times \text{ENV}_P$ of a program $P$ is a program point associated with an environment.

Informally, an environment can be said to be a memory configuration and a state is a memory configuration at a certain point of the program. A state completely describes an instant in the execution of a program. Since the program point and the environment of an state is often referred to, we introduce the following notation. If $s_j = \langle q_j, \sigma_j \rangle$, we use the suffix operations $\cdot q : \text{STATE}_P \to Q_P$ and $\cdot \sigma : \text{STATE}_P \to \Sigma_P$, with the following definitions:

$$s_j \cdot q = q_j$$
$$s_j \cdot \sigma = \sigma_j$$
The semantic function $\tau_P$ of a program $P$ is a mapping $\tau_P : \text{STATE}_P \rightarrow \text{STATE}_P$, mapping one state to another. Intuitively, the semantic function performs one step of execution. Formally, the semantic function defines how the program transforms any given state. Note that defining $\tau_P$ as a function makes our program model deterministic.

**Assumption 4.** We assume that programs under analysis are deterministic.

The semantic function over flow graphs is defined as follows. Let $q = \langle n_0, n_1 \rangle, q' = \langle n_1, n_2 \rangle$ such that $n_1$ is a basic block node or a join node. Then:

$$\tau_P(\langle q, \sigma \rangle) = \langle q', S_{\text{node}}(n_1)(\sigma) \rangle$$

Let $n_1$ be a conditional node associated with the Boolean condition $b$, and let $q = \langle n_0, n_1 \rangle, q_{\text{true}} = \langle n_1, n_2 \rangle, q_{\text{false}} = \langle n_1, n_2' \rangle$ where $q_{\text{true}}, q_{\text{false}}$ correspond to the program points taken depending on the evaluation of $b$, then

$$\tau_P(\langle q, \sigma \rangle) = \begin{cases} \langle q_{\text{true}}, \sigma \rangle & \text{if } B(b)(\sigma) = \text{true} \\ \langle q_{\text{false}}, \sigma \rangle & \text{if } B(b)(\sigma) = \text{false} \end{cases}$$

where $B : \text{bexp} \rightarrow \Sigma_P \rightarrow \{\text{true, false}\}$ is defined in a similar fashion as $A$, but it evaluates a Boolean expression to true or false depending on the given environment.

$$B(\text{true}) = \lambda \sigma.\text{true}$$
$$B(\text{false}) = \lambda \sigma.\text{false}$$
$$B(\text{not } b) = \lambda \sigma. (\text{not } B(b)(\sigma))$$
$$B(b_0 * b_1) = \lambda \sigma. (B(b_0)(\sigma) * B(b_1)(\sigma)) \quad \text{if } * \in \text{op}_B^2$$
$$B(a_0 \sim a_1) = \lambda \sigma. (A(a_0)(\sigma) \sim A(a_1)(\sigma)) \quad \text{if } \sim \in \text{op}_R^2$$

As an example, consider $q_2$ in Figure 3.2, assume that $\sigma_0 = [i \mapsto 1][n \mapsto 2]$ and that $q_4$ corresponds to true and $q_3$ corresponds to false. Then

$$\tau_L(\langle q_2, \sigma_0 \rangle) = \langle q_4, \sigma_0 \rangle$$

since

$$B(i <= n)(\sigma_0) = B(i(\sigma_0)) \leq B(n)(\sigma_0)$$
$$= \sigma_0(i) \leq \sigma_0(n) = 1 \leq 2 = \text{true}$$
Finally, if \( q = \langle n_0, n_1 \rangle \) such that \( n_1 \) is the exit node, then
\[
\tau_P(q, \sigma) = \langle q, \sigma \rangle
\]

**Definition 4.** We define a state as an initial state \( s_0 \) if \( s_0.q = q_0 \), i.e., if the associated program point of the state is the initial program point. We define the set of all initial states of a program as
\[
I_P = \{ s \in \text{STATE}_P \mid s.q = q_0 \}
\]

Furthermore, for any \( i \in I_P \), \( i.\sigma \) is defined as an input of a program.

Conversely, a state is defined as a final state if the associated program point is the final program point. We denote the set of final states \( F_P \).

**Definition 5.** The semantic closure function \( \tau_P^* : I_P \rightarrow F_P \) of a program is recursively defined as
\[
\tau_P^*(s) = \begin{cases} 
  s & \text{if } s \in F_P \\
  \tau_P^*(\tau_P(s)) & \text{otherwise}
\end{cases}
\]

When defined, the semantic closure function maps an initial state into a final state. In this way, the semantic closure function completely defines an execution. Note that if \( P \) does not terminate on state \( s \), then \( \tau_P^*(s) \) is undefined.

### 3.2.3 Example

We demonstrate how the semantic closure function models execution by executing \( L \) (see Figure 3.2) on input \( \sigma_0 = [i \mapsto 0][n \mapsto 2] \). This is done by computing \( \tau_L^*(s_0) \) from the initial state \( s_0 = \langle q_0, \sigma_0 \rangle \).

\[
\begin{align*}
\tau_L^*(\langle q_0, [i \mapsto 0][n \mapsto 2] \rangle) &= \tau_L^*(\langle q_1, [i \mapsto 0][n \mapsto 2] \rangle) \\
\tau_L^*(\langle q_2, [i \mapsto 0][n \mapsto 2] \rangle) &= \tau_L^*(\langle q_4, [i \mapsto 0][n \mapsto 2] \rangle) \\
\tau_L^*(\langle q_5, [i \mapsto 1][n \mapsto 2] \rangle) &= \tau_L^*(\langle q_2, [i \mapsto 1][n \mapsto 2] \rangle) \\
\tau_L^*(\langle q_4, [i \mapsto 1][n \mapsto 2] \rangle) &= \tau_L^*(\langle q_5, [i \mapsto 2][n \mapsto 2] \rangle) \\
\tau_L^*(\langle q_2, [i \mapsto 2][n \mapsto 2] \rangle) &= \tau_L^*(\langle q_4, [i \mapsto 2][n \mapsto 2] \rangle) \\
\tau_L^*(\langle q_5, [i \mapsto 3][n \mapsto 2] \rangle) &= \tau_L^*(\langle q_2, [i \mapsto 3][n \mapsto 2] \rangle) \\
\tau_L^*(\langle q_3, [i \mapsto 3][n \mapsto 2] \rangle) &= \langle q_3, [i \mapsto 3][n \mapsto 2] \rangle.
\end{align*}
\]

We can conclude that
\[
\tau_L^*(\langle q_0, [i \mapsto 0][n \mapsto 2] \rangle) = \langle q_3, [i \mapsto 3][n \mapsto 2] \rangle
\]
3.2.4 Program Timing

In our program model, each program point \( q \) is associated with a value \( \ell_P(q) \). This value represents the execution time of the program instructions modelled by the flow-chart node which precedes \( q \), and includes the execution time of the control-flow transition modelled by the arc \( q \) itself. The value \( \ell_P(q) \) may represent milliseconds, clock cycles or any suitable measure of time.

**Assumption 5.** Each program point \( q \) is considered to have a constant execution time \( \ell_P(q) \).

In reality the execution time of each program point may vary due to the current state of the hardware. A good low-level analysis would be able to take this into account and make context-sensitive estimations for the atomic worst-case execution times. It would be possible to model program points so that they model different contexts to utilise this. As an example, loops can be unrolled to separate the set of program points for the first iteration of the loop from the others. This is a common way of modelling cache-effects. In general a more complex definition of “program point” can be used, that takes the context into consideration. However, in our model program points are independent of context, so a constant time is used as model here. To make this safe, this constant time would be taken as the worst-case execution time for the node.

Below are the execution times of the running example \( L \) shown (these have been selected arbitrarily).

\[
\begin{align*}
\ell_L(q_0) &= 1 & \ell_L(q_1) &= 3 \\
\ell_L(q_2) &= 1 & \ell_L(q_3) &= 2 \\
\ell_L(q_4) &= 2 & \ell_L(q_5) &= 8
\end{align*}
\]  

(3.1)

3.3 Traces

Given an initial state \( s = (q_0, \sigma) \) we define a trace as follows.

**Definition 6.** Let \( \sigma \) be an environment. Then a trace \( T_\sigma : \mathbb{N} \rightarrow \text{STATE}_P \) is a function defined as

\[
\begin{align*}
T_\sigma(0) &= (q_0, \sigma) \\
T_\sigma(n) &= \tau_P(T_\sigma(n - 1))
\end{align*}
\]
That is, \( T_\sigma(n) \) represents the \( n \)-th step of execution starting on the initial program point with input \( \sigma \). Note that if there is an \( n \) such that \( T_\sigma(n) \in F_P \), then \( T_\sigma(m) = T_\sigma(n) \in F_P \) for all \( m \geq n \). In general, if the input is not important, the subscript \( \sigma \) is omitted.

**Definition 7.** Let \( T_\sigma \) be a trace, then

\[
\text{length}(T_\sigma) = \text{Min} \{ n \in \mathbb{N} \mid T_\sigma(n) \in F_P \} + 1
\]

where we define \( \text{Min}(\emptyset) = \infty \). If \( \text{length}(T_\sigma) = \infty \) we say that \( T_\sigma \) is infinite (modelling a non-terminating execution), and finite (modelling a terminating execution) otherwise.

**Definition 8.** Let \( s \) be a state and \( T \) be a trace, then we define the following

\[
\text{states}(T) = \{ T(n) \mid n \in \mathbb{N} \}
\]

\( s \prec T \iff s \in \text{states}(T) \)

The relation \( s \prec T \) is used to denote that a state belongs to a trace. The function \( \text{states}(T) \) is defined as all states that belongs to \( T \).

**Lemma 1.** Let \( T \) be a finite trace of length \( n \), and let \( i, j < n \) be integers. Then,

\[
T(i) = T(j) \Rightarrow i = j
\]

**Proof.** Assume that \( T(i) = T(j) \). Then, by definition 6, \( T(i+1) = \tau_P(T(i)) = \tau_P(T(j)) = T(j+1) \). Thus, in general it must hold that for any non-negative integer \( m \), \( T(i+m) = T(j+m) \). Assume for contradiction that \( i < j \) and let \( m_0 = n - j - 1 \), which is a non-negative number since \( j < n \). Then \( i + m_0 < j + m_0 = n - 1 \), while \( T(i + m_0) = T(j + m_0) \) by assumption. By definition 7, \( \forall k < n-1 : T(k) \notin F_P \), so we have \( T(j + m_0) = T(n-1) \in F_P \) while \( T(i + m_0) \notin F_P \), which is a contradiction so the assumption that \( i < j \) must be false. The same conclusion is reached for the assumption \( j < i \), so we must have \( i = j \). \( \Box \)

Another way to state Lemma 1 is that \( T \) is injective when restricted to input smaller than \( n \). This means that no non-final state is visited more than once per finite trace.

**Proposition 1.** If \( T \) is a finite trace, then

\[
\text{length}(T) = |\text{states}(T)|
\]
3.3 Traces

Proof. Let length(T) = n₀. Then,

\[ \text{states}(T) = \{ T(n) \mid 0 \leq n < n₀ \} \cup \{ T(n) \mid n \geq n₀ \}. \]

Due to the definition of length we have that \( \{ T(n) \mid n \geq n₀ \} = \{ T(n₀) \} \), and \( T(n₀ - 1) = T(n₀) \) so \( T(n₀) \in \{ T(n) \mid 0 \leq n < n₀ \} \). Thus,

\[ \text{states}(T) = \{ T(n) \mid 0 \leq n < n₀ \} \cup \{ T(n) \mid n \geq n₀ \} \]

\[ = \{ T(n) \mid 0 \leq n < n₀ \} \]

Since T is injective for all \( n < n₀ \) (Lemma 1), we have that \( |\{ T(n) \mid 0 \leq n < n₀ \}| = n₀. \)

3.3.1 Trace Semantics

Trace semantics [33] is informally defined as all possible execution traces of a program. Formally the trace semantics of a program \( P \) is defined as:

\[ \mathcal{T}S_P = \{ T_\sigma \mid \sigma \in \text{ENV}_P \}. \]

Thus, the trace semantics \( \mathcal{T}S_P \) is the set of all complete execution traces of a program.

3.3.2 Using Trace Semantics to Compute the Execution Time

Given \( \ell_P \) and a finite trace \( T_\sigma \), the execution time of \( T_\sigma \) is computed by:

\[ \text{ET}_P(T_\sigma) = \sum_{s \in \text{states}(T_\sigma)} \ell_P(s,q) \quad (3.2) \]

That is, the accumulated cost for each time a program point is visited by a state in a trace. Note that since the computation is based on \( \text{states}(T_\sigma) \), this definition of the execution time only works if \( T_\sigma \) is a finite trace.

Assumption 6. We assume that all programs under analysis are always terminating. In other words, we assume that a program under analysis contain only finite traces.

This might seem like a strong assumption, however, making WCET analysis on a program that might not terminate would not yield a useful result anyway. What should be noted though is that \( \text{ET}(T) \) would evaluate to a finite
The output of the parametric WCET analysis is a function that takes an input \( \sigma \) starting with input \( V \times Q \rightarrow \mathbb{N} \) (coming from the low-level analysis). The program \( P \) is supposed to do. The idea is that, given a program \( P \in \sigma \) has to compute all the sets \( Z \) has to compute a safe upper bound of the execution time of a trace. Formally, the parametric analysis computes the following function: 

\[
\text{ET}_{\sigma}(P) = \sum_{s \in \text{states}(P)} \ell_L(s, q) = 40
\]

### 3.4 Parametric WCET Analysis

Having defined programs, program timing, program input and safe bounds on execution time, we can now define exactly what our parametric WCET analysis is supposed to do. The output of the parametric WCET analysis is a function that takes an input \( \sigma \in \text{VAR}_P \rightarrow \mathbb{Z} \) and returns a safe estimate of the WCET of \( P \) when executing.
$T_\sigma$. Formally, the parametric analysis computes the following function:

$$\text{Input to analysis } (P, \ell_P) \xrightarrow{\left( (V_P \times Q_P) \times (Q_P \rightarrow \mathbb{N}) \right)} \text{Output from analysis } \text{PWCET}_P(\sigma) \xrightarrow{\left( (\text{VAR}_P \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z} \right)} \text{WCET estimate}$$

The program $P$ and basic block timing $\ell_P$ are inputs, and the program input $\sigma$ is processed to compute a safe upper bound of the execution time of a trace. This means that the result of the analysis should be a function.

An exact parametric analysis is represented by the equation

$$\text{PWCET}_P(\sigma) = ET_P(T_\sigma)$$

Unfortunately, it is not possible for a program to determine the function $ET_P$. The reason is that, to determine $ET_P$, the analysis would have to compute all the sets $\text{states}(T_\sigma)$ for all $T_\sigma \in \mathcal{T}S_P$. However, $\mathcal{T}S_P$ is not computable in the general case, and even if the analysed programs were to be restricted to terminating programs with a finite input domain, it would be infeasible in practice to compute $\mathcal{T}S_P$ as it essentially contains all information about every possible execution of $P$. This means that $\text{PWCET}_P(\sigma)$ has to compute an approximation of $ET_P$, and in particular a safe approximation. That is, we seek the property

$$\forall \sigma \in \text{ENV}_P : \text{PWCET}_P(\sigma) \geq ET_P(T_\sigma)$$

**Proposition 2.** Let $\widehat{\text{states}}$ be a function such that $\text{states}(T) \subseteq \widehat{\text{states}}(T)$ for all traces $T$. Then, trivially we have that

$$ET_P(T) \leq \sum_{s \in \text{states}(T)} \ell_P(s, q)$$

This means that any function $\widehat{\text{states}}$ with the above property can be used to compute a safe upper bound of the execution time of a trace.

### 3.4.1 Making it a Counting Problem

Proposition 2 shows how the timing of a trace can be approximated by finding at least all states that belongs to it and summing their individual times. Another way of stating the problem is to take each program point $q$ in the program and multiply its timing $\ell(q)$ with the number of times it can be visited, i.e., with the number of states associated with $q$. This way of representing the problem is more convenient since the problem is reduced to finding a number for each
Proof. Then \( \phi \) function defined as:

\[
\phi(S) = \lambda q.\{ \sigma \in \text{ENV} \mid \langle q, \sigma \rangle \in S \}
\]

That is, \( \phi(S) \) is a function mapping each program point to the set of environments that are associated with it.

We will now state a number of properties of the function \( \phi \) in the general case, in order to show how Proposition 2 can be reformulated as a counting problem.

**Lemma 2.** Let \( A, B \) be finite sets and let \( \phi: \mathcal{P}(A \times B) \to (A \to \mathcal{P}(B)) \) be a function defined as:

\[
\phi(S) = \lambda a.\{ b \in B \mid \langle a, b \rangle \in S \}
\]

Then \( \phi \) is bijective (that is, injective and surjective).

**Proof.** To see that \( \phi \) is injective, assume that \( \phi(S) = \phi(S') \) for some sets \( S, S' \). This means that for all \( a \in A \) we have that \( \phi(S)(a) = \phi(S')(a) \). Assume for contradiction that for all \( a \in A \) we have that \( \langle a, b \rangle \in S \) but \( \langle a, b \rangle \notin S' \). This would mean that \( b \in \phi(S)(a) \) but \( b \notin \phi(S')(a) \) which is a contradiction. By symmetry, the same holds if the element were chosen to be in \( S' \) but not in \( S \). Thus, we must conclude that \( S = S' \) which makes \( \phi \) injective.

To see that \( \phi \) is surjective (or onto), let \( f: A \to \mathcal{P}(B) \) be any function. We now construct a set \( S \) such that \( \phi(S) = f \) as follows:

\[
S = \bigcup_{a \in A} \{ \langle a, b \rangle \mid b \in f(a) \}
\]

Let \( a_0 \in A \), then we have:

\[
\phi(S)(a_0) = \{ b' \in B \mid \langle a_0, b' \rangle \in \bigcup_{a \in A} \{ \langle a, b \rangle \mid b \in f(a) \} \}
\]

\[
= \{ b' \in B \mid \langle a_0, b' \rangle \in \{ \langle a_0, b \rangle \mid b \in f(a_0) \} \}
\]

\[
= \{ b' \in B \mid b' \in f(a_0) \} = f(a_0)
\]

Since \( a_0 \) was taken arbitrarily this holds for all \( a \in A \) so that \( \phi(S) = f \). Thus, \( \phi \) is both injective and surjective and therefore bijective. \( \square \)
Lemma 3. Let $S, S' \subseteq A \times B$, then the following holds

$$S \subseteq S' \iff \forall a \in A : \phi(S)(a) \subseteq \phi(S')(a)$$

Proof. Let $S \subseteq S'$ and let $a \in A$. If $\phi(S')(a) = \emptyset$, then there is no $S'$ contains no element on the form $(a, b)$, and neither do $S$ since $S \subseteq S'$, so $\phi(S)(a) = \emptyset$ as well. Otherwise, let $b \in \phi(S')(a)$, then $(a, b) \in S'$ which implies that $(a, b) \in S$ since $S \subseteq S'$. Since $(a, b) \in S$, we have that $b \in \phi(S)(a)$. Since $a$ and $b$ were chosen arbitrarily, this holds for all $a$ and for all $b \in \phi(S')(a)$. \qed

Proposition 3. Let $S \subseteq A \times B$, and let $v : A \to \mathbb{Z}$ be a valuation of $A$, then

$$\sum_{(a, b) \in S} v(a) = \sum_{a \in A} |\phi(S)(a)| v(a)$$

Proof. First we see that

$$\sum_{(a, b) \in S} v(a) = \sum_{a' \in A} \left( \sum_{(a', b) \in S} v(a') \right)$$

That is, we split the summation into two parts; the outer summation goes through all elements $a'$ in $A$, and the inner summation sums the valuations for each pair in $S$ in which the first component is $a'$. Next we have,

$$\sum_{a' \in A} \left( \sum_{(a', b) \in S} v(a') \right) = \sum_{a' \in A} \left( \sum_{b \in \phi(S)(a')} v(a') \right)$$

by the definition of $\phi$, finally we have

$$\sum_{a' \in A} \left( \sum_{b \in \phi(S)(a')} v(a') \right) = \sum_{a' \in A} |\phi(S)(a')| v(a') = \sum_{a \in A} |\phi(S)(a)| v(a)$$

since $v(a')$ is fixed for the inner summation, all that matters is the number of elements $b$ to be summed over. \qed

Corollary 1.

$$\sum_{s \in \text{states}(T)} \ell(s.q) = \sum_{q \in \mathcal{Q}} C(T, q) \ell(q) \quad (3.3)$$

where

$$C(T, q) = |\phi(\text{states}(T))(q)|$$
Proof. This result follows directly from Proposition 3 by substituting \( \ell \) for \( v \), \( states(T) \) for \( S, Q \) for \( A \) and \( Env \) for \( B \). \( \square \)

The problem has now been reduced to determining \( C(T, q) \). \( C(T, q) \) is the number of environments associated with program point \( q \) in \( T \), which is the same as the number of times \( T \) visits \( q \), when \( T \) is finite. As an example, consider \( L \) and Table 3.1. Table 3.2 shows \( C(T_0, q) \) for the six program points of \( L \). Summing these now yields,

\[
\sum_{q \in Q_L} C(T_0, q)\ell_L(q) = \ell_L(q_0) + \ell_L(q_1) + 4\ell_L(q_2) + \ell_L(q_3) + 3\ell_L(q_4) + 3\ell_L(q_5) = 1 + 3 + 4 \cdot 1 + 2 + 3 \cdot 2 + 3 \cdot 8 = 40
\]

As expected, the result is the same as computed from (3.2).

Definition 9. The function \( C : ((\mathbb{N} \rightarrow \text{STATE}_P) \times Q_P) \rightarrow \mathbb{N} \), defined as

\[
C(T, q) = |\phi(\widehat{\text{states}}(T))(q)|
\]

where \( \text{states}(T) \subseteq \widehat{\text{states}}(T) \) for all traces \( T \), is called the Census function for this approximation of \( \text{states}(T) \).

The name Census\(^2\) comes from the fact that the “population” of states visited at each program point is counted.

We can now set

\[
PWCET_P(\sigma) = \sum_{q \in Q_P} C(T_\sigma, q)\ell_P(q)
\]

\[
= \sum_{s \in \text{states}(T_\sigma)} \ell_P(s.q) \geq \sum_{s \in \text{states}(T_\sigma)} \ell_P(s.q) = ET(T_\sigma)
\]

Note that by using \( \widehat{\text{states}}(T_\sigma) \) instead of \( \text{states}(T_\sigma) \) we have converted this into a computable problem under the assumption that it is possible to compute \( \widehat{\text{states}} \). It is possible to do so by letting \( \widehat{\text{states}} \) be a safe but computable approximation of \( \text{states} \). The rest of the chapter shows the developments towards this approximation.

\(^2\)Credited to Niklas Holst, Tidorum Ltd.
Table 3.2 The number of times the trace $T_0$ visits each of the program points in $L$.

<table>
<thead>
<tr>
<th>$q \in \mathbb{Q}_L$</th>
<th>$C(T_0, q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>$</td>
</tr>
</tbody>
</table>

3.5 Collecting Semantics

As shown in the previous section, we are interested in finding an approximation of the number of states that may visit each program point. We are interested in doing this in an efficiently computable manner. A first approximation of the trace semantics is to consider a set of states rather than a set of traces. Collecting semantics [33] follows the same idea as the trace semantics, but does not keep the information about individual traces or the order in which states was visited. Instead, collecting semantics collects every state in one set rather than in a set of sequences.

To define the collecting semantics of a program we define the following function:

$$CS_P(S) = S \cup \{\tau_P(s) \mid s \in S\} \cup I_P$$

This function takes a set of states $S$ and adds all successors of these states. Note that if $S$ contains all states that can possibly be reached in $P$, then $CS_P(S) = S$ since any successor of a state $s \in S$ is also by definition in $S$. Conversely, if $CS_P(S) = S$, then there are no successors to a state $s \in S$ which are not already in $S$. Moreover, since $I_P \subseteq CS_P(S)$ for any $S \subseteq S_P$, there are no successors to an initial state which are not already in $S$, meaning $S$ contains all states that can be reached from $I_P$. In summary, we can state the following

Proposition 4. A set $S$ contains all reachable states of $P$ iff

$$S = CS_P(S)$$

i.e., iff $S$ is a fixed point of $CS_P$.

In general, the smallest set (w.r.t. inclusion) with this property is known as the least fixed point of the equation (more on this in Chapter 4). The least fixed
point is the most precise and often the most interesting solution to the equation $S = \text{CS}_P(S)$. The least fixed point to the collecting semantics is referred to as $\text{CS}_P$. That is:

$$\text{CS}_P = \text{CS}_P(\text{CS}_P) \land (\forall S : S = \text{CS}_P(S) \Rightarrow \text{CS}_P \subseteq S)$$

Unfortunately, while collecting semantics is approximating trace semantics, collecting semantics is also uncomputable in the general case. However, it is easier to make computable approximations of the collecting semantics than trace semantics, and most approximations in literature are approximating the collecting semantics, which is why we have chosen to base our work on the collecting semantics.

A problem with using collecting semantics is that we have lost the information about which trace a state belongs to. In parametric analysis it is crucial to classify states according to certain criteria. We do so using the well-known concept of equivalence classes.

**Definition 10.** A relation $\sim_E \subseteq S \times S$, where $s \sim_E s'$ means $(s, s') \in \sim_E$ is an equivalence relation iff

- it is reflexive, i.e., $\forall s \in S : s \sim_E s$
- it is symmetric, i.e., $\forall s, s' \in S : s \sim_E s' \Rightarrow s' \sim s$
- it is transitive, i.e., $\forall s, s', s'' \in S : s \sim_E s' \land s' \sim_E s'' \Rightarrow s \sim_E s''$

An equivalence relation gives rise to a partition of $S$ into equivalence classes, such that every element $s \in S$ belongs to exactly one class defined by

$$[s]_E = \{ x \in S \mid x \sim_E s \}$$

The set $S/E = \{ [s]_E \mid s \in S \}$ then consists of all equivalence classes of a set with respect to an equivalence relation.
Definition 11. Let $T, T'$ be traces. Then $T$ and $T'$ are flow equivalent, denoted $T \simeq T'$ iff

$$\forall n \in \mathbb{N} : T(n).q = T'(n).q$$

The $\simeq$ relation is an equivalence relation. If $T \simeq T'$, then $T$ and $T'$ belong to the same essential trace.

Flow equivalent traces traverse the flow chart in exactly the same way, but may produce different environments.

Lemma 4. If $T \simeq T'$, then $ET_P(T) = ET_P(T')$.

Proof. In the computation of $ET_P$, only the program point part is used in the computation. The environment of a state has no impact on the execution time. \qed

While flow-equivalent traces may produce different output, they have the exact same execution time. As will be seen, a subset of variables decides which essential trace, and consequently, which execution time a trace has.

Definition 12. Let $A \subseteq \text{VAR}_P$ and let $\sigma, \sigma'$ be environments, $s, s'$ be states and $T, T'$ be traces. Then we can form the following equivalence relations:

- $\sigma \sim_A \sigma' \Leftrightarrow \forall v \in A : \sigma(v) = \sigma'(v)$
- $s \sim_A s' \Leftrightarrow s.\sigma \sim_A s'.\sigma \land s.q = s'.q$
- $T \sim_A T' \Leftrightarrow \forall n \in \mathbb{N} : T(n) \sim_A T'(n)$

We are using the same notation for three different relations here, and they should be distinguished by their context. Note that $T \sim_A T'$ for any two traces implies $T \simeq T'$ and that $T \simeq T'$ is the same as $T \sim_{A'} T'$. We generalise this concept further. If $P, P'$ are two different programs with a common subset of variables $A$, that is, $A \subseteq \text{VAR}_P$ and $A \subseteq \text{VAR}_{P'}$, then Definition 12 is well-defined even for traces, states and environments belonging to different programs.

Definition 13. Let $P$ be a program. Then $v \in \text{VAR}_P$ is a control variable of $P$ iff

- $v$ occurs in a bexp in a conditional node of $P$, OR
- $v$ occurs on the right hand side of an assignment $x := e$ such that $x$ is a control variable.
The set of control variables of a program $P$ are denoted $\text{CTRL}_P$. Furthermore, two states $s, s'$ are control-equivalent if $s \sim_{\text{CTRL}} s'$.

This is a recursive definition of a variable that may at some point affect the program flow. This definition is context-free and may classify variables that do not actually affect program flow as control variables. As an example, consider the following code snippet

```plaintext
x := y
x := 0
while (x < 10) do
    x := x + 1
end while
```

The variable $y$ would by Definition 13 be classified as a control variable, since it occurs in an assignment of the control variable $x$ (which is control variable because it occurs in a conditional). However, since $x$ is assigned before that value is read, $y$ does not actually affect the flow.

Definition 13 could be defined in a context-sensitive manner, but this would mean that variables can occur both as control and non-control variables at different program points, which would complicate our presentation. Definition 13 is therefore used as model for this presentation, even though it could be defined stricter.

**Lemma 5.** Let $\text{CTRL}$ be the set of control variables of a program and let $s, s'$ be states of a program $P$. Then $s \sim_{\text{CTRL}} s'$ implies $\tau_P(s) \sim_{\text{CTRL}} \tau_P(s')$ (using Definition 12).

**Proof.** Let $s \sim_{\text{CTRL}} s'$ and assume for contradiction that $\tau_P(s).q \neq \tau_P(s').q$. For this to happen, it must mean that $s.q$ has two different successors, so $\text{to}(s.q)$ must be a conditional node (the only node type with two successors) associated with a Boolean expression $b$. Moreover, this means that $B(b)(s.\sigma) \neq B(b)(s'.\sigma)$ since $\tau_P$ maps to two different states. Then there must exist a variable $x$ occurring in $b$, such that $s.\sigma(x) \neq s'.\sigma(x)$. However, since $x$ occurs in $b$, it is by Definition 13 a control variable, and since $s \sim_{\text{CTRL}} s'$ by assumption, we have that $s.\sigma(x) = s'.\sigma(x)$ which is the desired contradiction. This proves that $\tau_P(s).q = \tau_P(s').q$.

Left to prove is that $\tau_P(s).\sigma(v) = \tau_P(s').\sigma(v)$ for all control variables $v$. Assume for contradiction that there exists a control variable $v'$ such that $\tau_P(s).\sigma(v') \neq \tau_P(s').\sigma(v')$. Since $s.\sigma(v') = s'.\sigma(v')$ by assumption, the only possibility is if $v'$ is assigned at the node $n = \text{to}(s.q)$. Thus, let $v' := e$
Lemma 5. Let mean that variables can occur both as control and non-control variables at different points. Proof. Let \(b, s, s'\) be states of a program \(P\). The set of control variables of a program \(P\) is denoted by \(\text{CTRL}(P)\). This means that \(s.\sigma = s'.\sigma[v' \mapsto k]\). This means that \(s.\sigma(v') = s'.\sigma(v')\) which completes the proof.

Lemma 5 shows that the successors of two control-equivalent states are also control-equivalent. A direct consequence is:

Corollary 2. Let \(T, T'\) be states and let \(n, n'\) be natural numbers such that \(T(n) \sim \text{CTRL}(T(n'))\). Then \(T(n + m) \sim \text{CTRL}(T'(n' + m))\), for all non-negative integers \(m\).

Definition 14. Let \(P\) be a program and let \(x := e\) be an assignment associated with a basic block in \(P\) such that \(x\) is not a control variable of \(P\). Then \(x := e\) is called a non-control assignment.

By Definition 13, the expression \(e\) on the right-hand side of a non-control assignment does not contain any control variables. All expressions in a program \(P\) except non-control assignments contain control variables.

Definition 15. Let \(P = \langle V_P, Q_P \rangle\) be a program, then the program \(\text{SLICE}(P) = \langle V_P, Q_P \rangle\) is the same as \(P\) except that in \(\text{SLICE}(P)\) all non-control assignments are removed from the basic blocks.

By this definition we also have that \(\text{SLICE}(P)\) does not contain any non-control variables. Formally stated (proof omitted):

Lemma 6. Let \(P\) be a program, then \(\text{VAR}_{\text{SLICE}(P)} = \text{CTRL}_{\text{SLICE}(P)} = \text{CTRL}_P \subseteq \text{VAR}_P\).

Lemma 7. Let \(P\) be a program, and let \(T_\sigma\) be a trace of this program. Now let \(T'_\sigma\) be a trace of \(\text{SLICE}(P)\) such that \(\forall v \in \text{VAR}_{\text{SLICE}(P)} : \sigma'(v) = \sigma(v)\). Then, \(T_\sigma \sim \text{CTRL}_P, T'_\sigma\).

Proof. According to Definition 12 we have to show that for all \(n \in \mathbb{N}\) we have that \(T_\sigma(n) \sim \text{CTRL}_P T'_\sigma(n)\).

First, \(T_\sigma(0).q = T'_\sigma(0).q = q_0\). Second, we have \(T_\sigma(0).\sigma(v) = \sigma(v) = \sigma'(v) = T'_\sigma(0).\sigma(v)\) for all \(v \in \text{CTRL}_P = \text{CTRL}_{\text{SLICE}(P)}\) so \(T_\sigma(0) \sim \text{CTRL}_P T'_\sigma(0)\). Since \(P\) and \(P'\) share the program structure, including program points and nodes, and they also share control variables, Corollary 2 can be applied to prove the lemma.

\[\square\]
Lemma 7 implies that $\text{ET}(T_\sigma) = \text{ET}(T'_\sigma)$, in other words, for every trace in $P$ there is a trace in $\text{SLICE}(P)$ that has the same timing (by Lemma 4)! This might seem counter-intuitive, but the reason is that we are interested in the running time of the original program $P$, but we analyse the sliced program $\text{SLICE}(P)$, but since the graph structure is equal for $P$ and $\text{SLICE}(P)$, we can analyse $\text{SLICE}(P)$ with the timing $\ell_P$ which were derived from low-level analysis from the original program $P$. In effect we analyse a sliced program to obtain the result of the unsliced one.

Using the sliced program instead of the original has several benefits:

- The sliced program is usually smaller and never larger than the original program, so the analysis is potentially faster.
- The sliced program has less than or equal number of variables than the original program, leading to less complex analysis.
- The sliced program often has much smaller (and never larger) number of potential states to visit during execution.

The last point here is crucial: since the Census method uses the number of states, which might be over-approximated, slicing reduces the number of states, while still having the same timing as the original program. Thus, slicing is necessary to maximise the precision of this method. For this reason, we always analyse the sliced version of any program. A technique called program slicing [117, 118] can be used to obtain a sliced program. Details about how we obtain a sliced program can be found in Chapter 5.

### 3.7 Input Parameters

In this section we introduce the set of variables in a program that are considered to be input parameters. We earlier defined $\sigma$ as input to a trace, but as we will see, the initial value of some variables does not make any difference to the execution of the program, and in particular has no affect on the timing.

**Definition 16.** Let $v$ be a variable and $T$ be a trace. Then $\text{use}_T(v)$ is the smallest non-negative number $n$ such that $\text{from}(T(n))$ is either of the following:

- A basic block node containing the assignment $x:=v$ such that $v$ occurs in $e$.
- A conditional node associated with a Boolean expression $b$ such that $v$ occurs in $b$. 


if such an $n$ exists, otherwise $\infty$.

We define $\text{def}_T(v)$ as the smallest non-negative number $n$ such that the node from$(T(n))$ is a basic block containing the assignment $v:=e$ if such an $n$ exists, $\infty$ otherwise.

Thus, for any trace and variable, $\text{use}_T(v)$ denotes the earliest point in the trace where $v$ is used and $\text{def}_T(v)$ is defined as the earliest point in the trace in which $v$ is assigned.

**Definition 17.** A control variable $v \in \text{VAR}_P$ is an input parameter (or IP), or equivalently, $v \in \text{IP}_P$ iff there exists a trace $T \in \mathcal{T}S_P$ such that $\text{use}_T(v) \leq \text{def}_T(v) < \infty$ and where $v$ is used before (in the sequence of assignments) it is defined in basic block from$(\text{use}_T(v))$.

That is, an IP is a variable which *may* be used before its first assignment in a program. Note that $v$ occurring both on the right hand side and left hand side of an assignment counts as being used first.

**Lemma 8.** Let $\sigma, \sigma'$ be environments such that $\sigma \sim_{1P} \sigma'$, and let $T = T_\sigma, T' = T_{\sigma'}$. Then the following three statements all hold for all $n \in \mathbb{N}$

1. $T(n).q = T'(n).q$

2. $T(n).\sigma \sim_{1P} T'(n).\sigma$

3. $\forall v \in \text{VAR}_P : \text{def}_T(v) < n \Rightarrow T(n).\sigma(v) = T'(n).\sigma(v)$

**Proof.** First we show that the statements hold for the first state in each trace. For statement (1) we have that $T(0).q = T'(0).q = q_0$. For statement (2) we have that $T(0).\sigma = \sigma$ and $T'(0).\sigma = \sigma'$ which by assumption $\sigma \sim_{1P} \sigma'$. Statement (3) is trivially true for $n = 0$.

We make the inductive hypothesis that (1),(2),(3) holds for all non-negative integers $m < n$, and we want to show that (1),(2),(3) holds for $n$.

**Statement 1:**

The only possibility for $T(n).q \neq T'(n).q$ is if from$(T(n).q)$ is a conditional node associated with the Boolean expression $b$ and that $B(b)(T(n-1).\sigma) \neq B(b)(T'(n-1).\sigma)$. This means that there must be a control variable $v$ such that $T(n-1).\sigma(v) \neq T'(n-1).\sigma(v)$. Then $v$ cannot be an IP since that would contradict inductive hypothesis (2) since $n-1 < n$. Since $v$ is used at $n$, i.e., $\text{use}_T(v) = n - 1$ and $v$ is not an IP, we must have that $\text{def}_T(v) < n - 1$
by the converse of Definition 17, however, this is impossible due to inductive hypothesis (3) so the assumption that \( T(n).q \neq T'(n).q \) must be discarded.

**Statement 2:**
The only possibility for \( T(n).\sigma(v) \neq T'(n).\sigma(v) \) for some \( v \in IP_P \) would be if \( \text{from}(T(n).q) \) is a basic block node containing an assignment \( v:=e \) such that \( \mathcal{A}(e)(T(n-1).\sigma) \neq \mathcal{A}(e)(T'(n-1).\sigma) \). Then there must be some control variable \( v' \) occurring in \( e \) such that \( T(n-1).\sigma(v') \neq T'(n-1).\sigma(v') \). According to the inductive hypothesis, this can only happen if the two following conditions are both true:

- \( v' \) is not an IP (by inductive hypothesis (2)), implying by the converse of Definition 17 that \( \text{def}_{T}(v') < \text{use}_{T}(v') \leq n - 1 \).
- \( \text{def}_{T}(v') \geq n - 1 \) (by inductive hypothesis (3))

As can be seen, these cannot both be true, so the assumption that \( T(n).\sigma(v) \neq T'(n).\sigma(v) \) cannot hold.

**Statement 3:**
If \( \text{def}_{T}(v) < n - 2 \), then statement 3 holds by inductive hypothesis, so assume that \( \text{def}_{T}(v) = n - 1 \) meaning that \( \text{from}(T(n-1).q) \) is a basic block node containing the assignment \( v:=e \). The only way for statement 3 to be false would be if there is a control variable \( v' \) occurring in \( e \) such that \( T(n-1).\sigma(v') \neq T'(n-1).\sigma(v') \) which is impossible due to the same reasoning as in Statement 2. This finalises the proof of Lemma 8.

Lemma 8 implies the following important result.

**Proposition 5.** Let \( \sigma, \sigma' \) be states such that \( \sigma \sim_{IP} \sigma' \). Then

- \( T_{\sigma} \sim_{IP} T_{\sigma'} \)
- \( \text{ET}(T_{\sigma}) = \text{ET}(T_{\sigma'}) \)

**Proof.** The first bullet follows directly from statement 2 of Lemma 8. The first bullet implies that \( T_{\sigma} \simeq T_{\sigma'} \), which, by Lemma 4 implies the second. 

This means that two initial states that differ only in variables that are not IPs will have the same essential trace and therefore the same timing. This is because non-IP variables are always overwritten before they are used. When specifying an input \( \sigma \), we need only to specify the IPs, since the values of non-IPs are not used. For this reason, our definition of input only concerns the initial values of IPs and two inputs are considered equal if and only if the values of all IPs are equal.
Definition 18. Let \( v \in \text{VAR}_P \) be a variable which does not occur on the left hand side in any assignment \( x:=e \) in \( P \). In other words, a variable which is used, but never assigned in \( P \). A variable with this property is referred to as an invariant variable. An invariant control variable is referred to as an invariant control variable or IV. The set of IVs in a program is denoted \( \text{IV}_P \subseteq \text{VAR}_P \).

An IV \( v \) trivially has the following property:

Lemma 9. Let \( s \) be a state and \( T_\sigma \) be a trace of a program \( P \). Then

\[
s \triangleleft T_\sigma \Rightarrow \forall v \in \text{IV}_P : s.\sigma(v) = \sigma(v)
\]

That is, an IV keeps its initial input value throughout execution of \( P \) since it is never assigned. Furthermore, for an IV \( v \) it holds that \( \text{def}_T(v) = \infty \) for all traces \( T \in \mathcal{T}S_P \). Since we have that \( \text{use}_T(v) \leq \infty \) for every IV, we have by Definition 17 that every IV is also an IP. In general: \( \text{IV}_P \subseteq \text{IP}_P \subseteq \text{VAR}_P \).

An example of an IV is the variable \( n \) in the program \( L \).

Lemma 10. Let \( T_\sigma \) be a trace, then if \( s \triangleleft T_\sigma \) and \( s' \triangleleft T_\sigma \), then \( s.\sigma \sim_{\text{IV}} s'.\sigma \).

Proof. Let \( v \) be an IV, then by Lemma 9 \( s.\sigma(v) = \sigma(v) \) and \( s'.\sigma(v) = \sigma(v) \). Since \( v \) is taken arbitrary it holds for every IV \( v \), concluding that \( s.\sigma \sim_{\text{IV}} s'.\sigma \).

A direct consequence of Lemma 10 is

\[
\text{states}(T_\sigma) \subseteq \{ s \in \mathcal{CS}_P \mid s.\sigma \in [\sigma]_{\text{IV}} \} \tag{3.4}
\]

where \( [\sigma]_{\text{IV}} \) is the class of environments that are \( \sim_{\text{IV}} \)-equal. For all states \( s \) that belong to a trace \( T_\sigma \) it holds that \( s.\sigma(v) = \sigma(v) \) for all IVs. Thus we have found a necessary condition for a given state \( s \) to belong to a trace. Next, we show that if every input parameter is an IV in a program we get a very convenient property.

Lemma 11. Let \( P \) be a program such that \( \text{IV}_P = \text{IP}_P \), then

\[
\text{states}(T_\sigma) \subseteq \{ s \in \mathcal{CS}_P \mid s.\sigma \in [\sigma]_{\text{IP}_P} \}
\]

This means that given \( \mathcal{CS}_P \), we can find all states that belong to a given trace \( T_\sigma \) by identifying the states with the property \( s.\sigma \in [\sigma]_{\text{IP}_P} \). This is a very desirable property since it gives each state an identity and can be traced back to its input. In general, a given state \( s \in \mathcal{CS}_P \) belongs to \( T_\sigma \) only if
was lost in the collecting semantics has been partially recovered.

However, programs don’t necessarily have the precondition that $\text{IV}_P = \text{IP}_P$ to enjoy this property. Luckily it is simple to instrument any program so that this is the case, without affecting its semantics.

### 3.7.1 Representing Programs

Given a program $P$ we can construct a program $P'$ such that $P'$ has the same semantics (and timing) as $P$ but with the property $\text{IV}_{P'} = \text{IP}_{P'}$, as follows.

Let $X = \text{IP}_P \setminus \text{IV}_P$. If $X = \emptyset$ then $\text{IV}_P = \text{IP}_P$ and we let $P' = P$. Otherwise, let $X = \{x_0, \ldots, x_{m-1}\}$ and let $N_A$ be a basic block associated with an assignment list $l_A$ defined as follows:

$$l_A = x_0 := x'_0, \ldots, x_{m-1} := x'_{m-1}, \epsilon$$

We refer to the set $X' = \{x'_0, \ldots, x'_{m-1}\}$ as a set of artificial input parameters. Now we let $P'$ be constructed from $P$ as shown in Figure 3.3.

![Figure 3.3](image)

Figure 3.3: The difference between the original program $P$ and the altered program $P'$ is that the basic block $N_A$ has been inserted. The cloud $B$ represents the rest of the program $P$.

This means that $\text{VAR}_{P'} = \text{VAR}_P \cup X'$. Since $P'$ should have the same timing as $P$ we consider the execution time of basic block $N_A$ to be zero, and therefore let $\ell_{P'}(\langle N_A, n_1 \rangle) = 0$, and we let the initial program point of $P'$ has the same timing as $P$, $\ell(\langle n_0, N_A \rangle_{P'}) = \ell(\langle n_0, n_1 \rangle_P)$. The idea here is that the
actual program \( P \) should not be changed physically, we only change how we model \( P \), though the program \( P' \). In other words, no actual instrumentation of the program to be analysed has to be done, only the representation of it. That the timing of \( P \) and \( P' \) is equal is stated in the following lemma:

**Lemma 12.** Let \( \sigma \) be an input to \( P \). Then we can construct an input \( \sigma' \) to \( P' \) such that \( \sigma'(v) = \sigma(v) \) for all \( v \in \text{VAR}_P \) and \( \sigma'(x') = \sigma(x) \) for all variables \( x' \in X' \). Then,

\[
ET_P(T_\sigma) = ET_{P'}(T_{\sigma'})
\]

Since \( P \) and \( P' \) have equal execution time, left to show is that \( \text{IV}_{P'} = \text{IP}_{P'} \). Since we know that \( \text{IV}_{P'} \subseteq \text{IP}_{P'} \), all we have to show is that \( \text{IP}_{P'} \subseteq \text{IV}_{P'} \). That is, that each input to \( P' \) is an invariant in \( P' \). We start by the following lemma:

**Lemma 13.**

\[
\text{IP}_{P'} = \text{IV}_P \cup X'
\]

**Proof.** \( \subseteq \)

All variables in \( X' \) are naturally input parameters since they are guaranteed to be used in \( N_A \) before they are defined (since they are never defined). All invariants of \( P \) are invariants of \( P' \) since variables in \( \text{IV}_P \) does not occur in \( N_A \) so we have \( \text{IV}_P \subseteq \text{IV}_{P'} \subseteq \text{IP}_{P'} \).

Let \( x \in \text{IP}_{P'} \) and assume for contradiction that \( x \notin \text{IV}_P, x \notin X' \) which means that \( x \notin X \). Since \( x \in X, x \) is assigned before it is used in \( N_A \) and can therefore not be an input parameter so our assumption must be false. \( \square \)

**Proposition 6.** Let \( P \) be a program and construct \( P' \) as described above. Then

\[
\text{IV}_{P'} = \text{IP}_{P'}
\]

**Proof.** By Lemma 13 we have that \( \text{IP}_{P'} = \text{IV}_P \cup X' \). It is now sufficient to show that \( X' \subseteq \text{IV}_{P'} \) and \( \text{IV}_P \text{IV}_{P'} \). In the first case, all variables in \( X' \) are invariants since they are used only in \( N_A \) and never defined. In the second case, every invariant in \( P \) is still an invariant in \( P' \) since any invariant in \( P \) are guaranteed to not be in \( X \) and is therefore not affected by \( N_A \) which is the only difference between \( P \) and \( P' \). \( \square \)

By the outlined procedure, we can take any program \( P \) and construct a program \( P' \) with the same semantics and timing as \( P \) such that \( \text{IV}_{P'} = \text{IP}_{P'} \),
and by this property (see Lemma 11) we can determine for any state \( s \in CS_{P'} \)
if \( s \) is part of the trace \( T_\sigma \) or not by:

\[
(\forall v \in IV_{P'} : s.\sigma(v) = \sigma) \Rightarrow s \in T_\sigma
\]

In other words, let

\[
S_\sigma = \{ s \in CS_{P'} | \forall v \in IV_{P'} : s.\sigma(v) = \sigma(v) \}
\]

then the following two statements hold:

\[
\text{STATE}(T_\sigma) \subseteq S_\sigma
\]

\[
\text{ET}_{P'}(T_\sigma) \leq \sum_{s' \in S_\sigma} \ell(s', q)
\]

Finally, we convert this into a counting problem by:

\[
\text{ET}_{P'}(T_\sigma) \leq \sum_{q \in Q_{P'}} C(S_\sigma, q) \ell(q)
\]

(3.5)

where \( C \) is the Census function (3.3), defined on page 37.

Thus, we have found a safe approximation of the execution time of a program \( P \) with a given input \( \sigma \) by using the collecting semantics. The next section summarises the process.

### 3.8 Summary

In this section, a summary of the foundation of our method is presented. Given a program \( P \), we start by constructing \( \text{SLICE}(P) \) (Definition 15), which removes most non-control variables. After this, we construct the program \( P' \) (Section 3.7.1) which has the properties that all input variables are invariant variables. This introduces an artificial variable \( x' \) for each non-invariant input variable \( x \) of \( P \), for which the initial value will be the same as for \( x \). By using the collecting semantics (Section 3.5) for the resulting program \( P' \) together with a low-level analysis, the WCET of running the program \( P \) with a given input \( \sigma \) can be bound by (3.5). This equation comes from counting the upper bound of the number of states that can be visited during a given trace.

The final thing we have to do to make this approach practical is to find a safe approximation of the collecting semantics. This is what the next chapter is about.
Chapter 4

Abstract Interpretation and the Polyhedral Domain

As explained, our method is based on computing the set of states that can visit each program point via the collecting semantics. Unfortunately, the collecting semantics is not computable in the general case. Fortunately, there are efficient and computable methods that approximate the collecting semantics.

Abstract Interpretation [34] is a well-known technique which can be used to approximate program semantics. Static WCET analysis aims to derive safe estimates of the WCET, meaning that any approximation must be done conservatively. Abstract interpretation can approximate the collecting semantics by a superset of $CS_P$. That is, all states that may be visited during an execution will be contained in the approximation. Another way to say this is to say that the approximation is sound. However, it is possible that the approximation introduces states that cannot possibly be visited during any execution. This means that the result of abstract interpretation will not be as precise as the collecting semantics, but will be computable as compensation.

4.1 Domain Theory

This section introduces some elementary domain theory in order to be able to properly define abstract interpretation. The results here are well-known, so no proofs of claims are given here, but can generally be found in [87, 1].
4.1.1 Lattices

Definition 19. (Poset)
A poset (or partially ordered set) \(\langle L, \sqsubseteq \rangle\) is a set \(L\) and a relation such that \(\sqsubseteq\) is

- reflexive: \(\forall l \in L : l \sqsubseteq l\)
- anti-symmetric: \(\forall l, m \in L : l \sqsubseteq m \land m \sqsubseteq L l \Rightarrow l = m\)
- and transitive: \(\forall k, l, m \in L : k \sqsubseteq l \land l \sqsubseteq L m \Rightarrow k \sqsubseteq m\)

Definition 20. (Upper and lower bounds)
Let \(\langle L, \sqsubseteq \rangle\) be a poset and let \(M \subseteq L\). An element \(u \in L\) is an upper bound of \(M\) if it holds that \(m \sqsubseteq u\) for all \(m \in M\). Conversely, an element \(l \in L\) is considered to be a lower bound if \(l \sqsubseteq m\) for all \(m \in M\).

Definition 21. (Supremum and infimum)
Let \(\langle L, \sqsubseteq \rangle\) be a poset and let \(M \subseteq L\). If \(M\) has upper bounds and there exists an upper bound \(u_0\) such that for all other upper bounds \(u \in L\) it holds that \(u_0 \sqsubseteq u\), then \(u_0\) is the supremum of \(M\) and is denoted \(\sqcup M\). Similarly, if \(M\) has lower bounds and there exists a lower bound \(l_0\) such that for all other lower bounds \(l \in L\) it holds that \(l \sqsubseteq l_0\), then \(l_0\) is the infimum of \(M\) and is denoted \(\sqcap M\). The supremum or infimum of a subset \(M \subseteq L\) is always unique if it exists.

Note that a subset of a poset does not necessarily have upper and lower bounds, and if they do, they don’t necessarily have an infimum or supremum.

Definition 22. (Complete lattice)
A poset \(L\) such that for all subsets \(M \subseteq L\), \(\sqcup M\) and \(\sqcap M\) exist, is called a complete lattice. A complete lattice can therefore be described by a tuple \(L = \langle L, \sqcup_L, \sqcap_L \rangle\).

For a complete lattice \(L\), supremum and infimum are defined for each subset, making them an operation on subsets of \(L\). It is very common to use the supremum and infimum operation on subsets with two elements, so we will generally use the notation \(l \sqcup l' = \sqcup\{l, l'\}\) and \(l \sqcap l' = \sqcap\{l, l'\}\).

In domain theory the element \(\sqcap L = \bot_L\) is called the bottom element of \(L\) and \(\sqcup L = \top_L\) is called the top element.

Given \(\sqcup_L, \sqcap_L\) the lattice order \(\sqsubseteq_L\) is induced by

- \(l \sqsubseteq_L l' \iff l = l \sqcap_L l'\)
- \(l \sqsubseteq_L l' \iff l' = l \sqcup_L l'\)
Examples

There are two general examples of complete lattices that are used often throughout this thesis.
Let $A$ be a countable set. Then $\langle P(A), \subseteq \rangle$ is a complete lattice with $\bot_{P(A)} = \emptyset$, $\top_{P(A)} = A$.
Let $C = A \rightarrow L$ where $A$ is a set and $L$ is a complete lattice. Then $\langle C, \subseteq_C \rangle$ is a complete lattice, with

$$f \subseteq_C g \iff \forall a \in A : f(a) \subseteq_L g(a)$$

and

$$\bot_C = \lambda l. \bot_L, \top_C = \lambda l. \top_L$$

4.1.2 Fixed Point Theory

Definition 23. (Monotone functions) Let $\langle L, \subseteq_L \rangle$ and $\langle L', \subseteq_{L'} \rangle$ be posets and let $f : L \rightarrow L'$ be a function. Then $f$ is a monotone function iff

$$l \subseteq_L m \Rightarrow f(l) \subseteq_{L'} f(m)$$

A well-known and important result is that any monotone self-map over a complete lattice has a least fixed point.

Proposition 7. (Tarski [110])

Let $L$ be a complete lattice and $f : L \rightarrow L$ be a monotone function. Then the set $\text{fix} f = \{ l \in L \mid f(l) = l \}$ is a complete lattice.

A consequence of Proposition 7 is that since $\text{fix} f$ is a complete lattice, $\cap(\text{fix} f)$ is the least element in this lattice, and consequently the least fixed point of $f$. In order to compute this fixed point, a few more definitions are needed.

Definition 24. (Chains)

Let $L$ be a complete lattice, then $M \subseteq L$ is a chain if it is non-empty and for all elements $m, m' \in M$ either $m \subseteq m'$ or $m \supseteq m'$.

In other words, a chain is a subset of a complete lattice where the elements are completely ordered. Thus, chains can be described as decreasing or increasing sequences (e.g., $\ldots \subseteq m_1 \subseteq m_2 \subseteq \ldots$).

Definition 25. (Continuity)

A monotone function $f : L \rightarrow L$ is Scott-continuous iff for every chain $M \subseteq L$, it holds that $f(\sqcup M) = \sqcup \{ f(m) \mid m \in M \}$. 


The following theorem shows how the least fixed point of a continuous function can be computed. The result is due to Kleene, but the theorem is not presented in full generality here.

**Proposition 8.** (Kleene [70])

Let $L$ be a complete lattice and $f : L \rightarrow L$ a Scott-continuous function, then

$$\text{lfp } f = \bigsqcup \{ f^n(\bot) \mid n \in \mathbb{N} \}$$

This implies that the least fixed point can be computed by iteratively computing the sequence $f^0(\bot), f^1(\bot), \ldots$ until the sequence stabilises. For this to be practical, the sequence has to stabilise within a finite number of steps, which in general cannot be guaranteed.

### 4.2 Introduction to Abstract Interpretation

Abstract interpretation makes approximations based on a selected abstract domain. The abstract domain specifies how, and to which extent the sets are approximated. A great variety of abstract domains have been proposed in literature and the choice of domain offers a trade-off between precision and computational complexity. The following sections will introduce the theory of abstract interpretation.

#### 4.2.1 Abstraction

The idea of abstract interpretation is to have a certain relationship between two complete lattices. One lattice is referred to as the concrete domain $L$ and the other as the abstract domain $M$. Intuitively, the abstract domain is a simpler version of the concrete domain, which preserves all aspects of a selected property. This is done by having a Galois connection $\langle L, \alpha, \gamma, M \rangle$ between the two lattices, consisting of an abstraction function $\alpha : L \rightarrow M$ and a concretisation function $\gamma : M \rightarrow L$. The relationship is depicted in Figure 4.1.

**Definition 26.** A Galois connection $\langle L, \alpha, \gamma, M \rangle$ is a tuple consisting of two complete lattices $L, M$ and two monotone functions $\langle \alpha, \gamma \rangle \in (L \rightarrow M) \times (M \rightarrow L)$, such that

$$\alpha \circ \gamma \subseteq_M \lambda m. m \text{ and } \gamma \circ \alpha \supseteq_L \lambda l. l$$

If it also holds that $\alpha \circ \gamma \supseteq_M \lambda m. m$, then $\langle L, \alpha, \gamma, M \rangle$ is called a Galois insertion.
puting the sequence $f$ of lattices, consisting of an abstraction $L$, $M$.

Complete lattices in general cannot be guaranteed. Thus, the sequence has to stabilise within a finite number of steps, which can be practical, given a Galois connection. This is done by having a Galois connection $\langle L, \alpha, \gamma, M \rangle$ between the two concrete domain versions of the concrete domain, which preserves all aspects of a selected property. This is done by having a Galois connection $\langle L, \alpha, \gamma, M \rangle$ between the two concrete domain versions of the concrete domain, which preserves all aspects of a selected property.

The idea of abstract interpretation is to have a certain relationship between two domains. Abstract interpretation makes approximations based on a selected abstract domain. As an example, consider the set $\{\bot, -, 0, +, \top\}$ with an ordering as shown in Figure 4.2. We then form the following Galois connection:

$$\begin{align*}
\gamma_{\text{Sign}}(\bot) &= \emptyset & \alpha_{\text{Sign}}(\emptyset) &= \bot \\
\gamma_{\text{Sign}}(\neg) &= \mathbb{Z}_- & \alpha_{\text{Sign}}(A) &= - \iff \forall a \in A : a < 0 \\
\gamma_{\text{Sign}}(0) &= \{0\} & \alpha_{\text{Sign}}(\{0\}) &= 0 \\
\gamma_{\text{Sign}}(+) &= \mathbb{Z}_+ & \alpha_{\text{Sign}}(A) &= + \iff \forall a \in A : a > 0 \\
\gamma_{\text{Sign}}(\top) &= \mathbb{Z} & \alpha_{\text{Sign}}(A) &= \top \text{ in all other cases}
\end{align*}$$

Note that Definition 26 holds for $\langle \alpha_{\text{Sign}}, \gamma_{\text{Sign}} \rangle$. The intuition behind this is that the set of integers are abstracted by sign by this Galois connection. The $\alpha_{\text{Sign}}$ function abstracts a set by mapping the set into its minimum representation in the abstract domain. As an example, consider the set $\{1, 2, 3\} \in \mathcal{P}(\mathbb{Z})$. The abstract version of this element is obtained by $\alpha_{\text{Sign}}(\{1, 2, 3\}) = +$. The set $\{1, 2, 3\}$ is represented by a “+” in the abstract domain, meaning that the set
an example, consider the Galois connection
\[ \langle P \hat{\alpha}, \hat{\gamma} \rangle. \]
This relation is depicted in Figure 4.3. The idea here is that
interpretation of the semantics of \( f \) gives a correct
approximating. Now, we define the abstract multiplication
on our concrete domain
\[ \langle P \hat{\alpha}, \hat{\gamma} \rangle. \]
instance,
\[ \langle P \hat{\alpha}, \hat{\gamma} \rangle. \]
computable, but then naturally with some lost precision. Let
\( \langle \hat{\gamma} \rangle \) be a
Galois-connection and let \( f : L \rightarrow L \) be a monotone function over the concrete
lattice \( L \). Then we say that \( \hat{f} : M \rightarrow M \) is approximating \( f \) or that \( \hat{f} \) is an
abstract version of \( f \), iff
\[ \forall l \in L : f(l) \sqsubseteq \gamma \circ \hat{f} \circ \alpha(l). \]
This relation is depicted in Figure 4.3. The idea here is that \( \hat{f} \) gives a correct
interpretation of the semantics of \( f \), but with possible loss of information. As
an example, consider the Galois connection
\( \langle \mathcal{P}(\mathbb{Z}), \alpha_{\text{Sign}}, \gamma_{\text{Sign}}, \text{Sign} \rangle \) from
Section 4.2.1 again. First, consider the lifted multiplication operation \( \cdot_{\mathcal{P}} : \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \rightarrow \mathcal{P}(\mathbb{Z}) \) defined as
\[ A \cdot_{\mathcal{P}} B = \{ a \cdot b \mid a \in A \land b \in B \}. \]
This operation is simply normal multiplication defined over sets of integers, for
instance, \( \{1, 2, 3\} \cdot_{\mathcal{P}} \{-1, -2\} = \{-1, -2, -3, -4, -6\} \). This is an operation
on our concrete domain \( \mathcal{P}(\mathbb{Z}) \) and is the operation which we are interested in
approximating. Now, we define the abstract multiplication
\( \hat{\cdot} : M \rightarrow M \) as
4.2 Introduction to Abstract Interpretation

\[
\begin{align*}
L & \xrightarrow{f} L \\
\alpha & \downarrow \subseteq \gamma \\
M & \xrightarrow{\hat{f}} M
\end{align*}
\]

Figure 4.3: Relation between concrete and abstract functions

follows

\[
\begin{align*}
+\hat{\cdot} + &= + \\
-\hat{\cdot} - &= + \\
-\hat{\cdot} + &= - \\
0\hat{\cdot} a &= 0 \\
\top\hat{\cdot} a &= \top \\
\bot\hat{\cdot} b &= \bot
\end{align*}
\]

where \(a\) is any non-bottom element and \(b\) is any element. This is a correct definition of an abstract operation, which should be easy to verify. As an example, we see that

\[
\{1, 2, 3\} \cdot_P \{−1, −2\} = \{−1, −2, −3, −4, −6\} \subseteq \\
\gamma(\alpha\{1, 2, 3\}) \hat{\cdot} \alpha\{−1, −2\}) = \gamma(−\hat{\cdot} −) = \\
\gamma(−) = \mathbb{Z}_-
\]

**Lifted Operations**

When abstract interpretation is applied in static analysis, the abstract functions approximate functions available in the programming language semantics. Our program model is restricted to integer valued variables, so we will be interested in approximating functions over integers (such as addition, subtraction, multiplication and addition), i.e., functions of the type \(f : \mathbb{Z}^n \rightarrow \mathbb{Z}\). However, the concrete domain used in abstract interpretation operates over sets of integers rather than integers themselves. Thus, for any \(n\)-ary operation, \(f : \mathbb{Z}^n \rightarrow \mathbb{Z}\), it is possible to define a *lifted* version \(f^P : \mathcal{P}(\mathbb{Z})^n \rightarrow \mathcal{P}(\mathbb{Z})\) defined as

\[
f^P(X_0, ..., X_{n-1}) = \{f(x_0, ..., x_{n-1}) \mid x_i \in X_i \text{ for all } 0 \leq i < n\}
\]
In practice, when operations over the integers are used, the concrete domain will be \(\mathcal{P}(\mathbb{Z})\), correspondingly, it is the lifted versions of the operations that will be approximated. When clear from context, we will use write \(f\) for \(f^P\). Note that lifted functions are always monotone.

**Fixed Points of Abstract Functions**

The reason to define abstract functions is to iterate in the abstract domain rather in the concrete one. A basic result from abstract interpretation is that for any monotone functions \(f : L \to L\) and \(\hat{f} : M \to M\) such that \(\hat{f}\) is approximating \(f\), it holds that

\[
lfp f \subseteq \gamma(lfp \hat{f}).
\]

This means that the least fixed point of the abstract function is a safe approximation of the least fixed point of the concrete function.

**4.2.3 Widening and Narrowing**

To find the least fixed point of a monotone operator \(f : L \to L\) over a lattice \(L\), two cumbersome requirements are imposed on \(L\) and \(f\):

- \(f\) must be Scott-continuous.
- The sequence \(\bot \subseteq f(\bot) \subseteq f(f(\bot)) \subseteq \ldots\) must stabilise after a finite number of steps.

By “stabilises after a finite number of steps”, we mean that there exist a \(k \in \mathbb{N}\) such that for the increasing sequence \((f^n(\bot))_{n \in \mathbb{N}} = \bot \subseteq f(\bot) \subseteq f(f(\bot)) \subseteq \ldots\) it holds that \(f^k(\bot) = f(f^k(\bot))\). In practice, these two requirements are too restrictive. For instance, it would disable the possibility of having infinite ascending chains in the abstract domain. Fortunately, it is possible to approximate the fixed point by introducing a so-called widening operator [34].

**Definition 27.** A widening operator \(\nabla : L \times L \to L\) is an operator over a lattice fulfilling \(\forall l, l' : l, l' \subseteq (l \nabla l')\) and for any increasing sequence \(l_0 \subseteq l_1 \subseteq \ldots\), the increasing sequence \(l_0 \nabla l_1 \subseteq l_0 \nabla l_1 \nabla l_2 \subseteq \ldots\) eventually stabilises.

Thus if the sequence \((f^n(\bot))_{n \in \mathbb{N}} = \bot \subseteq f(\bot) \subseteq f(f(\bot)) \subseteq \ldots\) is replaced by the sequence \((f^n_\nabla)_{n \in \mathbb{N}} = \bot \subseteq f(\bot) \nabla \bot \subseteq f(f(\bot)) \nabla f(\bot) \nabla \bot \subseteq \ldots\), then the sequence will eventually stabilise at \(lfp(f_\nabla)\) and by definition of
the widening operation, it will hold that \( \text{lfp}(f_{\gamma}) \supseteq \text{lfp}(f) \). Thus, an approximation of the fixed point can be found in a finite number of steps. This to the cost of possible lost precision; the widening operation is greater or equal to the supremum of its arguments. However, the situation can be improved by having the dual concept of a narrowing operator.

**Definition 28.** A narrowing operator \( \Delta : \mathcal{L} \times \mathcal{L} \to \mathcal{L} \) is an operator over a lattice fulfilling \( \forall l, l' : l \subseteq l' \to l \subseteq (l \Delta l') \subseteq l' \) and for any decreasing sequence \( l_0 \supseteq l_1 \supseteq \ldots \), the decreasing sequence \( l_0 \supseteq l_0 \Delta l_1 \supseteq l_0 \Delta l_1 \Delta l_2 \supseteq \ldots \) eventually stabilises.

Note that the sequence \( (f^n_{\gamma})_{n \in \mathbb{N}} \) is stable at \( \text{lfp}(f_{\gamma}) \), so the sequence \( \text{lfp}(f_{\gamma}) = f(\text{lfp}(f_{\gamma})) = f(f(\text{lfp}(f_{\gamma}))) = \ldots \) is a stable sequence. Furthermore, the sequence \( \text{lfp}(f_{\gamma}) \supseteq f(\text{lfp}(f_{\gamma})) \Delta \text{lfp}(f_{\gamma}) \supseteq \ldots \) eventually stabilises (by definition of the narrowing operator) at \( \text{lfp}(f_{\gamma}^{\Delta}) \) and any element in the sequence is greater or equal to \( \text{lfp}(f) \) as shown in [87]. Thus,

\[
\text{lfp}(f) \subseteq \text{lfp}(f_{\gamma}^{\Delta}) \subseteq \text{lfp}(f_{\gamma}).
\]

In summary, to find a good approximation of the least fixed point of \( f \) in finite time without the requirements that \( f \) is Scott-continuous or that the increasing sequence \( (f^n)_{n \in \mathbb{N}} \) should stabilise:

- First compute \( \text{lfp}(f_{\gamma}) \)
- Use \( \text{lfp}(f_{\gamma}) \) as starting point and compute \( \text{lfp}(f_{\gamma}^{\Delta}) \).

### 4.3 The Polyhedral Domain

One of the most widely used abstract domains is the polyhedral domain [35]. The polyhedral domain is popular because it can efficiently capture linear relationships between program variables as well as providing upper and lower bounds for the variables. The domain have been used in several applications [17, 56, 14, 25], and due to its popularity there exist a number of open source implementations of the domain [86, 91, 92, 8].

Our implementations and evaluations of our parametric WCET method use the polyhedral domain due to its accuracy and the fact that it maintains relationship between variables, which is essential to our method. The rest of this chapter outlines the polyhedral abstract domain in detail.
4.3.1 Convex Polyhedra

A half-space \( H \subseteq \mathbb{R}^n \) is a set on the form:

\[
H = \left\{ \langle x_0, \ldots, x_{n-1} \rangle \in \mathbb{R}^n \middle| \sum_{i=0}^{n-1} a_i x_i \leq c \right\}
\]

where \( c, a_i \in \mathbb{R} \) for all \( 0 \leq i < n \). The intersection of a finite number of half-spaces is called a \textit{convex polyhedron}\(^1\). We will denote the set of convex polyhedra in \( n \)-space \( \text{Poly}^n \), or just \( \text{Poly} \) when the dimension is not important.

4.3.2 Representation

Since a set of points in \( \mathbb{R}^n \) is not a very convenient or compact way of representing convex polyhedra, we will define two different representations of convex polyhedra. As it turns out, both of these representations are needed to efficiently represent convex polyhedra as an abstract domain.

The Constraint Representation

According to our definition of convex polyhedra, a finite set of half-spaces effectively represents a convex polyhedron. Thus, we will define a \textit{constraint} \( h \) as an element with the syntactical form

\[
h = \sum_{i=0}^{n-1} a_i x_i \leq c
\]

a finite set \( H = \{h_0, \ldots, h_{k-1}\} \) of constraints then represents a convex polyhedron via the function:

\[
\mathbb{P}_\mathcal{H}(H) = \bigcap_{(\sum_{i=0}^{n-1} a_i x_i \leq c) \in H} \left\{ \langle x_0, \ldots, x_{n-1} \rangle \middle| \sum_{i=0}^{n-1} a_i x_i \leq c \right\}
\]

A finite set of constraints is called a \( \mathcal{H} \)-polyhedron and the set of all such polyhedra is denoted \( \mathcal{H} \). A \( \mathcal{H} \)-polyhedron \( H_0 \) such that \( \mathbb{P}_\mathcal{H}(H_0) = \mathcal{P} \) is \textit{minimal} if \( \forall H \in \mathcal{H} : \mathbb{P}_\mathcal{H}(H) = \mathcal{P} \rightarrow |H_0| \leq |H| \).

\(^1\)or \textit{convex polytope} which would be a more accurate term, but in this context they are conventionally called polyhedra.
The Frame Representation

Let $F = (V, R, D)$ where $V = \{v_0, ..., v_{n-1}\} \subseteq \mathbb{R}^n$ is a finite set of vertices, $R = \{r_0, ..., r_{m-1}\} \subseteq \mathbb{R}^n$ is a finite set of rays and $D = \{d_0, ..., d_{p-1}\}$ is a finite set of lines. Then $F$ is called a frame and represents a convex polyhedron by the mapping

$$
\mathbb{P}_\mathcal{F}(F) = \left\{ \sum_{i=0}^{n-1} \lambda_i v_i + \sum_{j=0}^{m-1} \mu_j r_j + \sum_{k=0}^{p-1} \nu_k d_k \mid \sum_{i=0}^{n-1} \lambda_i = 1 \right\}
$$

where

$$
\lambda_{i\in[0,n-1]} \in \mathbb{R}_+ \land \mu_{j\in[0,m-1]} \in \mathbb{R}_+ \land \nu_{k\in[0,p-1]} \in \mathbb{R}
$$

A bounded polyhedron is formed by the convex hull of the vertices and has no rays or lines. A ray represents unboundedness in some direction, meaning that from any point in the polyhedron, a ray can be “followed” indefinitely. A line represents a ray that can be followed both in positive and negative direction. The class of frames is denoted $\mathcal{F}$, and a frame may sometimes be called an $\mathcal{F}$-polyhedron.

Given a frame $F$ or a polyhedron $\mathcal{P}$ we write

$$
\mathbb{P}_\mathcal{F}(F) = \mathcal{P}
$$

to denote that polyhedron $\mathcal{P}$ is represented by the frame $F$. In this way we distinguish a polyhedron (set of points) from its representation (a frame). Or we can write:

$$
\mathbb{P}_\mathcal{H}(H) = \mathcal{P}
$$

to denote that the polyhedron $\mathcal{P}$ is represented by a set of constraints $H$.

4.3.3 The Lattice of Convex Polyhedra

The set of polyhedra forms a complete lattice $\langle \text{Poly}, \sqcap, \sqcup \rangle$ where $\sqcap_{\text{Poly}}(M) = \bigcap M$ and $\sqcup_{\text{Poly}}(M)$ is the smallest (wrt. inclusion) convex polyhedron $\mathcal{P}$ such that $\bigcup M \subseteq \mathcal{P}$. These lattice operations can be efficiently computed using the representations of convex polyhedra. These operation are presented here without proof of their correctness (for proofs and detailed information see [56]).

In the following examples, let

$$
\mathcal{P}_1 = \mathbb{P}_\mathcal{H}(H_1) = \mathbb{P}_\mathcal{F}(F_1), F_1 = \langle V_1, R_1, D_1 \rangle
$$

$$
\mathcal{P}_2 = \mathbb{P}_\mathcal{H}(H_2) = \mathbb{P}_\mathcal{F}(F_2), F_2 = \langle V_2, R_2, D_2 \rangle
$$
We can assume that $H_1, H_2, F_1, F_2$ are minimal representations of $P_1, P_2$. Then we define

$$
P_1 \cup P_2 = \mathbb{P}_F(\{V_1 \cup V_2, R_1 \cup R_2, D_1 \cup D_2\})$$
$$
P_1 \cap P_2 = \mathbb{P}_H(H_1 \cup H_2)$$

This means that given two polyhedra represented by sets of constraints, the constraint set of the infimum of these two polyhedra can be computed. If the frames of the two polyhedra are given, then the frame of the supremum of the polyhedra can be computed. Different operations need different representations.

Furthermore, we define the standard widening operation $\nabla$ over convex polyhedra (as defined in [9], alternative definitions also exist in [9, 79]):

$$
P_1 \nabla P_2 = \mathbb{P}_H(A \cup B)$$

where $A$ is the set of constraints in $P_1$ that are satisfied by $P_2$ and $B$ is the set of constraints in $P_2$ that have an equivalent constraint in $P_1$. Or formally:

$$
A = \{h \in H_1 \mid \mathbb{P}_H(h) \subseteq P_2\}$$
$$
B = \{h \in H_2 \mid \exists h' \in H_1. \mathbb{P}_H((H_1 \setminus \{h'\}) \cup \{h\}) = P_1\}$$

Furthermore, we define an operation on polyhedra that performs a linear transformation in a dimension. The polyhedron $\bar{P}[x_j \leftarrow \sum_{i=0}^{n-1} a_i x_i + c]$ is the result of taking polyhedron $P$ and applying a linear transformation in dimension $j$. It is defined as follows:

$$
P_1[x_j \leftarrow \sum_{i=0}^{n-1} a_i x_i + c] = \mathbb{P}_F(\{V'_1, R'_1, D'_1\})$$
where

\[
V'_1 = \left\{ \langle v_0, \ldots, v'_j, \ldots, v_{n-1} \rangle \mid \langle v_0, \ldots, v_j, \ldots, v_{n-1} \rangle \in V_1 \land v'_j = \sum_{i=0}^{n-1} a_i v_i + c \right\}
\]

\[
R'_1 = \left\{ \langle r_0, \ldots, r'_j, \ldots, r_{m-1} \rangle \mid \langle r_0, \ldots, r_j, \ldots, r_{m-1} \rangle \in R_1 \land r'_j = \sum_{i=0}^{m-1} a_i r_i \right\}
\]

\[
D'_1 = \left\{ \langle d_0, \ldots, d'_j, \ldots, d_{p-1} \rangle \mid \langle d_0, \ldots, d_j, \ldots, d_{p-1} \rangle \in D_1 \land d'_j = \sum_{i=0}^{p-1} a_i d_i \right\}
\]

It is also possible to model that some non-linear transformation is being done in dimension \(j\). This is modelled by:

\[
\mathcal{P}_1[x_j \leftarrow ?] = \mathbb{P}_\mathcal{F}(\langle V_1, R_1, D_1 \cup \{e^j\} \rangle)
\]

where \(e^j_i = 0\) for all \(i \in [0, n-1]\) such that \(i \neq x_j\) and \(e^j_i = 1\). This means that if a non-linear transformation is being done, a line is added in that dimension, meaning that all information about this dimension is discarded.

As seen, given the \(\mathcal{H}\)-representation and the \(\mathcal{F}\)-representation (preferably minimal), we can simply and efficiently compute the various operations on polyhedra. However, the operations \(\cap\) and \(\nabla\) compute a constraint representation while the operations \(\cup\) and \(\mathcal{P}[x \leftarrow a]\) compute a frame representation. This means that both representations must be available to be able to compute all operations. In order to have both the frame and the constraint representation available for a given polyhedron there must be a way to make a conversation between these two representations. Such conversions are costly but possible, for details see [56]. In a practical implementation conversion should happen only when necessary (i.e., when an operation that requires a representation that is not currently available).

### 4.3.4 Convex Polyhedra as Abstract Domain

Let the concrete domain be \(\mathbb{R}^n\) and the set of \(n\)-dimensional convex polyhedra \(\text{Poly}^n\) be the abstract domain. The abstract domain is then formed by a Galois-connection between these two domains, that is \(\langle \text{Poly}^n, \alpha_{\text{CP}}, \gamma_{\text{CP}}, \mathbb{R}^n \rangle\).

\(^2\text{We use the subscript CP do denote the classical polyhedra domain. This is to distinguish it from other polyhedral domains presented in this thesis.}\)
the concrete domain is $\mathbb{R}^n$ we can use:

$$\gamma_{CP} = \lambda P. P$$

since $P$ is defined as a set of points, and where $P$ is given by either $P = \mathbb{P}_F(F)$ or $P = \mathbb{P}_H(H)$. For the abstraction function, $\alpha_{CP}(S)$ where $S \subseteq \mathbb{R}^n$, is defined as the smallest (w.r.t.inclusion) convex polyhedron containing $S$. If $S$ is finite, we can easily represent this with a frame. The convex hull of a finite set of points is the smallest polyhedron containing it, so we can set

$$\alpha_{CP} = \lambda S. \mathbb{P}_F((S, \emptyset, \emptyset))$$

### 4.4 Approximating Program Semantics

This section explains how abstract interpretation can be used to approximate program semantics. Specifically, we explain how to approximate the semantics of the program model given in Chapter 3, and in order to be practical, we focus on how to do this with the polyhedral domain. More general information about how to approximate program semantics in general using abstract interpretation can be found in for instance in [34, 87, 21].

Essentially we are interested in obtaining an approximation of $CS_P$. I.e., we want to compute a set $\widehat{CS}_P$ such that

$$CS_P \subseteq \widehat{CS}_P$$

We can then use (3.5) to compute a safe parametric WCET of $P$. By abstract interpretation, the set $\widehat{CS}_P$ is computable, in contrast to $CS_P$, so this is what makes our method practical and possible.

Assume that there is a Galois connection $\langle \mathcal{P}(\text{ENV}), \alpha_{\text{ENV}}, \gamma_{\text{ENV}}, \overset{\text{ENV}}{\text{ENV}} \rangle$ and that $\overset{\text{ENV}}{\text{ENV}}$ is a lattice of abstract environments. Since $\mathcal{P}(\text{ENV})$ and $\overset{\text{ENV}}{\text{ENV}}$ are both complete lattices, we can see from Section 4.1.1 that $\mathcal{Q} \rightarrow \mathcal{P}(\text{ENV})$ and $\mathcal{Q} \rightarrow \overset{\text{ENV}}{\text{ENV}}$ are also both complete lattices.

Moreover, the lattice $\langle \mathcal{Q} \rightarrow \mathcal{P}(\text{ENV}), \subseteq \rangle$ has a bijective, order-preserving map to $\langle \mathcal{P}(\text{STATE}), \subseteq \rangle$, meaning that for any sets of states $S, S'$ we have that

$$S \subseteq S' \iff \phi(S) \subseteq_\phi \phi(S')$$

where $\phi$ is defined in Section 3.4.1 on page 35 and $\subseteq_\phi$ is defined as in the example section of Section 4.1.1 on page 53.
We can then form the Galois connection:

\[ \left\langle \mathcal{Q} \rightarrow \mathcal{P}(\text{ENV}), \alpha_{\text{STATE}}, \gamma_{\text{STATE}}, \mathcal{Q} \rightarrow \text{ENV} \right\rangle \]

by

\[ \alpha_{\text{STATE}}(S) = \lambda q.\alpha_{\text{ENV}}(S(q)) \]
\[ \gamma_{\text{STATE}}(\hat{S}) = \lambda q.\gamma_{\text{ENV}}(\hat{S}(q)) \]

Thus, from a Galois connection between the set of environments and abstract environments \( \text{ENV} \) we can derive a Galois connection between the set of states and abstract states. This means that in practice the abstract domain will be abstract states, but derived from a domain of abstract environments.

### 4.4.1 The Polyhedral Domain as Abstract Environment

The classical polyhedral domain is defined as a Galois connection between \( \mathcal{P}(\mathbb{R}^n) \) and \( \text{Poly}^n \). To approximate program semantics using the polyhedral domain, we need to change the concrete domain to \( \mathcal{P}(\text{ENV}) \). The way to do this is to interpret a set of points in \( n \)-space as a set of environments.

A point \( \langle s_0, \ldots, s_{n-1} \rangle \in \mathbb{Z}^n \) (i.e. a point with integer coordinates) can be interpreted as an environment in the following way:

\[ \text{ENV}_{\mathbb{Z}^n}(s) = \lambda v_j.s_j \]

That is, each dimension is treated as a variable, and a point in \( n \)-space then represents values for \( n \) variables. In this way, the set \( \mathbb{R}^n \) can be used as concrete domain. A set of points \( S \subseteq \mathbb{R}^n \) can be interpreted as a set of environments:

\[ \text{ENV}_{\mathbb{R}^n}(S) = \{ \lambda v_j.s_j \mid s \in S \cap \mathbb{Z}^n \} \]

That is, given a set of points \( S \), our program model interprets all integer points in this set as a possible environment. A polyhedron \( \mathcal{P} \) contains a (possibly empty) set of integer points, namely \( \mathcal{P} \cap \mathbb{Z}^n \). All of these points are contained in the polyhedron and each of them represents a possible environment. Thus, given a polyhedron \( \mathcal{P} \), the function \( \text{ENV}_{\mathbb{R}^n}(\mathcal{P}) \) describes a set of environments.

We now have the tools to interpret both the concrete domain \( \mathcal{P}(\mathbb{R}^n) \) as \( \text{P}(\text{ENV}) \) and to interpret the abstract domain \( \text{Poly}^n \) as the abstract domain \( \text{ENV} \).
4.4.2 The Abstract Semantic Function

Since $CS_P$ is computed by the function $CS_P$, program semantics is approximated by finding an abstract semantic function $\widehat{CS}_P : (Q \rightarrow \mathcal{E}nv) \rightarrow (Q \rightarrow \mathcal{E}nv)$. 

In the following, we use the abstract domain $\langle \mathbb{R}^n, \alpha_{CP}, \gamma_{CP}, Poly^n \rangle$, this means that

$$\widehat{CS}_P : (Q \rightarrow Poly^n) \rightarrow (Q \rightarrow Poly^n)$$

and we generally denote an element of $Q \rightarrow Poly^n$ as $\widehat{s}$.

We will now define $\widehat{CS}_P$ by type of program point and informally argue why it approximates $CS_P$.

Initial States

Let $q \in I_P$, then

$$\widehat{CS}_P(\widehat{s})(q) = \mathbb{P}_H(\emptyset)$$

that is for any initial state, we associate the polyhedron without constraints, that is, the polyhedron that contains all of $\mathbb{R}^n$, which represents all possible environments. This is a safe approximation of

$$CS_P(S)(q) = I_P$$

since $I_P$ does not put any restrictions on what the environments, an initial state can have any environment.

Basic Blocks

Let $q = \langle n_0, n_1 \rangle$, $q' = \langle n_1, n_2 \rangle$ such that $n_1$ is a basic block node associated with the assignment list $l$, then

$$\widehat{CS}_P(\widehat{s})(q') = \widehat{S}_{node}(n_1)(\widehat{s}(q))$$
where
\[
\hat{S}_{\text{node}}(n) = \hat{S}_{\text{assignment list}}(l)
\]
\[
\hat{S}_{\text{assignment list}}(\epsilon) = \lambda P. P
\]
\[
\hat{S}_{\text{assignment list}}(a, l) = \hat{S}_{\text{assignment list}}(l) \circ \hat{S}_{\text{assignment}}(a)
\]
\[
\hat{S}_{\text{assignment}}(x_j = \sum_{i=0}^{n-1} a_i x_i + c) = \lambda P. P[x_j \leftarrow \sum_{i=0}^{n-1} a_i x_i + c]
\]
\[
\hat{S}_{\text{assignment}}(x_j = f(...)) = \lambda P. P[x_j \leftarrow ?]
\]

For each assignment in a basic block we make a linear transformation if the assignment is linear, and an elimination if the assignment uses some non-linear function \(f(...)\) in its expression. This ensures that all points in the polyhedron \(\hat{s}(q')\) are transformed according to the assignment.

**Join Nodes**

If the abstract domain used in static analysis contains infinite ascending chains (as the polyhedral domain does), it is necessary to introduce widening as described in Section 4.2.3. It can be shown that it is sufficient to use the widening operator at least once per cycle in the flow chart to ensure termination [34]. Thus, instead of using the sequences described in Section 4.2.3, it is sufficient to introduce the widening operation once per loop. Let \(q_1 = \langle n_0, n \rangle, q_2 = \langle n_1, n \rangle, q' = \langle n, n' \rangle\) such that \(n\) is a loop join node, then
\[
\hat{C}S_P(\hat{s})(q') = \hat{s}(q') \triangledown (\hat{s}(q_1) \sqcup \hat{s}(q_2))
\]
and if \(n\) is a normal join node, then
\[
\hat{C}S_P(\hat{s})(q') = \hat{s}(q_1) \sqcup \hat{s}(q_2)
\]

That is, widening is done at loop join nodes, which ensures that the fixed point iteration terminates.

**Conditionals**

Let \(q = \langle n_0, n_1 \rangle, q_{\text{true}} = \langle n_1, n_2 \rangle, q_{\text{false}} = \langle n_1, n_3 \rangle\) such that \(n_1\) is a conditional node associated with a linear conditional \(b\). This means that the set of environments associated with \(\hat{C}S_P(\hat{s})(q_{\text{true}})\) should be all environments
Chapter 4. Abstract Interpretation and the Polyhedral Domain

associated with \( \hat{s}(q) \) except for those that are satisfied by constraint \( b \) when seen as a set of points. The largest polyhedron containing all points such that \( b \) is true is the polyhedron \( \mathbb{P}_H(\{b\}) \). For this reason we define:

\[
\mathcal{CS}_{\mathcal{P}}(\hat{s})(q_{\text{true}}) = \hat{s}(q) \cap \mathbb{P}_H(\{b\})
\]

And by analogy, we define:

\[
\mathcal{CS}_{\mathcal{P}}(\hat{s})(q_{\text{false}}) = \hat{s}(q) \cap \mathbb{P}_H(\{\neg b\})
\]

4.5 Abstract Interpretation Example

To see how abstract interpretation is used in practice, this section shows an example of applying abstract interpretation to program \( L \) depicted in Figure 3.2 on page 27. Using the definition of \( \mathcal{CS} \) above, we see that we get \( \mathcal{CS}_L \):

The equations for the polyhedral domain are:

\[
\begin{align*}
\mathcal{CS}_L(\hat{s})(q_0) &= \mathbb{P}_H(\emptyset) \\
\mathcal{CS}_L(\hat{s})(q_1) &= \hat{s}(q_0)[i \leftarrow 0] \\
\mathcal{CS}_L(\hat{s})(q_2) &= \hat{s}(q_2) \nabla (\hat{s}(q_1) \sqcup \hat{s}(q_5)) \\
\mathcal{CS}_L(\hat{s})(q_3) &= \hat{s}(q_2) \cap \mathbb{P}_H(\{i > n\}) \tag{4.1} \\
\mathcal{CS}_L(\hat{s})(q_4) &= \hat{s}(q_2) \cap \mathbb{P}_H(\{i <= n\}) \\
\mathcal{CS}_L(\hat{s})(q_5) &= \hat{s}(q_3)[i \leftarrow i + 1]
\end{align*}
\]

We are now interested in computing \( \mathcal{CS}'_L \), which is defined as \( \text{lfp}(\mathcal{CS}_L) \). According to Proposition 8, this can be done by fixed point iteration. Thus,
starting from \( \hat{s}_0 = \lambda q . \perp \) gets:

\[
\begin{align*}
\hat{CS}_L (\hat{s}_0)(q_0) &= P_H(\emptyset) \\
\hat{CS}_L (\hat{s}_0)(q_1) &= P_H(\emptyset)[i \leftarrow 0] = P_H(\{i = 0\}) \\
\hat{CS}_L (\hat{s}_0)(q_2) &= P_H(\emptyset) \nabla (P_H(\{i = 0\}) \cup P_H(\emptyset)) \\
&= P_H(\emptyset) \nabla P_H(\{i = 0\}) = P_H(\{i = 0\}) \\
\hat{CS}_L (\hat{s}_0)(q_3) &= P_H(\{i = 0\}) \cap P_H(\{i > n\}) = P_H(\{i = 0, i > n\}) \\
\hat{CS}_L (\hat{s}_0)(q_4) &= P_H(\{i = 0\}) \cap P_H(\{i \leq n\}) = P_H(\{i = 0, i \leq n\}) \\
\hat{CS}_L (\hat{s}_0)(q_5) &= P_H(i = 0, i \leq n)[i \leftarrow i + 1] \\
&= P_F(\{(0, 0), (0, 1), \emptyset\}[i \leftarrow i + 1] \\
&= P_F(\{(1, 0), (0, 1), \emptyset\} = P_H(\{i = 1, i - 1 \leq n\})
\end{align*}
\]

Here both the constraint and the frame representations are used for convenience of the presentation. A second iteration where \( \hat{s}_1(q) = \hat{CS}_L (\hat{s}_0)(q) \) yields:

\[
\begin{align*}
\hat{CS}_L (\hat{s}_1)(q_0) &= P_H(\emptyset) \\
\hat{CS}_L (\hat{s}_1)(q_1) &= P_H(\{i = 0\}) \\
\hat{CS}_L (\hat{s}_1)(q_2) &= P_H(\{i = 0\}) \nabla (P_H(\{i = 0\}) \cup P_H(\{i = 1, i - 1 \leq n\})) \\
&= P_H(\{i = 0\}) \nabla P_H(\{i = 1, i - 1 \leq n\}) = P_H(\{i \geq 0\}) \\
\hat{CS}_L (\hat{s}_1)(q_3) &= P_H(\{i \geq 0\}) \cap P_H(\{i > n\}) = P_H(\{i \geq 0, i > n\}) \\
\hat{CS}_L (\hat{s}_1)(q_4) &= P_H(\{i \geq 0\}) \cap P_H(\{i \leq n\}) = P_H(\{i \geq 0, i \leq n\}) \\
\hat{CS}_L (\hat{s}_1)(q_5) &= P_H(i \geq 0, i \leq n)[i \leftarrow i + 1] \\
&= P_F(\{(0, 0), (1, 1), \emptyset\}[i \leftarrow i + 1] \\
&= P_F(\{(1, 0), (1, 1), \emptyset\} = P_H(\{i \geq 1, i - 1 \leq n\})
\end{align*}
\]

If we now compute \( \hat{s}_2(q_2) = \hat{CS}_L (\hat{s}_1)(q_2) \) we find that \( \hat{s}_2(q_2) = \hat{s}_1(q_2) \) meaning that the least fixed point has been found. That is \( CS_L = \hat{s}_1 \).
Chapter 4. Abstract Interpretation and the Polyhedral Domain

An Example

Using the example of program $L$ again, we can extract a set of possible environments for program point $q_3$ by

$$\text{ENV}_{\mathbb{R}^2}(CS_L(q_2)) = \text{ENV}_{\mathbb{R}^n}(P_H(\{i \geq 0, i \leq n\}))$$

$$= \text{ENV}_{\mathbb{R}^2}\{\langle x, y \rangle \mid \forall x, y \in \mathbb{R} \land x \geq 0, x \leq y\}$$

$$= \{\sigma \in \text{ENV} \mid \sigma(i) \geq 0, \sigma(i) \leq \sigma(n)\}$$

That is, all environments in which $i$ is greater than zero and the value of $i$ is no larger than as the value of $n$. In this example, the value of $n$ bounds the size of $|\text{ENV}_{\mathbb{R}^2}(CS_L(q_2))|$ to $\sigma(n) + 1$. 

Chapter 5

Parametric WCET Analysis

5.1 Introduction

Chapter 3 outlined our Census method for computing the execution time of a program by counting the number of possible states and summing each state multiplied by the execution time of the block that computed that state. Chapter 4 showed how to get a computable approximation of the possible set of states by abstract interpretation. In this Chapter we formulate how we use these tools to perform a parametric WCET analysis.

5.2 Overview

Our approach is divided into a number of independent phases which each consists of a number of steps. The phases are outlined in Figure 5.1. Since we are not focused on low-level analysis in this thesis, we assume that there exists a low-level analysis that generates $\ell_P$ for the program. The state counting phase and the parametric calculation phase are independent of each other and the results are combined in the combination phase which then results in a parametric estimate of the WCET.
5.3 The State Counting Phase

The first process is the using the Census method to calculate the maximal number of states that can be visited at each program point. This phase is called the state counting phase. The state counting phase takes a program as input and returns a function \( SC : Q_P \times ENV \rightarrow \mathbb{Z} \) that takes a program point \( q \) and an input \( \sigma \) and returns the maximum number of states that can possibly visit

Figure 5.1: Overview over our parametric WCET analysis divided into three phases.
5.3 The State Counting Phase

program point $q$ given input $\sigma$. Formally, we define it as

$$SC(q, \sigma) = C(T_\sigma, q)$$

where $C$ is the Census function defined in (3.3) on page 37. In practice, this needs to be computed in a set of steps: First the program is sliced, and the input parameters are determined. Second, an approximation of the set of states that may be visited during execution is computed. This is done by abstract interpretation. Third, the number of concrete states that are contained in the abstract states must be counted for each program point. The state counting phase is visualised in the top of Figure 5.1.

5.3.1 Preparing the Program

As explained in Section 3.6, we are interested in removing all non-control variables from the program under analysis, because states that differ only in the values of non-control variables are not on the same trace. Not removing control variables would likely result in considerably many more states in $\hat{\text{CS}}$ which would result in potential over-approximations.

An efficient way of removing control variables from a program is to use program slicing [117, 118]. The basic idea of program slicing is to find all statements of a program that can affect the computed values at selected points in the program. The resulting sequence of statements is known as a program slice. If the program slice is executed as a program, the selected values evaluate to the same as in the original program. To use program slicing to remove non-control variables, slicing can be configured to find all statements that affect the conditional expressions of the program. Ideally, doing this results in a program slice that contains control variables only. However, a program slicing cannot be perfect in the general case. While all statements containing control variables remain in a conservative slicing, there might be some statements containing non-control variables in the resulting slice.

A slicing method that quickly removes all variables that are not control variables (as defined as in Definition 13) can be found in [100] under the name SIMPLESLICE. SIMPLESLICE begins by adding all variables occurring in conditionals in a set called $keep$. Variables that affect variables in the set $keep$ variables are then recursively added to the set until there are no more such variables. When this happens, the variables which are not in $keep$ cannot be control variables.

Definition 13 and SIMPLESLICE are flow-independent, meaning that variables are treated independent of their position, i.e., independent on the flow of
the program. However, more sophisticated slicing methods [117, 118, 100] that take program flow into consideration can potentially remove many more statements and variables at different points in a program. Since slicing is important to achieve precision in the Census method, best results are achieved by using as aggressive slicing as possible (as long as it is safe, that is, not removing any control variables). However, in this presentation we use SIMPLESLICE since it corresponds to Definition 13, while it would certainly be possible to use a more precise slicing for the Census method to work.

In the process of preparing the program, the program is also modelled as described in Section 3.7.1 to make sure that the set of input variables are the set of invariant variables. In this way, it is always possible to tell whether a given state is corresponding to a given input or not. The process to find the set of artificial input variables \( X' \) as described in Section 3.7.1 involves the functions \( \text{def}_T \) and \( \text{use}_T \). While we have not developed an automatic process for this, there are standard static analysis techniques to find def-use chains in programs [2]. For small examples it is sufficient to determine manually which variables may be parameters.

### 5.3.2 Relational Abstract Interpretation

When the program is prepared, abstract interpretation is performed. We use the polyhedral domain to do this since it is very accurate while having acceptable complexity. However, it is possible to use any relational abstract domain in this phase. A relational domain is a domain that does not analyse all variables independently of each other. The domain \( \langle P(Z), \alpha_{\text{Sign}}, \gamma_{\text{Sign}}, \text{Sign} \rangle \) presented in Section 4.2.1 is not a relational domain, for instance, because each variable gets an abstract value independent of all other variables. However, the polyhedral domain is relational, since the bounds of variables depends on the values of other variables. The reason that the domain has to be relational is that the number of states are always counted in the context of the given input.

### 5.3.3 State Counting

When the abstract interpretation has been performed, each program point \( q \in Q_P \) is associated with an abstract environment that represents all states that can possibly be visited at \( q \). The interesting thing is the maximum number of states that can reach \( q \) on input \( \sigma \), that is \( C(T_\sigma, q) \). This is achieved by counting the number of states that are represented by the abstract environment and which has all invariant variables consistent with the given input. So if \( T_\sigma \)}
is an abstract state corresponding to program point \( q \), then
\[
\text{SC}(q, \sigma) = |\{\sigma' \in \gamma(\hat{\sigma}; q) \mid \forall v \in \text{IV}_P : \sigma'(v) = \sigma(v)\}|
\] (5.1)

The process of finding this number is dependent on the abstract domain. This function is an output of the process. When a program has been analysed, the function \( C \) is available and it should return a value for any given input. This means that \( \sigma \) is an unknown in (5.1), in other words, (5.1) is a function of \( \sigma \).

In our case an abstract environment consists of a convex polyhedron. The problem then is to find the number of integer points in a polyhedron with the additional constraint that the variables corresponding to the input are fixed. This means that the number of points should be counted symbolically in terms of the input variables.

There are several techniques that makes it possible to symbolically count the number of integer points inside a convex polyhedron. The following sections outlines three possible techniques.

**Ehrhart Polynomials**

Ehrhart presents in [29] a method of symbolically counting the number of integer points inside the union of bounded polyhedra. The method uses Ehrhart’s theory to find quasi-polynomials (polynomials having periodic functions as coefficients, which can be seen as a finite set of polynomials) which corresponds to the number of integer points in polyhedra.

**Pugh’s Method**

Pugh introduces a method of successive projections in [96]. The method is used to count the number of solutions to a Presburger formula, which is more general than unions of bounded polyhedra. The method is presented as a set of rules to successively transform a symbolic summation to a compound formula. The method is a set of rules rather than an algorithm, meaning that some additional work has to be done in order to make an implementation.

The method computes the result of generalised sums \((\Sigma V : P : x)\) where \( V \) is a set of variables to sum over, \( P \) is a Presburger formula (the guard) and \( x \) is any formula. The result of such a sum is the sum for all variables \( v \in V \) which satisfy \( P \) of \( x \). As a simple example, the sum \( \sum_{i=1}^{u} v \) is represented by the general summation \((\Sigma\{v\} : l \leq v \leq u : v)\) and the sum \( \sum_{i=0}^{n} \sum_{j=i}^{m} 1 \) would be represented by \((\Sigma\{i, j\} : 0 \leq i \leq n \land i \leq j \leq m : 1)\). The result is then computed by choosing an appropriate projection rule to simplify the
formula. The most important rule to reduce a generalised sum \((\Sigma V : P : x)\) is to choose a variable \(v \in V\) and compute the general sum

\[
(\Sigma V \setminus \{v\} : P' : (\Sigma \{v\} : l \leq v \leq u : x))
\]

where \(P'\) is \(P\) where all information about \(v\) is removed. Since \((\Sigma \{v\} : l \leq v \leq u : x)\) is equivalent to \(\sum_{v=l}^{u} x\), known formulae of summations over the form of \(x\) can be used to simplify it. If \(V \setminus \{v\}\) is non-empty, another variable is chosen and the procedure is repeated until \(V = \emptyset\), and the result is a sum of generalised sums \((\Sigma : G : x')\) which should be read as “\(x'\) if \(G\) holds, else \(0\)”. This result is symbolic in the variables occurring free in \(x\) or \(P\) but not in \(V\). To exemplify, take the sum \((\Sigma \{i, j\} : 0 \leq i \leq n \land i \leq j \leq m : 1)\) again. Applying the rule, projecting the variable \(i\), above would yield

\[
(\Sigma \{i, j\} : 0 \leq i \leq n \land i \leq j \leq m : 1) = (\Sigma \{j\} : i \leq j \leq m : (\Sigma \{i\} : 0 \leq i \leq n : 1))
\]

Now, naturally \((\Sigma \{i\} : 0 \leq i \leq n : 1) = n+1\) (this is a “known formula”), so we can conclude

\[
(\Sigma \{j\} : i \leq j \leq m : (\Sigma \{i\} : 0 \leq i \leq n : 1)) = (\Sigma \{j\} : i \leq j \leq m : n+1)
\]

and continue applying rules until we cannot apply more rules. The situation is not always this easy however; variables may have several lower/upper bounds or be unbounded and bounds can be negative and/or rational. All these cases are handled in [96].

This method is implemented in our first prototype implementation (see also Chapter 9) of our parametric WCET method [25]. Our implementation is a bit modified since we have the special case where the integer points are counted inside a convex polyhedron. Since this is less general than having Presburger formulae, the problem becomes a bit simpler. In the following, we present our implementation that is restricted to counting integer points inside convex polyhedra. We assume two restrictions of the generalised sums; first, the guard is always an \(H\)-polyhedron (since this information will be available from the abstract interpretation) rather than a Presburger formula. Second, the formula \(x\) in \((\Sigma V : P : x)\) is always a polynomial with integer coefficients. This simplifies both representation and computation. Polynomials can easily be modelled as a sum of terms, where a term is a vector representing an integer coefficient and exponents of the variables. As an example we can model the polynomial
5.3 The State Counting Phase

$3a^2b^3 + 5a^4$ (assuming $\text{VAR}_P = \{a, b\}$) as the sum of the polynomial-terms $(3\ 2\ 3)$ and $(5\ 4\ 0)$. This also makes arithmetical operations on these vectors straightforward to implement. Furthermore, since the guards are polyhedra, the lower and upper bound of any variable are sets of linear expressions. Since summing a linear expression over a polynomial is again a polynomial, this model is closed under summations. However, these restrictions sometimes require the result to be slightly over-approximated. The constraint $3a - b \leq 0$ gives an upper bound for $a$ as $a \leq \lceil \frac{b}{3} \rceil$, since $a$ and $b$ are integers. As seen, the upper bound is not a polynomial. Since $\frac{b}{3}$ is a safe upper bound for $\lceil \frac{b}{3} \rceil$ and is on polynomial form, we can use it as approximation. Lower bounds are handled in a similar fashion.

Barvinok’s Rational Functions

In [114], a parametric version of Barvinok’s rational functions [13] is presented, using a similar method of [29] to find quasi-polynomials that represent the number of integer points inside polyhedra. There also exists a library for symbolically counting points inside polyhedra called Barvinok\textsuperscript{1}. This library is distributed with an independent tool called iscc that can parse text-files describing polyhedra and quickly give symbolical counts as output in the form of a text-file. Details about this library and its implementation can be found in [115]. Our implementation of parametric WCET analysis in the static analysis tool SWEET uses iscc to count integer points (see Chapter 9).

5.3.4 Example of the State Counting Phase

This section gives an example of the counting phase using program $L$. Slicing $L$ will not remove anything, since both $n$ and $i$ are control variables. In addition, there are no artificial parameters to add to $L$ since the only input variable $n$ in this case is invariant. Thus, $L$ is already “prepared” for the state counting phase. Next, we perform polyhedral abstract interpretation on $L$ resulting in the abstract states given in (4.2) on page 69. Finally, we symbolically count the number of states in each program point. In this example we use Pugh’s method to do this. Since $n$ is the input variable, we sum only over the variable $i$ since $n$ should remain fixed. The guard in the general sums will correspond to the inequalities given for a convex polyhedron. The result of computing the sizes is shown below:

\footnote{http://freecode.com/projects/barvinok}
Note that no finite bound could be found for states that can be reached at the given program point. The definition of and a program point and returns an upper bound of the possible number of expression (similar to C-syntax), it should be understood as "if then, \[ n \geq 0 \] will be taken exactly once. Thus, a better definition of has been obtained. The function takes an instance of the input parameter \( n \) and a program point and returns an upper bound of the possible number of states associated with the program point.

The state counting phase and the low-level analysis provides everything that is needed to use (3.5) since each program point \( q \) is accessible from the results of abstract interpretation alone. The timing is still of a program subject to the constraints that exists in a program that may not be assigned a non-linear expression for instance.

The purpose of the parametric calculation is to provide the maximum cost of states associated with the program point. In fact, \( SC(\sigma, q) \) may very well be an over-approximation of the number of possible states. In this sense, (3.5) is the calculation phase that combines the low-level analysis results with the flow-analysis results. However, using (3.5) directly is a naive approximation of the set of states, meaning that \( SC(\sigma, q) \) is not necessarily the tightest one.

\[
\begin{align*}
| \widehat{CS}(q_0) | &= (\Sigma i : \varnothing : 1) = \infty \quad \text{(unbounded sum)} \\
| \widehat{CS}(q_1) | &= (\Sigma i : i = 0 : 1) = 1 \\
| \widehat{CS}(q_2) | &= (\Sigma i : i \geq 0 : 1) = \infty \\
| \widehat{CS}(q_3) | &= (\Sigma i : i \geq 0 \land i \geq n + 1 : 1) = \infty \\
| \widehat{CS}(q_4) | &= (\Sigma i : 0 \leq i \leq n : 1) = (\Sigma : n \geq 0 : n + 1) \\
| \widehat{CS}(q_5) | &= (\Sigma i : 1 \leq i \leq n + 1 : 1) = (\Sigma : n \geq 0 : n + 1)
\end{align*}
\]

Note that we have used the short hand \( \Sigma i \) for \( \Sigma \{i\} \). By this, the function \( SC_L \) has been obtained. The function takes an instance of the input parameter \( n \) and a program point and returns an upper bound of the possible number of states that can be reached at the given program point. The definition of \( SC_L \) is then,

\[
\begin{align*}
SC_L(q_0, \sigma) &= \infty \\
SC_L(q_1, \sigma) &= 1 \\
SC_L(q_2, \sigma) &= \infty \\
SC_L(q_3, \sigma) &= \infty \\
SC_L(q_4, \sigma) &= (\sigma(n) \geq 0 \land \sigma(n) + 1 : 0) \\
SC_L(q_5, \sigma) &= (\sigma(n) \geq 0 \land \sigma(n) + 1 : 0)
\end{align*}
\]

where the notation \( (n \geq 0 \land n + 1 : 0) \) is a compressed conditional expression (similar to C-syntax), it should be understood as “if \( n \geq 0 \) then \( n + 1 \) else 0”. Assumption 1 says that the entry and exit points of a program will be taken exactly once. Thus, a better definition of \( SC_L \) is to always set the state count of these points to one (rather than the number given from abstract interpretation), for \( L \) that is the program points \( q_0 \) and \( q_3 \). This gives us,

\[
\begin{align*}
SC_L(q_0, \sigma) &= 1 \\
SC_L(q_3, \sigma) &= 1
\end{align*}
\]

Note that no finite bound could be found for \( q_2 \). This is because the widening in the abstract interpretation yielded an unbounded polyhedron for \( \widehat{CS}(q_2) \).
However, it is not necessary to have bounds on all program points in order to obtain a finite bound on the WCET since it is in general the minimum of the bounds that are interesting as is seen in the next section.

5.4 The Parametric Calculation Phase

The state counting phase and the low-level analysis provides everything that is needed to use (3.5) since each program point \( q_k \) has a timing \( f(q_k) \) and a function in \( \sigma \) that bounds the number of states that may visit it: \( SC(\sigma, q_k) \). In this sense, (3.5) is the calculation phase that combines the low-level analysis results with the flow-analysis results. However, using (3.5) directly is a naïve bound and not necessarily the tightest one.

Abstract interpretation is an approximation of the set of states, meaning that \( SC(\sigma, q_k) \) may very well be an over-approximation of the number of possible states. In fact, \( SC(\sigma, q_k) \) may not even be finite, since abstract interpretation may fail to obtain safe finite bounds on some variables (see for example \( SC_L(\sigma, q_2) \)). For the polyhedral domain this can happen when a variable is assigned a non-linear expression for instance.

The purpose of the parametric calculation is to provide the maximum cost of a program subject to the constraints that exists in a program that may not be accessible from the results of abstract interpretation alone. The timing is still affected by the maximum number of states that can visit each program, so the result of the parametric calculation is a function:

\[
PC_P : \mathbb{N}^{|Q_P|} \rightarrow \mathbb{N}
\]

where each \( i \)th argument to \( PC_P \) correspond to the maximum number of times that program point \( q_i \) can be executed, or equivalently the maximum number of states associated with the program point.

The purpose of the calculation phase in this case is to find a tighter version of (3.5) by taking constraints on the reachable states into consideration which could not be obtained from abstract interpretation alone. A common type of constraints are structural constraints that affect how many times a program point can be visited – a program point can never be visited more times than the sum of its predecessors or successors.

We present two approaches to parametric calculation in the following sections. Both have been implemented and tested in our experiments (see Chapter 8).
5.4.1 Parametric Integer Programming

A common calculation method for WCET analysis is the Implicit Path Enumeration Technique (IPET)\[73, 74\]. IPET can be described as maximising a function corresponding to the worst possible time to execute a program, i.e.

$$\sum_{q \in Q} \ell_P(q) x_q$$

where $x_q$ is an integer representing the maximum possible times that $q$ can execute. This is all subject to a set of linear constraints in the values of $x_q$. These constraints can be of various types, for instance structural constraints (coming from the graph structure), or infeasible paths (excluding paths that are not possible) or bounds on loops (a program point cannot be visited more times than the maximum number of iterations in a loop), or of other types as long as they are linear. The problem is then solved by using integer linear programming which is typically quite efficient as there are fast and efficient tools for solving integer linear programs such as CPLEX or CLP.

One method that we have used to perform parametric calculation (and which was suggested in Lisper’s original publication [76]) is a parametric version of the IPET calculation. The difference between a parametric and non-parametric IPET calculation is that the parametric version has a set of parameters in the problem and in the solution. Parametric IPET maximises (5.4) subject to a set of constraints, but with additional parametric constraints. In our application, we add parameters $p_q$ for each $q \in Q_P$ that represent the maximum number of states that can reach each program point $q$. Along with the following set of constraints:

$$\{x_q \leq p_q \mid q \in Q_P\}$$

Feautrier [43] introduced a method for solving parametric integer problems in general. That is, a method for solving ILP-problems with the possibility to add unknowns into the constraints and solution. This means that this method can be used to perform the parametric IPET as described.

The result of a parametric integer problem is a binary tree where the leaves correspond to linear solutions and the other nodes correspond to linear conditionals. The tree is represented by a nested if-statement (an example of a solution can be seen in Figure 5.2 on page 82). This formula then represents the WCET of a program given the maximum number of states for each program point $p_q$. Parametric integer programming is naturally more complex than normal ILP, and in order to be efficient the number of parameters should be few.
Our first prototype implementation uses a tool called Piplib\(^2\) to obtain \(PC_P\), see Chapter 9.

**Parametric IPET Example**

To show how to obtain \(PC_P\) by parametric integer programming, we illustrate again on program \(L\). The function \(PC_L\) is then obtained by maximising:

\[
\sum_{q \in Q_L} \ell_L(q)x_q = x_0 + 3x_1 + x_2 + 2x_3 + 2x_4 + 8x_5
\]  

(5.5)

subject to some constraints. To get a bounded problem it is enough to provide the structural constraints of the program together with the parametric constraints.

The structural constraints of a program are obtained by observing that the number of executions of a node equals the sum of the number of executions of arcs that enter the node, and also equals the sum of the number of executions of arcs that leave the node. As an example, if \(q\) is a conditional node, and \(q_{\text{true}}, q_{\text{false}}\) are the succeeding program points, then:

\[
x_q = x_{q_{\text{true}}} + x_{q_{\text{false}}}
\]

in addition, the initial program point \(q_0\) and the final program point \(q_f\) are considered to be taken exactly once, formally:

\[
x_0 = 1 \\
x_f = 1
\]

The structural constraints of \(L\) are:

\[
x_0 = 1 \\
x_1 = x_0 \\
x_2 = x_1 + x_5 \\
x_2 = x_3 + x_4 \\
x_5 = x_4 \\
x_3 = 1
\]  

(5.6)

Finally, the symbolic constraints \(x_q \leq p_q\) for all \(q \in Q_L\) are be added. Maximising (5.5) subject to these constraints with integer linear programming

\(^2\text{http://www.piplib.org/}\)
\[ PC_L = \lambda p_0, p_1, p_2, p_3, p_4, p_5. \]

\[ \begin{align*}
& \text{if } p_2 \geq 1 \text{ then} \\
& \quad \text{if } p_0 \geq 1 \text{ then} \\
& \quad \quad \text{if } p_1 \geq 1 \text{ then} \\
& \quad \quad \quad \text{if } p_3 \geq 1 \text{ then} \\
& \quad \quad \quad \quad \text{if } p_2 \leq p_4 + 1 \text{ then} \\
& \quad \quad \quad \quad \quad \text{if } p_2 \leq p_5 + 1 \text{ then}
& \quad \quad \quad \quad \quad \quad 11p_2 - 4
& \quad \quad \quad \quad \text{else}
& \quad \quad \quad \quad \quad 11p_5 + 7
& \quad \quad \text{end if}
& \quad \text{else}
& \quad \quad \text{if } p_4 \leq p_5 \text{ then}
& \quad \quad \quad 11p_4 + 7
& \quad \text{else}
& \quad \quad \quad 11p_5 + 7
& \quad \text{end if}
& \text{end if}
& \text{end if}
& \text{end if}
& \text{else}
& \quad 0
& \text{end if}
\end{align*} \]

Figure 5.2: The function \( PC_L \) resulting from parametric integer programming

yields the result shown in Figure 5.2. As can be seen in the figure, this result is the definition of \( PC_L \). As an example, suppose that \( p_0, ..., p_5 \) are instantiated as \( \langle 1, 1, \infty, 1, 3, 3 \rangle \) (corresponding to \( SC_L([n \mapsto 2], q_{0,1,2,3,4,5}) \)). Then,

\[ PC_L(1, 1, \infty, 1, 3, 3) = 11 \cdot 3 + 7 = 40 \]

which should be easy to see by studying Figure 5.2.

## 5.5 Reducing the Number of Variables

Parametric integer programming has exponential complexity in the number of variables in the worst-case, making scalability problematic. In this section we
introduce a way of reducing the number of variables. This section uses a lot of terminology from linear algebra and definitions can be found in any book in linear algebra, such as [7].

The structural constraints of a program produce an under-determined system of equations. In such a system with \( n \) variables, the solution space is the span of a set of \( n - r \) vectors (where \( r \) is the rank of the constraint matrix). Thus, these \( n - r \) vectors form a basis for the solution space. The variables can be expressed as linear combinations of this basis, meaning that the problem can be computed using only the basis. Let \( Ax = b \) be a system of structural and possibly other linear equations obtained from flow analyses and \( y = c^T x \) be the cost function. The constraints together with the cost function is

\[
\begin{pmatrix}
1 & -c \\
0 & A
\end{pmatrix}
\begin{pmatrix}
y \\
x
\end{pmatrix} =
\begin{pmatrix}
0 \\
b
\end{pmatrix}
\]

If we perform Gauss-Jordan elimination on the above (including the right-hand side by augmenting the constraint matrix by \((0 \ b)^T\)) and re-arrange the columns of \( A \) and the components of \( x \) such that all pivot columns are to the left, and \( x \) is re-arranged accordingly, we get

\[
\begin{pmatrix}
1 & 0 & -c' \\
0 & I_r & A'
\end{pmatrix}
\begin{pmatrix}
y \\
x_{BV} \\
x_{FV}
\end{pmatrix} =
\begin{pmatrix}
z \\
b'
\end{pmatrix}
\]

where \( I_r \) is the \( r \times r \) identity matrix and \( r \) is the rank of \( A \). Furthermore, \( z \) is the last column of the solution of the augmented matrix after elimination. The vector \( x \) has now been partitioned into two vectors, one partition is the vector of basic variables \( x_{BV} \) with \( r \) components, and the other being \( n - r \) free variables \( x_{FV} \) (where \( n \) is the number of columns of \( A \)). Note that this transformation also removes any redundant constraints from the system. From this we can derive two important equations. One is the objective function expressed in terms of the free variables

\[
y = z + c'x_{FV}\]

and a way to express the basic variables in terms of the free ones

\[
x_{BV} = b' - A'x_{FV}\]

As we model the parametric upper bounds as the constraints \( x \leq p \), we can
now simply model our IPET problem as

\[
\begin{pmatrix}
z + c' x_{FV} \\
b' - A' x_{FV} \\
x_{FV}
\end{pmatrix} \leq
\begin{pmatrix}
y \\
p_{BV} \\
p_{FV}
\end{pmatrix}
\]

(5.11)

where we have partitioned and re-arranged \( p \) exactly as for \( x \). Now it suffices to solve the IPET with these constraints, thus reducing the number of unknowns by the rank of \( A \). This method of eliminating variables is not restricted to the parametric case, but can be used to reduce the dimensionality of any IPET problem.

### 5.5.1 Concrete Example of Variable Reduction

As an example on how the variable reduction can be applied we shall perform variable reduction on the program \( L \). First we assemble a matrix like (5.7) by assembling the cost function (5.5) of \( L \) and the structural constraints (5.6). Thus, we have

\[
c = (1 \ 3 \ 1 \ 2 \ 2 \ 8)
\]

and

\[
b = \begin{pmatrix}
1 \\
0 \\
0 \\
0 \\
0 \\
1
\end{pmatrix}
\]

and

\[
A = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & -1 \\
0 & 0 & -1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
\]

and finally,

\[
x = (x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ x_5)
\]
The resulting constraint matrix as in (5.7) is:
\[
\begin{pmatrix}
1 & -1 & 3 & -1 & -2 & -2 & -8 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & -1 \\
0 & 0 & 0 & -1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5
\end{pmatrix} =
\begin{pmatrix}
y \\
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5
\end{pmatrix}
\]

Now we perform Gauss-Jordan elimination of the above to obtain a matrix as in (5.8). The result is:
\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & -11 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1
\end{pmatrix}
\begin{pmatrix}
y \\
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5
\end{pmatrix} =
\begin{pmatrix}
7 \\
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5
\end{pmatrix}
\]

This means that for \( L \) we have:
\[
c' = (11), A' = (0 0 0 -1 0 -1)^T
\]
\[
x_{BV} = (x_0 x_1 x_2 x_3 x_4)^T, x_{FV} = (x_5)
\]
\[
b' = (1 1 1 0)^T, z = 7
\]

Thus, we can conclude (5.9) and (5.10) as
\[
y = 7 + 11x_5
\]
and
\[
x_0 = 1, x_1 = 1, x_2 = x_5 + 1, x_3 = 1, x_4 = x_5
\]

We can model the IPET problem by (5.11), which gives the following concrete constraints:
\[
7 + 11x_5 \leq y
\]
\[
1 + x_5 \leq p_2
\]
\[
1 \leq p_0, p_1, p_3
\]
\[
x_5 \leq p_4
\]
\[
x_5 \leq p_5
\]
thus, we have reduced the problem to one unknown variable \((x_5)\) from the original six. Solving this system with Piplib results in the following, which can be inspected to realise that it is the same as the solution shown in Figure 5.2:

\[
\lambda p_2, p_4, p_5.
\]

if \(p_2 \geq 1\) then

if \(p_4 + 1 \geq p_2\) then

if \(p_5 + 1 \geq p_2\) then

\[y = 11p_2 - 4\]

else

\[y = 11p_5 + 7\]

end if

else

if \(p_4 \leq p_5\) then

\[y = 11p_4 + 7\]

else

\[y = 11p_5 + 7\]

end if

end if

else

0

end if

5.5.2 The Minimum Propagation Algorithm

Our experiments in our first prototype implementation of parametric WCET analysis showed that parametric calculation with parametric integer programming was a bottle-neck in the method (see Chapter 9 for details). For this reason, we have developed another approach to parametric calculation, called the minimum propagation algorithm or MPA. While parametric integer programming is very flexible and precise it is also very costly, especially when the set of parameters gets large. Since the number of parameters in our case corresponds to the number of program points, the number of parameters quickly grows as the size of the program under analysis grows. MPA is designed to be more efficient than parametric integer programming, this is possible by making it specific to parametric calculation and by sacrificing some precision. MPA is described in detail in Chapter 6.
5.6 The Combination Phase

The combination phase consists of combining the result from the state counting phase and the parametric calculation and to simplify the result in order to get an as simple as possible formula for the WCET of a program in terms of its input.

To obtain a function that takes the input of a program and returns the WCET with the given input, the results from previous phases are combined by functional composition:

\[
PWCET_P(\sigma) = PC_P(SC_P(\sigma, q_0), ..., SC_P(\sigma, q_f))
\]

Figure 5.3 shows the composition of \(PC_L\) and \(SC_L\) that were derived in Section 5.4 and 5.3 respectively. As an example, we compute \(PWCET_L\) with the argument \([n \mapsto 2]\).

\[
PWCET_L([n \mapsto 2]) = PC_L(SC_L([n \mapsto 2], q_0), ..., SC_L([n \mapsto 2], q_5))
= PC_L(1, 1, \infty, 3, 3)
= 40
\]

Note that this result is also obtained by substituting 2 for \(\sigma(n)\) in Figure 5.3. Compare this result to the trace computation in Table 3.1 on page 34. The worst-case trace was computed as 40 which correspond exactly to the result in this case. So in this case the exact result was obtained from the method.

5.7 Simplification

The final formula for \(PWCET_L\) shown in Figure 5.3 contains a lot of redundancy and is unnecessarily complex for this small example. Of course, this redundancy and complexity is worse for larger programs. Thus, there is a need for simplification. Examples of trivial simplifications that can be done:

- All then branches can be cut from conditionals on the form \(\infty \leq x\).
- All else branches can be cut from conditionals on the form \(\infty \geq x\).
- All else branches can be cut from conditionals on the form \(x \leq \).
- If \(x, y\) are constants such that \(x \leq y\), then all else branches can be cut from conditionals on the form \(x \leq y\).
\[ PWCET_L = \lambda \sigma. \]

\[
\begin{aligned}
\text{if } \infty \geq 1 \text{ then} \\
\quad \text{if } 1 \geq 1 \text{ then} \\
\quad \quad \text{if } 1 \geq 1 \text{ then} \\
\quad \quad \quad \text{if } 1 \geq 1 \text{ then} \\
\quad \quad \quad \quad \text{if } \infty \leq (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0) + 1 \text{ then} \\
\quad \quad \quad \quad \quad \text{if } \infty \leq (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0) + 1 \text{ then} \\
\quad \quad \quad \quad \quad \quad 11 \cdot \infty - 4 \\
\quad \quad \quad \quad \quad \text{else} \\
\quad \quad \quad \quad \quad \quad 11(\sigma(n) \geq 0 ? \sigma(n) + 1 : 0) + 7 \\
\quad \quad \quad \quad \text{end if} \\
\quad \quad \text{else} \\
\quad \quad \quad \quad (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0) \leq (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0) \text{ then} \\
\quad \quad \quad \quad 11(\sigma(n) \geq 0 ? \sigma(n) + 1 : 0) + 7 \\
\quad \quad \quad \quad \text{else} \\
\quad \quad \quad \quad \quad 11(\sigma(n) \geq 0 ? \sigma(n) + 1 : 0) + 7 \\
\quad \quad \quad \quad \text{end if} \\
\quad \quad \text{end if} \\
\quad \text{end if} \\
\text{end if} \\
\text{else} \\
\quad 0 \\
\text{end if}
\end{aligned}
\]

Figure 5.3: The resulting WCET$_L$ function
Applying these trivial simplifications to $\text{PWCET}_L$ reduces the formula to:

\[
\text{PWCET}_L = \lambda \sigma \cdot 11(\sigma(n) \geq 0 \ ? \ \sigma(n) + 1 : 0) + 7
\]

\[
= \lambda \sigma \cdot \begin{cases} 
11 \sigma(n) + 18 & \text{if } \sigma(n) \geq 0 \\
7 & \text{otherwise.}
\end{cases}
\]

There are more sophisticated simplifications that might take longer to execute (such as checking if two branches of a tree are equal) but they are still worth it in order to get a simple formula that can be computed very quickly.

### 5.8 Summary

The phases in Figure 5.1 illustrate our approach to parametric WCET analysis which is based on the Census method together with parametric calculation. The state counting phase relies on the theory developed in Chapter 3. This approach has been implemented into a prototype tool, and then in a static analysis tool called SWEET. The details of these implementations are found in Chapter 8 and the evaluation of this method is found in Chapter 9. The next chapter digs deeper into parametric calculation and our algorithm MPA.
Chapter 6

The Minimum Propagation Algorithm

6.1 Introduction

Chapter 5 introduced our approach to parametric WCET analysis by state counting and parametric calculation, in which the example is using parametric integer programming. Unfortunately, the complexity of parametric integer programming is very high and this turns out to be the bottleneck in our approach for the purposes of parametric calculation (see Chapter 9). The solutions obtained from parametric integer programming tend to grow exponentially in program size and for large programs the tool Piplib that we used in the evaluation failed to deliver solutions. This chapter describes an alternative algorithm for parametric calculation called the minimum propagation algorithm (MPA) introduced in [23]. MPA scales much better than parametric integer programming in both computation time and solution size, as seen in Chapter 9.

6.2 The Minimum Propagation Algorithm

The Minimum Propagation Algorithm obtains the function $PC_P$ of a program $P$ given its flow chart and results from low-level analysis ($\ell_P$). Thus, it may operate as the parametric calculation phase as seen in Figure 5.1 on page 72.
Chapter 6

The Minimum Propagation Algorithm

6.1 Introduction

Chapter 5 introduced our approach to parametric WCET analysis by state counting and parametric calculation, in which the example is using parametric integer programming. Unfortunately, the complexity of parametric integer programming is very high and this turns out to be the bottle-neck in our approach for the purposes of parametric calculation (see Chapter 9). The solutions obtained from parametric integer programming tend to grow exponentially in program size and for large programs the tool Piplib that we used in the evaluation failed to deliver solutions. This chapter describes an alternative algorithm for parametric calculation called the minimum propagation algorithm (MPA) introduced in [23]. MPA scales much better than parametric integer programming in both computation time and solution size, as seen in Chapter 9.

6.2 The Minimum Propagation Algorithm

The Minimum Propagation Algorithm obtains the function $PC_P$ of a program $P$ given its flow chart and results from low-level analysis ($\ell_P$). Thus, it may operate as the parametric calculation phase as seen in Figure 5.1 on page 72.
Being a parametric calculation, MPA computes the maximum of
\[ \sum_{q \in Q_P} \ell_q x_q \]
given the structural constraints of \( P \). In addition, the algorithm assumes that each program point \( q \) is associated with a symbolic upper bound for the number of states \( p_q \) which we call the parameters of the problem. That is,
\[ \forall q \in Q_P : x_q \leq p_q \quad (6.1) \]
The idea of the algorithm is to start from the assumption
\[ PC_P \leq \sum_{q \in Q_P} \ell_q p_q \quad (6.2) \]
and then iteratively improve the bound by propagating the structural constraints through the program. It relies on the following two observations:

- A program point can never be visited more times than the sum of its immediate predecessors.
- A program point can never be visited more times than the sum of its immediate successors.

or formally,
\[ x_q \leq \sum_{q' \in \text{pred}(q)} x_{q'} \quad (6.3) \]
\[ x_q \leq \sum_{q' \in \text{succ}(q)} x_{q'} \quad (6.4) \]

where we define
\[ \text{pred}(q) = \{ \langle n, n_0 \rangle \in Q_P \mid n_0 = \text{from}(q) \} \]
\[ \text{succ}(q) = \{ \langle n_0, n \rangle \in Q_P \mid n_0 = \text{to}(q) \} \]

Note here that these are inequalities rather than equalities, since an incoming arc to a successor of a program point may cause the successor to be visited more
times the program point (see Figure 6.1). Now, the execution count $x_q$ for each program point has three upper bounds: (6.1), (6.3) and (6.4). Obviously the smallest one of these is the tightest and best one. The idea of MPA is to use (6.2) as basis, but to substitute $t_q$ for $p_q$ where $t_q$ is an upper bound computed from (6.1), (6.3) and (6.4). The upper bound $t_q$ is computed by propagating the upper bounds through the graph and construct a tree which represents the upper bound.

![Figure 6.1: The program point $q_3$ can be visited more times than $q_1$ even though $q_3$ is the only successor of $q_1$](image)

### 6.2.1 The Min-Tree

The upper bound for a program point needs to be valid for all possible combinations of parameters $p_q \in Q$. An upper bound $t_q$ will be represented as a tree with three types of nodes: minimum nodes, plus nodes and leaf nodes. Minimum nodes (denoted ♢) express the minimum of all its children. Plus nodes (denoted ⊕) express the sum of all its children. Leaf nodes (denoted $p_q$) express the value of $p_q$. Such a tree will be referred to as a Min-Tree. Figure 6.2 depicts an example of a Min-Tree. This tree is in fact representing the upper bound of $x_0$ in $L$ (see Figure 3.2, on page 27).

### 6.2.2 The Algorithm

MPA is shown in Algorithm 1. It is a recursive procedure which takes as argument a program point, a context and a set of constraints and returns a Min-Tree as described in previous section. The context is a set of visited program points for internal book keeping. The algorithm is always called with the empty set as context when used. The set of constraints corresponds to that of (6.3) and (6.4) and is obtained directly from the graph structure. The constraint (6.1) is
Algorithm 1 MPA($q_i$, context, constraints)

1: node ← mkMinNode()
2: worklist ← push(NIL, $i$)
3: branch ← $\emptyset$
4: while worklist $\neq$ NIL do
5:  $k$ ← peek(worklist)
6:  worklist ← pop(worklist)
7:  if $k \notin$ context then
8:     context ← context $\cup \{k\}$
9:     node ← addLeaf(node, $p_k$)
10:    for all $[x_k \leq x_j] \in$ constraints do
11:       if $j \notin$ context then
12:          worklist ← push(worklist, $j$)
13:     end if
14:  end for
15:  for all $[x_k \leq \sum_{n \in N} x_n] \in$ constraints such that $|N| \geq 2$ do
16:    if $N \cap$ context $= \emptyset$ then
17:       branch ← branch $\cup \{N\}$
18:  end if
19: end for
20: end if
21: end while
22: for all $N \in$ branch do
23:  plusNode ← mkPlusNode()
24:  for all $n \in N$ do
25:    child ← MPA($n$, context, constraints)
26:   plusNode ← addChild(plusNode, child)
27:  end for
28: node ← addChild(node, plusNode)
29: end for
30: return node
Algorithm 1

\[ \text{MPA}(q_i, \text{context}, \text{constraints}) \]

1: \[ \text{node} \leftarrow \text{mkMinNode}() \]
2: \[ \text{worklist} \leftarrow \text{push}(\text{NIL}, i) \]
3: \[ \text{branch} \leftarrow \emptyset \]
4: while \( \text{worklist} \neq \text{NIL} \) do
5: \[ k \leftarrow \text{peek} (\text{worklist}) \]
6: \[ \text{worklist} \leftarrow \text{pop} (\text{worklist}) \]
7: if \( k \in \text{context} \) then
8: \[ \text{context} \leftarrow \text{context} \cup \{k\} \]
9: \[ \text{node} \leftarrow \text{addLeaf}(\text{node}, p_k) \]
10: for all \( x \leq x_j \in \text{constraints} \) do
11: if \( j \in \text{context} \) then
12: \[ \text{worklist} \leftarrow \text{push}(\text{worklist}, j) \]
13: end if
14: end for
15: for all \( x \leq \sum n \in N x_n \in \text{constraints} \) such that \( |N| \geq 2 \) do
16: if \( N \cap \text{context} = \emptyset \) then
17: \[ \text{branch} \leftarrow \text{branch} \cup \{N\} \]
18: end if
19: end for
20: end if
21: end while
22: for all \( N \in \text{branch} \) do
23: \[ \text{plusNode} \leftarrow \text{mkPlusNode}() \]
24: for all \( n \in N \) do
25: \[ \text{child} \leftarrow \text{MPA}(n, \text{context}, \text{constraints}) \]
26: \[ \text{plusNode} \leftarrow \text{addChild}(\text{plusNode}, \text{child}) \]
27: end for
28: \[ \text{node} \leftarrow \text{addChild}(\text{node}, \text{plusNode}) \]
29: end for
30: return \( \text{node} \)

Figure 6.2: A Min-Tree representing the formula \( \min(p_0, p_1, p_3 + \min(p_4, p_5), p_2) \).

Implicit and is not needed in the constraint argument of the algorithm. MPA searches the given constraints and recursively builds a Min-Tree by adding visited nodes as children to a minimum node. It searches all simple paths first, leaving the branches for later. The branches are then recursively computed as children for plus nodes.

The root of the Min-Tree will always be a minimum node, and its children will be all maximum bounds found for the program point under analysis. MPA maintains a worklist and a branch set; the worklist keeps track of visited program points and the branch set keeps track of pending plus nodes. Whenever a program point has single predecessors and successors, the neighbouring capacities alone constitute upper bounds for the program point and are therefore put in the worklist for continued processing. In the case of branching program points, the program points are put in the pending branch set for recursive processing as children of a plus node. The reason for this can be read directly from (6.3) and (6.4), where it is obvious that it is the sum of the upper bounds of the other program points that needs to be computed.
Detailed Explanation of the Algorithm

Row 1 creates the root of the tree which is always a minimum node, the primitive mkMinNode returns a minimum node without children. The Rows 2-3 initialise the worklist and the branch set. The worklist is implemented as a stack and using the stack primitives push, pop and peek (peek returns the top element of the stack, pop returns the stack with the top element removed) to manipulate it. The loop in rows 4-21 builds the leaves of the minimum node and puts the pending plus nodes in the branch set. Row 7 ensures that nodes which have already been considered (and thus don’t contribute to any tighter result) are skipped. Row 9 adds leaves to the minimum node by using the primitive addLeaf which takes a node and a leaf and returns the node with the leaf added. Then, in rows 10-14, all single entry/exit constraints are added to the worklist for further processing. Rows 15-19 add the multiple entry/exit constraints to the pending branch set.

When no more program points are present in the worklist, the algorithm has added all leaves to the current min node, and enters the part of the algorithm which builds the plus nodes (row 22). By now, node is a minimum node, possibly with a couple of leaves, which are all maximum bounds on the program point \( i \). In other words, the constraints from (6.1) have been added. Left to add are the plus nodes, which correspond to (6.3) and (6.4). This is done in rows 16-23. Each constraint which is corresponding to a branch in the program (i.e., a constraint which is a sum of program points) will produce a plus node (row 23), this is done by the primitive mkPlusNode which simply returns a plus node without children. The children of the plus node are then recursively computed from each term in the constraint (row 25), and then added as children to the plus node via the primitive addChild (row 26). Finally, each plus node is added as a child of the minimum node (row 28) and the root node is returned (row 30).

6.2.3 Example of MPA

Consider the example program \( L \) in Figure 3.2. We will show how to compute a Min-Tree for \( q_0 \). The set of constraints obtained from (6.1), (6.3) and (6.4)
are the following

\[ \forall q \in Q_L : x_q \leq p_q \]

\[ x_0 \leq x_1 \]

\[ x_1 \leq x_2, x_0 \]

\[ x_2 \leq x_1 + x_5, x_3 + x_4 \]

\[ x_3 \leq x_2 \]

\[ x_4 \leq x_2, x_5 \]

\[ x_5 \leq x_4, x_2. \]

We start by calling \( \text{MPA}(q_0, \emptyset, \text{constraints}) \). Processing in row 4-21 will generate the following intermediate results:

<table>
<thead>
<tr>
<th>node</th>
<th>worklist</th>
<th>branch</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{min()}</td>
<td>[0]</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>\text{min}(p_0)</td>
<td>[1]</td>
<td>\emptyset</td>
<td>{0}</td>
</tr>
<tr>
<td>\text{min}(p_0,p_1)</td>
<td>[2]</td>
<td>\emptyset</td>
<td>{0,1}</td>
</tr>
<tr>
<td>\text{min}(p_0,p_1,p_2)</td>
<td>[]</td>
<td>{3,4}</td>
<td>{0,1,2}</td>
</tr>
</tbody>
</table>

After the worklist has become empty and the main loop has finished, the algorithm is in row 22 and the plus nodes will be evaluated. We have that \( N = \{3,4\} \) and so this leads to two recursive calls: \( \text{MPA}(q_3, \{0,1,2\}, \text{constraints}) \) and \( \text{MPA}(q_3, \{0,1,2\}, \text{constraints}) \). The following tables show the intermediate results for these calls.

<table>
<thead>
<tr>
<th>node</th>
<th>worklist</th>
<th>branch</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{min()}</td>
<td>[0]</td>
<td>\emptyset</td>
<td>{0,1,2}</td>
</tr>
<tr>
<td>\text{min}(p_3)</td>
<td>[1]</td>
<td>\emptyset</td>
<td>{0,1,2,3}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>worklist</th>
<th>branch</th>
<th>context</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{min()}</td>
<td>[4]</td>
<td>\emptyset</td>
<td>{0,1,2}</td>
</tr>
<tr>
<td>\text{min}(p_4)</td>
<td>[5]</td>
<td>\emptyset</td>
<td>{0,1,2,4}</td>
</tr>
<tr>
<td>\text{min}(p_4,p_5)</td>
<td>[]</td>
<td>\emptyset</td>
<td>{0,1,2,4,5}</td>
</tr>
</tbody>
</table>
The result of these two calls will both be children to a plus node, which in turn will be child to the minimum node that will be returned from the original call. This plus node is then added as child to the previous minimum node. The final Min-Tree for $q_0$ expresses:

$$\min(p_0, p_1, p_2, \min(p_3) + \min(p_4, p_5)),$$

Computing the Min-Tree $t_q$ for all program points $q \in Q_L$ results in

$$

t_0 = \min(p_0, p_1, p_3 + \min(p_4, p_5), p_2) \\
\quad \quad t_1 = \min(p_0, p_1, p_3 + \min(p_4, p_5), p_2) \\
\quad \quad t_2 = \min(p_2, \min(p_0, p_1) + \min(p_4, p_5), p_3 + \min(p_4, p_5)) \\
\quad \quad t_3 = \min(p_2, p_3, \min(p_0, p_1) + \min(p_4, p_5)) \\
\quad \quad t_4 = \min(p_2, p_4, p_5) \\
\quad \quad t_5 = \min(p_2, p_4, p_5).
$$

The function $PC_L$ is then computed by taking

$$
PC_L = \sum_{q \in Q_L} \ell_L(q) t_q,
$$

where

$$
PC_L = \lambda p_0, p_1, p_2, p_3, p_4, p_5.
\min(p_0, p_1, p_3 + \min(p_4, p_5), p_2) \\
\quad + 3(\min(p_0, p_1, p_3 + \min(p_4, p_5), p_2)) \\
\quad + \min(p_2, \min(p_0, p_1) + \min(p_4, p_5), p_3 + \min(p_4, p_5)) \\
\quad + 2(\min(p_2, p_3, \min(p_0, p_1) + \min(p_4, p_5)) \\
\quad \quad + 2(\min(p_2, p_4, p_5)) \\
\quad \quad + 8(\min(p_2, p_4, p_5)).
$$

Computing $PC_L \circ SC_L$ (by substituting $SC_L(\sigma, q_k)$ for each parameter $p_k$) will result in
\[ \text{PWCET}_L = \lambda \sigma. \]
\[
\min(1, 1 + (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0), \infty) \\
+3(\min(1, 1 + (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0), \infty)) \\
+\min(\infty, 1 + (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0)) \\
+2(\min(\infty, 1 + (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0))) \\
+2(\min(\infty, (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0))) \\
+8(\min(\infty, (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0))) \\
= \lambda \sigma. \\
4(\min(1, 1 + (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0))) \\
+1 + (\sigma(n) \geq 0 ? \sigma(n) + 1 : 0) \\
+2 + 10(\sigma(n) \geq 0 ? \sigma(n) + 1 : 0) \\
= \lambda \sigma.7 + 11(\sigma(n) \geq 0 ? \sigma(n) + 1 : 0) \\
= \lambda \sigma. \begin{cases} 18 + 11\sigma(n) & \text{if } \sigma(n) \geq 0 \\ 7 & \text{otherwise.} \end{cases}
\]

which equals PWCET\(_L\) that was obtained by parametric integer programming in this case.

## 6.3 Properties of MPA

This section investigates different properties of MPA. In particular, the algorithm is proven to terminate and to be correct. A bound on the complexity of the algorithm is also given. MPA contains five loops referred to as \(L_4, L_{10}, L_{15}, L_{22}\) and \(L_{24}\), where the subscript is the row number of the loop header in Algorithm 1.

### 6.3.1 Termination

In order to prove that MPA terminates, we show that the recursion and all five loops of MPA terminate. First, we see that the loops \(L_{10}\) and \(L_{15}\) terminate because they iterate over stable and finite sets. For the same reason, the loops \(L_{22}\) and \(L_{24}\) terminate \emph{provided} that the recursive calls to MPA all terminate. The fact that \emph{branch} is a finite set is a consequence of that \(L_4\) terminates, which
is shown below. This means that \( L_4 \) executes a finite number of times and thus adding a finite number of elements to \( \text{branch} \).

In each iteration of \( L_4 \) exactly one of the following holds:

1. The set \( \text{context} \) will contain one more element than in the previous iteration.

2. The list \( \text{worklist} \) will contain one less element than in the previous iteration.

Row 7 will execute in every iteration of \( L_4 \). If the conditional evaluates to \( \text{true} \), then \( k \) was not part of \( \text{context} \) and will on row 8 be added. Thus, \( \text{context} \) contains one more element than the previous iteration. No more elements are added to \( \text{context} \) in the loop. If the conditional in row 7 evaluates to \( \text{false} \), no elements will be added to \( \text{worklist} \) (row 12 can not execute). Since row 6 will be executed in any case, the list \( \text{worklist} \) will contain one element less.

By definition, \( \text{context} \subseteq P \), and if \( \text{context}_i = Q \), then \( L_4 \) would terminate immediately. This is because the conditional on row 7 will evaluate to \( \text{false} \) and continue to loop \( L_{22} \). Now, \( \text{branch} = \emptyset \), since row 3 was the last assignment of \( \text{branch} \). Thus, \( L_{22} \) will terminate immediately and \( L_{24} \) will never execute. Now, assume for contradiction that there exists an input \( I = (q_i, \text{context}_i, \text{constraints}_i) \) such that \( L_4 \) does not terminate. Thus, \( \text{context}_i \subseteq P \). Statement 1 in the list above may only occur a finite number of times since \( Q_P \) is a finite set. This means that there exists an infinite sequence of iterations such that statement 2 occur. However, \( \text{worklist} \) is a finite set (since a finite number of elements can be added to it a finite number of times), meaning that \( \text{worklist} \) in this infinite sequence of iterations will become empty, but the conditional on row 4 terminates \( L_4 \) as \( \text{worklist} \) is empty. This contradicts that there exists an input such that \( L_4 \) never terminates.

Left to prove is that the recursive calls to MPA eventually terminate. If MPA is called with \( \text{context}_i \subset Q_P \) as input, then MPA may recursively call MPA a finite number of times with \( \text{context}_j \) as input. We want to show that for every recursive call \( \text{context}_i \subset \text{context}_j \) (i.e., the recursive calls to MPA are called with a context with at least one more element than the current call), which would imply that in a finite number of steps MPA will either have terminated or will be called with \( \text{context}_j = Q_P \), which also means that MPA will terminate. First, note that row 25 is executed only if \( N \) is non-empty. The set \( N \) is non-empty only if \( \text{branch} \) is non-empty. Finally, \( \text{branch} \) is non-empty only if row 17 executes, when the conditional on row 7 evaluates to \( \text{true} \), in which case row 8 executes, meaning that \( \text{context} \) has at least one more ele-
6.3 Properties of MPA

6.3.2 Complexity

The complexity of MPA involves a lot of factors. In fact the complexity depends more on the structure of the input program than on the actual size of the program (i.e., the number of program points). While it is possible to derive a worst-case complexity of MPA in terms of the number of program points of a program, it would not be a very useful one. To investigate the complexity of MPA we take an informal alternative approach in which we investigate the behaviour of MPA rather than the algorithm itself. All primitives used in MPA can be implemented so that they take constant time. The first thing that needs to be said about the complexity of MPA is that $L_{10}$ and $L_{15}$ actually do not need to be implemented as loops. This is because there are maximum two constraints associated with a single program point (incoming and outgoing edges). Thus, the constraints can be stored in such a way that each program point can access two constraints, meaning that it is not necessary to loop through all constraints in $L_{10}$ and $L_{15}$. Consequently, we consider $L_{10}$ and $L_{15}$ to be $O(1)^1$.

Now, the loop $L_4$ iterates through all program points found in any non-branching path in both directions (row 10-12). All branching paths are put into branch, which then recursively call MPA for every branching path (row 25), in a recursive call a previously explored path will not be explored again, since previously explored edges will be stored in context. However, an edge may be explored several times in different recursive calls of MPA. In summary, this means that MPA will explore every non-cyclic path of the flow chart. Thus, the complexity of MPA is directly proportional to the number of non-cyclic paths in the analysed program.

6.3.3 Correctness of MPA

In this section we will prove that MPA is correct. By correct we mean that edge $q$ is guaranteed to be visited less than or equal to the expression represented by the Min-Tree returned by $\text{MPA}(q, \varnothing, \text{constraints})$. This will be proven by induction over the depth of the MPA tree.

---

1although the arrangement of constraints would require $O(n)$ of memory, where $n$ is the size of $Q_P$
Proposition 9. Let \( q \) be a program point \( q \in Q_P \) and let \( t \) be a Min-Tree produced by calling Algorithm 1 for program point \( q \) with an empty context. Assume that \( t \) consists only of a set of leaf nodes \( L_t \) and no plus nodes. Then \( t \) is a correct upper bound of the number of times \( q \) is visited.

The first level of the Min-Tree is the tree produced before any recursive calls to MPA is performed. Thus, a one-level Min-Tree consists only in a Min-Node and a set of leaf nodes.

Proof. All leaf nodes are added by row 9 of Algorithm 1. The first node added is the constraint \( x_q \leq p_q \) which is directly taken from (6.1). Then, the constraints \( x_q \leq p_r \) where \( r \) is any program point which may be in the worklist. The program points that may be in the worklist are the neighbours to \( q \) (including \( q \) itself) which must be or must have been visited when \( q \) is visited (see row 10). All nodes in this set of program points must be visited the same number of times, say \( m \). Since all nodes in this set must be visited every time \( q \) is visited, the least capacity of these nodes constitutes maximum bound on \( m \).

This means that it is safe to terminate the algorithm before any recursive calls. However, to reach a potentially tighter result, the recursive calls may contribute to a tighter, yet still correct bound.

Proposition 10. Let \( q \) be a program point \( q \in Q_P \) and let \( t \) be the Min-Tree produced by calling Algorithm 1 for program point \( q \) with an empty context. Assume that \( t \) consists of a (possibly empty) set of leaf nodes \( L_t \), and a set of plus nodes \( P_t \), where each plus node \( p \in P_t \) has a set of \( n \)-level Min-Trees. Then, if all \( n \)-level Min-Trees are correct (that is, they constitute correct upper bounds of the program points they represent), then \( t \) is an \( n + 1 \)-level Min-Tree representing a correct upper bound of the number of times \( q \) is visited.

Proof. First of all, the minimum of the set of leaf nodes \( L_t \) is a correct upper bound for \( q \) as stated by Proposition 9. Now, every set of program points \( N \) in the branch set (see row 17) represents a selection of edges in the flow chart, that is, exactly one of the program points in \( N \) will be taken for every time program point \( q \) is visited. This means that the sum of all upper bounds of the program points in \( N \) is an upper bound also on the number of times \( q \) can be visited. Assuming that all \( n \)-level Min-Trees produced by MPA are correct, this also corresponds to a correct upper bound of the number of times \( q \) is visited. The proposition holds since the minimum of a set of correct upper bounds (i.e., the bound derived by the leaf nodes and the bound derived by the plus nodes) are again a correct upper bound.
Proposition 9 and Proposition 10 together proves that MPA produces correct bounds.

### 6.3.4 Upper Bounds on Tree Depth

An important consequence of the arguments in Section 6.3.3 is that every level of the Min-Tree is a safe upper bound of the execution time of a program point. This means that it is safe to skip the computation of any subtree in the Min-Tree, although it may result in a less precise sub tree. Thus, it is possible to set an upper bound on the depth of the produced Min-Trees to ensure a faster termination of the MPA algorithm, to the cost of possible precision loss. However, the deeper a node is in a Min-Tree, the less likely it is to actually contribute to a tighter upper bound. This is because subtrees are children to plus nodes, which in turn probably will give a larger bound than the children of Min-nodes, and since the root node is always a Min-node, the larger nodes will not contribute to the final solution.

To summarise: the first levels in the Min-Tree are the nodes most likely to contribute to the final results. Thus, computing very deep sub-trees will in many cases be a waste of computation time. In Section 9.2.2, we will show how setting an upper bound of the depth of Min-Trees affects computation time and precision. Chapter 9 also show how MPA compares with parametric integer programming.
Chapter 7

Fully Bounded Polyhedra

7.1 Analysis on Low-level Code

For a WCET analysis to be accurate it needs to model the execution of a program very accurately. This means that analysing source code is usually not the best way of estimating the WCET since it is not the source code itself that is being executed.

The static analysis tool SWEET, in which we have implemented our parametric method, analyses code in an intermediate level format called ALF [53, 52] (see details in Chapter 8). This format can be generated both from source code and from machine code, meaning that it does not necessarily contain the high-level constructs found in source code.

Balakrishnan and Reps invented the term WYSINWYX (What You See Is Not What You eXecute) [12], which says that there are many fallacies involved in trusting a source code level analysis. The term is referring to the fact that what is actually being executed on a machine, the machine code, may behave significantly different from the source code due to compiler optimisations or target platform properties. Thus, especially in safety critical software, an analysis made on the source code may be misleading. Optimisations such as removal of dead code, loop unrolling or parallelisation also affects the flow of the program. Moreover, the executable may contain information which simply is not present in source code form. This includes which values reside in registers, and information about contents of the cache or pipeline. This kind of information is very important in analyses which extract timing behaviour from programs.

On source level, a program may also lack full semantic meaning. For example...
Chapter 7

Fully Bounded Polyhedra

7.1 Analysis on Low-level Code

For a WCET analysis to be accurate it needs to model the execution of a program very accurately. This means that analysing source code is usually not the best way of estimating the WCET since it is not the source code itself that is being executed.

The static analysis tool SWEET, in which we have implemented our parametric method, analyses code in an intermediate level format called ALF [53, 52] (see details in Chapter 8). This format can be generated both from source code and from machine code, meaning that it does not necessarily contain the high-level constructs found in source code.

Balakrishnan and Reps invented the term WYSINWYX (What You See Is Not What You eXecute) [12], which says that are many fallacies involved in trusting a source code level analysis. The term is referring to the fact that what is actually being executed on a machine, the machine code, may behave significantly different from the source code due to compiler optimisations or target platform properties. Thus, especially in safety critical software, an analysis made on the source code may be misleading. Optimisations such as removal of dead code, loop unrolling or parallelisation also affects the flow of the program.

Moreover, the executable may contain information which simply is not present in source code form. This includes which values reside in registers, and information about contents of the cache or pipeline. This kind of information is very important in analyses which extract timing behaviour from programs.

On source level, a program may also lack full semantic meaning. For ex-
ample, a computation leading to an overflow have an undefined behaviour at
the (target independent) source level. On the other hand, executable code al-
ways have exact semantics for every instruction. Thus, code where undefined
behaviour is not present allows a more precise analysis.

To summarise, recently there have been an increasing interest in analysing
executables rather than source code. The program model introduced in Chap-
ter 3 is fairly generic and could describe low-level or executable code with
some modifications. We have made a survey of the problems involved when
analysing low-level or executable code in [22], and what follows is a summary
of these issues, details can be found in the survey.

Control Flow Most static analyses rely on a graph representation of the pro-
gram flow. In other words, the control flow is assumed to be (mostly)
statically known. However, in executable code, the program flow typi-
cally cannot be extracted precisely. This is due to dynamic control flow
such as indirect jumps. The most common approach to control flow is
to re-build the control flow graph from scratch. This is usually done
bottom-up by reading and decoding one instruction at a time and build-
ing edges where jumps are found, as in [111, 63, 67].

Memory Model As seen in Chapter 4 abstract interpretation can be used to
find bounds and relation between the values of variables. However, in
executable code, values may reside in registers, or be loaded from var-
ious memory locations. Moreover, addresses might be computed using
arithmetic, several id’s may refer to the same value (aliasing) and the
concept of “a variable” is in general vague or not present. This is a prob-
lem, especially for analyses which attempt to maintain relations between
variables such as polyhedral abstract interpretation. Solutions to this
problem may involve modeling the memory exactly if it is small [19],
resolving aliasing issues with abstract interpretation [37, 38], or having
an analysis identifying variables [10, 11, 99].

Bit-Precise Modelling Subtle bugs even on source level can be due to what
happens on the bit-level in compiled code. To accommodate for this
there have been suggestions for abstract domains operating on the bit-
level [68, 19].

Integer Representation It is often convenient to formulate value analyses math-
ematically and assume that integer-valued variables assume values from
the set of mathematical (unbounded) integers (in fact, the most com-
monly used abstract domains had this assumption in their original presentation [34, 45, 46, 66, 56]). However, in practice, at least on executable code, integers are stored as binary strings of a fixed length. Because of this, integers that are bounded may cause arithmetic operations to cause overflows. Moreover, binary strings can often be interpreted as signed or unsigned. This is often ignored or assumed not to be a problem in source level analysis. However, many studies have indicated the importance of accurate modelling of integers.

Among these problems we have focused on solving the program of integer representation. Our contribution is described in this chapter.

7.2 Representing Integers

Our program model introduced in Chapter 3 defines environments as a function of variable names to integers from the set \( \mathbb{Z} \). When analysing real-time embedded software, however, variables are often represented by a finite string of bits, usually 64, 32, 16 or 8 bits long. This means that variables cannot take the value of an arbitrary integer, rather they take values in the sets \( \mathbb{N}_{64}^2, \mathbb{N}_{32}^2, \mathbb{N}_{16}^2 \) or \( \mathbb{N}_{8}^2 \) (i.e., bit-strings of the given length).

This means that there is a mismatch between the analysis results and the actual program under analysis. This mismatch can potentially make analysis results unsafe. For WCET analysis safety is typically a requirement which is why this chapter investigates how to make polyhedral analysis safe even in this context.

7.3 Introduction and Related Work

Our initial aim was to develop an abstract interpretation that can be used with the Census method that is safe even when program variables are represented by bit-strings. In [105], Sen and Srikant present a combination of an abstract of congruences and integers which handles special cases when overflow occurs. They also use a relational analysis of affine equalities to make it relational. Gustafsson et al. [50] modify the abstract domain of intervals (see [34]) so that variables are bounded to within their size-induced range, and wrap-arounds are handled by using a more powerful representation of intervals. Müller-Olm and Seidl [85] present an analysis that can derive all affine equalities (but not
inequalities) among variables of programs which is safe in the case of wrap-arounds. Brauer and King [18] suggest a method to derive transfer functions for relational domains, and do so for the octagon domain (see [82]), while considering wrap-around effects by using a SAT-solver. Finally, Simon and King [106] present a way to use classical polyhedral analysis in a way that is safe for programs with wrap-around semantics for integers.

Since the polyhedral analysis works very well for the Census method, that is, it is precise and relational, Simon and King’s method works very well in our approach to parametric WCET analysis. However, the method has some problems with precision. In this chapter we present our contribution by developing an analysis which is based on Simon and King’s method but which increases its precision. We call the method fully bounded polyhedra. Since it builds on Simon and King’s method, we start by introducing their method in Section 7.4 and develop our method in Section 7.5.

## 7.4 Simon and King’s Method

Simon and King assumes in their method [106] that each program variable has a size and a type; the size is the number of bits used to store the integer and the type is whether the value of the variable is to be interpreted as signed or unsigned. From this information we can associate a set of constraints for each variable.

**Definition 29.** The function $R : \text{VAR}_P \to \mathbb{C}$ is associating each program variable $v$ with a $\mathcal{H}$-polyhedron called the boundary polyhedron $R(v) = \{l \leq v, v, v \leq u\}$ dictated by the size and interpretation of $v$.

For example, if $v_0$ is an 8-bit signed variable, then the boundry polyhedron for $v_0$ is $R(v_0) = \{-128 \leq v_0, v_0 \leq 127\}$. We define the boundry polyhedron for a set of variables $A \subseteq \text{VAR}_P$ as

$$R(A) = \bigcup_{v \in A} R(v)$$

It is assumed that the values for any set of variables are within the boundry polyhedron, that is $v, l \leq \sigma(v) \leq u$ where $R(v) = \{l \leq v, v \leq u\}$. The set $R(\text{VAR}_P)$, which is the boundry polyhedron for all variables of $P$, is called the base window. The concept of the base window is important because it represents an invariant environment $\sigma$ that can occur during any execution.
Next we define a specific modulo operator on integers. Let \( x \in \mathbb{Z} \) and \( v \in \text{VAR}_P \), then

\[
x \mod v = l + (x \mod (u - l))
\]

where \( l, u \) comes from \( R(v) = \{ l \leq v, v \leq u \} \). For example, if \( R(v_0) = \{-128 \leq v_0, v_0 \leq 127\} \), then \( 129 \mod R(v_0) = -128 + (129 \mod 255) = 1 \). As can be seen, this modulo operation simulates the wrap-around effect when value \( x \) is stored in variable \( v \). This operation can lifted to a point \( x \in \mathbb{Z}^n \) modulo a set of variables \( S \):

\[
\langle x_0, \ldots, x_{n-1} \rangle \mod S = \langle x_0 \mod s_0, \ldots, x_{n-1} \mod s_{n-1} \rangle
\]

where

\[
s_i = \begin{cases} v_i & \text{if } v_i \in S \\ \infty & \text{if } v_i \notin S \end{cases}
\]

where the definition \( x \mod \infty = x \) is used.

### 7.4.1 Implicit wrapping

Simon and King define the concretisation function \( \gamma_{SK} \) as

\[
\gamma_{SK}(P) = \{ x \mod V | x \in \gamma_{CP}(P) \}
\]

This interpretation does not affect the actual analysis in any way, but correctly interprets all values to be within the base window. Note that the points in \( \gamma_{SK}(P) \) will not necessarily form a convex polyhedron. This implicit wrapping works for the analysis of equality-based linear transfer functions, such as linear assignments, because linear transformations commute with modulo. However, polyhedral abstract interpretation also includes intersection with linear constraints at conditionals. Intersecting with a linear constraint does not commute with the modulo operation and therefore Simon and King introduced an explicit wrapping procedure which has to be performed at conditionals.

### 7.4.2 Explicit Wrapping

The idea with the explicit wrapping is to make sure that the abstract environment (i.e., the polyhedron) is within the base window before the linear constraint is intersected with the polyhedron.

Let \( P \) be a convex polyhedron and let \( b \) be a linear constraint involving the set of variables \( A_b \). Intuitively, the wrapping procedure consist of first
partitioning the subspace of \( \mathbb{Z}^n \) involving the variables \( A_b \) into a finite grid of windows of the same size and dimension as \( R(A_b) \) as shown in Figure 7.1(a). If any variable \( v \in A_b \) is unbounded in \( \mathcal{P} \) (meaning that the frame representation of \( \mathcal{P} \) contains a ray or a line in the dimension represented by variable \( v \in A_b \)) then the known information about \( v \) is discarded in \( \mathcal{P} \), by the operation \( \mathcal{P}[v \leftarrow \ ?] \cap \mathbb{P}_{\mathcal{H}}(R(v)) \).

Then each partition, which is the size of the base window, is shifted to the position of the base window and intersected by \( b \). Finally, the convex hull of all these shifted and intersected partitions is taken as the result (see Figure 7.1(b)). The procedure is explained in detail in [106]. The result of wrapping \( \mathcal{P} \) with constraint \( b \) and a set of variables \( A \) is denoted as follows:

\[
\text{wrap}(\mathcal{P}, b, A)
\]

It is important to notice that explicit wrapping potentially loses precision of \( \mathcal{P} \). This is for two reasons: computing the convex hull is naturally approximate and the elimination of unbounded variables in the condition may lose relational information. Thus, explicit wrapping should be applied only when necessary.

### 7.5 Fully Bounded Polyhedra

In this section we present our method of bounded polyhedral analysis. Our idea was first presented in [26]. Our approach is an extension of Simon and King’s method: we use the same model for implicit and explicit wrapping.

The main idea of our approach is to reduce the imprecision introduced by explicit wrapping. Explicit wrapping may introduce loss of precision, especially for unbounded polyhedra. Thus, our approach is to make an analysis where unbounded polyhedra never occur.

### 7.6 Motivation and Illustrating Example

In this section we use the example program \( L' \) depicted in Figure 7.2. Assume that \( L' \) is executed on an 8-bit processor and let \( x \) and \( N \) be 8-bit unsigned integer variables so that \( R(\{x, N\}) = \{0 \leq x, N \leq 255\} \).

Classical polyhedral analysis to analyse \( L' \) uses the following equation system (we mark the equations \( \text{CP} \) to emphasize that this is achieved with classical
The procedure is explained in detail in [106]. The result of wrapping these shifted and intersected partitions is taken as the result (see Figure 7.1(b)).

Finally, the convex hull of all windows of the same size and dimension as ... This is for two reasons: computing the convex hull is naturally approximate (we mark the equations CP).

Then each partition, which is the size of the base window, is shifted to the position of the base window and intersected by any variable of P contains a ray or a line in the dimension represented by variable v. Thus, explicit wrapping potentially loses precision of information. Thus, explicit wrapping should be applied only when necessary.

In this section we use the example program depicted in Figure 7.2. Assume L′ uses the following equation system (7.1) to emphasize that this is achieved with classical polyhedral analysis (polyhedral analysis):

\[
\begin{align*}
\hat{CS}_{L'}^{CP}(\hat{s})(q_0) &= \mathbb{P}_H(\emptyset) \\
\hat{CS}_{L'}^{CP}(\hat{s})(q_1) &= \hat{s}(q_0)[x \leftarrow N] \\
\hat{CS}_{L'}^{CP}(\hat{s})(q_2) &= \hat{s}(q_2) \nabla (\hat{s}(q_1) \cup \hat{s}(q_5)) \\
\hat{CS}_{L'}^{CP}(\hat{s})(q_3) &= \hat{s}(q_2) \cap \mathbb{P}_H(\{x \leq 0\}) \\
\hat{CS}_{L'}^{CP}(\hat{s})(q_4) &= \hat{s}(q_2) \cap \mathbb{P}_H(\{x \geq 1\}) \\
\hat{CS}_{L'}^{CP}(\hat{s})(q_5) &= \hat{s}(q_4)[x \leftarrow x + 1]
\end{align*}
\]
and the following results:

\[
\begin{align*}
\hat{\mathcal{CS}}_{L'}^{CP}(q_0) &= \mathbb{P}{_H(\emptyset)} \\
\hat{\mathcal{CS}}_{L'}^{CP}(q_1) &= \mathbb{P}{_H(\{ x = N \})} \\
\hat{\mathcal{CS}}_{L'}^{CP}(q_2) &= \mathbb{P}{_H(\{ x \geq N \})} \\
\hat{\mathcal{CS}}_{L'}^{CP}(q_3) &= \mathbb{P}{_H(\{ x \geq N, x \leq 0 \})} \\
\hat{\mathcal{CS}}_{L'}^{CP}(q_4) &= \mathbb{P}{_H(\{ x \geq N, x \geq 1 \})} \\
\hat{\mathcal{CS}}_{L'}^{CP}(q_5) &= \mathbb{P}{_H(\{ x \geq N + 1, x \geq 1 \})}
\end{align*}
\]

The program \( L' \) terminates, regardless of the input \( N \) because \( x \) wraps around when \( 255 + 1 \) returns 0. However, from \( \hat{\mathcal{CS}}_{L'}^{CP}(q_3) \) above it is evident that unless \( N = 0 \), it is not possible for \( x \) to be 0 at \( q_3 \). This means that classical polyhedral analysis in this case deliver unsound results.

To make the result sound, we apply Simon and King’s wrapping operator, substituting the equations \( \hat{\mathcal{CS}}_{L'}^{SK} \) for \( \hat{\mathcal{CS}}_{L'}^{CP} \). The equations for \( q_0, q_1, q_2 \) and \( q_5 \) are equal to \( CP \), but for \( q_3 \) and \( q_4 \) we have:

\[
\begin{align*}
\hat{\mathcal{CS}}_{L'}^{SK}(q_3) &= \text{wrap}( \hat{s}(q_2), \{ x \leq 0 \}, \{ x \}) \\
\hat{\mathcal{CS}}_{L'}^{SK}(q_4) &= \text{wrap}( \hat{s}(q_2), \{ x \geq 1 \}, \{ x \})
\end{align*}
\]

The result of the iterates is shown in Table 7.1. The polyhedron of the final iterate of \( \hat{\mathcal{CS}}_{L'}^{SK}(q_3) \) correctly implies that \( N \) can be anything in the range
Table 7.1 Iterating abstract interpretation of program $L$ using wrapped polyhedra.

<table>
<thead>
<tr>
<th>$\text{CS}_{L'}^{SK}$</th>
<th>iter $n = 1$, $\mathbb{P}_H$</th>
<th>iter $n = 2$, $\mathbb{P}_H$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>${x = N}$</td>
<td>${x = N}$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>${x = N}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>${x = 0, 0 \leq N \leq 255}$</td>
<td>${x = 0, 0 \leq N \leq 255}$</td>
</tr>
<tr>
<td>$q_4$</td>
<td>${1 \leq x \leq 255, 0 \leq N \leq 255}$</td>
<td>${1 \leq x \leq 255, 0 \leq N \leq 255}$</td>
</tr>
<tr>
<td>$q_5$</td>
<td>${2 \leq x \leq 256, 0 \leq N \leq 255}$</td>
<td>${2 \leq x \leq 256, 0 \leq N \leq 255}$</td>
</tr>
</tbody>
</table>

0 to 255. However, during the process, the relational information between $x$ and $N$ has been lost. This is due to the explicit wrapping of the unbounded polyhedron $\widehat{\text{CS}}_{L'}^{SK}(q_2) = \mathbb{P}_H(\{x = N\})$ in the first iterate. Wrapping of an unbounded polyhedron discards the relational information between $x$ and $N$ in order to make a sound and safe wrapping. While this is a simple example that could have been avoided by imposing bounds on $x$ and $N$ at the start of the program, unbounded polyhedra are frequent (caused by any widening, non-linear assignment or unbounded initial state) and apparently make the wrapping algorithm lose a lot of precision. This has led us to devise a fully bounded polyhedral analysis. Our method uses limited widening, special placement of widening, and uses type information to impose bounds variables. To show the idea we sketch here how our method analyses $L'$. The equations for our method are the following:

\[
\begin{align*}
\widehat{\text{CS}}_{L'}^{BD}(q_0) &= \mathbb{P}_H(R(\text{VAR}_{L'})) \\
\widehat{\text{CS}}_{L'}^{BD}(q_1) &= \widehat{s}(q_0)[x \leftarrow N] \\
\widehat{\text{CS}}_{L'}^{BD}(q_2) &= \widehat{s}(q_1) \sqcup \widehat{s}(q_5) \\
\widehat{\text{CS}}_{L'}^{BD}(q_3) &= \text{wrap}(\widehat{s}(q_2), \{x \leq 0\}, \{x\}) \\
\widehat{\text{CS}}_{L'}^{BD}(q_4) &= \widehat{s}(q_4) \bigtriangleup R(\text{VAR}_{L'}) \cup \{x \geq 1\} \text{ wrap}(\widehat{s}(q_2), \{x \geq 1\}, \{x\}) \\
\widehat{\text{CS}}_{L'}^{BD}(q_5) &= \widehat{s}(q_4)[x \leftarrow x + 1]
\end{align*}
\]

The difference between BD and SK can be seen in three places. First, the initial program point $\widehat{\text{CS}}_{L'}^{BD}(q_0)$ bounds the variables according to their type. Second, the widening point has been moved to $q_4$, and finally, the widening has been replaced by limited widening (explained in Section 7.6.3).
All these polyhedra are fully bounded in each iteration. This means that it would be possible to represent their frames with sets of vertices only, no rays or lines are needed. Table 7.2 shows the iterates using the equations (7.4). The results are most conveniently presented as frames in this case. Since there are no rays or lines in this presentation, we just present a frame polyhedron as $P_{\mathcal{F}}(V)$ where $V$ is a set of vertices. Moreover, the first dimension corresponds to $x$ and the second to $N$. Note that the frame representation of $P_{\mathcal{H}}(R(\text{VAR}_{L'}))$ is $\{(0, 0), (0, 255), (255, 0), (255, 255)\}$.

In this example, our approach takes a few more iterations before stabilisation, but this result, while still sound w.r.t. wrap-arounds, is more precise than the previous approach. In particular, notice that in $P_{4}^{\text{BD}}(3)$ we have that $x$ remains zero, but $N$ can be any number between 0 and 255 (soundness), and we have kept the valuable relation between $x$ and $N$ in $P_{4}^{\text{BD}}(2)$, $P_{4}^{\text{BD}}(4)$ and $P_{4}^{\text{BD}}(5)$, as the polyhedra have triangular shapes. This information was not retained in $P_{2}^{\text{SK}}(2)$, $P_{2}^{\text{SK}}(4)$ or $P_{2}^{\text{SK}}(5)$.

### Table 7.2 Iterating abstract interpretation for $L'$ using fully bounded polyhedra.

<table>
<thead>
<tr>
<th>Iter</th>
<th>$P_{\mathcal{F}}$</th>
<th>Iter 2, $P_{\mathcal{F}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>${(0, 0), (0, 255), (255, 0), (255, 255)}$</td>
<td>${(0, 0), (0, 255), (255, 0), (255, 255)}$</td>
</tr>
<tr>
<td>q1</td>
<td>${(0, 0), (255, 255)}$</td>
<td>${(0, 0), (255, 255)}$</td>
</tr>
<tr>
<td>q2</td>
<td>${(0, 0), (255, 255)}$</td>
<td>${(0, 0), (255, 255), (2, 1), (256, 255)}$</td>
</tr>
<tr>
<td>q3</td>
<td>${(0, 0)}$</td>
<td>${(0, 0), (255, 255)}$</td>
</tr>
<tr>
<td>q4</td>
<td>${(1, 1), (255, 255)}$</td>
<td>${(1, 1), (255, 255), (255, 1)}$</td>
</tr>
<tr>
<td>q5</td>
<td>${(2, 1), (256, 255)}$</td>
<td>${(2, 1), (256, 255), (256, 1)}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter 3</th>
<th>$P_{\mathcal{F}}$</th>
<th>Iter 4, $P_{\mathcal{F}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>q0</td>
<td>${(0, 0), (0, 255), (255, 0), (255, 255)}$</td>
<td>${(0, 0), (0, 255), (255, 0), (255, 255)}$</td>
</tr>
<tr>
<td>q1</td>
<td>${(0, 0), (255, 255)}$</td>
<td>${(0, 0), (255, 255)}$</td>
</tr>
<tr>
<td>q2</td>
<td>${(0, 0), (255, 255), (256, 255), (256, 1)}$</td>
<td>${(0, 0), (255, 255), (256, 255), (256, 0)}$</td>
</tr>
<tr>
<td>q3</td>
<td>${(0, 0), (255, 255)}$</td>
<td>${(0, 0), (255, 255)}$</td>
</tr>
<tr>
<td>q4</td>
<td>${(1, 1), (255, 255), (255, 0)}$</td>
<td>${(1, 1), (255, 255), (255, 0)}$</td>
</tr>
<tr>
<td>q5</td>
<td>${(2, 1), (256, 255), (256, 0)}$</td>
<td>${(2, 1), (256, 255), (256, 0)}$</td>
</tr>
</tbody>
</table>

### 7.6.1 Unboundedness in Polyhedral Analysis

When performing polyhedral abstract interpretation on a program, either with the classical domain or with Simon and King’s method, a polyhedron may become unbounded in three cases: First, at the initial program point, where nothing is known about the program variables (in CP and SK it the initial abstract environment is $P_{\mathcal{H}}(\emptyset)$). Second, any non-linear assignment to a variable $x$ will eliminate any bounds on $x$ via the operation $P[x \leftarrow ?]$. Finally, the widening
operation, which removes constraints from a $H$-polyhedron which might cause it to become unbounded.

### 7.6.2 Making Polyhedra Bounded

We consider each of the possible ways of making a polyhedron unbounded and show how to soundly and precisely make it bounded.

**Entry point**

In abstract interpretation, a common assumption is that nothing is known about the values of program variables at the entry point of the program. In CP, this is represented by the polyhedron with no constraints $P_H(\emptyset)$. However, in SK and BD, each variable is associated with a type and a size. This means that at least the constraints $R(VAR_P)$ holds for any variable $v \in VAR_P$ at the start of the program. Consequently, in BD we assign the whole base window as the polyhedron at the entry point of the program. So if $q_0$ is the initial program point then:

$$CS^{BD}(\hat{s})(q_0) = R(VAR_P)$$

This is sound and more precise than CP and SK.

**Non-Linear Assignments**

In CP and SK, a non-linear assignment discards all information about the assigned variable by the operation $P[x \leftarrow?]$. However, even if $x$ is applied to a non-linear assignment, $x$ will still be bounded by the type and size constraints $R(x)$. Let $q = \langle n, n' \rangle$ and $q' = \langle n', n'' \rangle$ such that $n'$ is a basic block containing only a non-linear assignment to $x$, then

$$CS^{BD}(\hat{s})(q') = \hat{s}(q)[x \leftarrow?] \cap P_H(R(x))$$

Again, this is sound and more precise than SK and BD.

**Widening**

For a program whose flow chart contains cycles, widening is necessary to ensure termination of the analysis. In classical abstract interpretation, the widening is usually inserted immediately after a join-node. The classical widening operation removes unstable bounds from the polyhedron which often results
in an unbounded polyhedron. However, removing a constraint involving the variable \( x \) from a polyhedron does not necessarily destroy all relational information about \( x \) and other variables. For this reason it is not be safe to intersect \( R(x) \) after widening. This is because linear relationships might still be present outside the base window, and restricting the widened polyhedron to be within the base window would be unsound.

### 7.6.3 Making Widening Bounded

The standard widening operation, as mentioned, often makes polyhedra unbounded. However, by using limited widening in a specific way, it is possible to make the result of the widening both sound and fully bounded.

#### Limited Widening

Limited widening was suggested in [55]. The idea with limited widening is to have a set of constraints \( C \) and define limited widening \( \nabla_C \) as follows:

\[
P \nabla_C Q = (P \nabla Q) \cup \{ c \in C | P \subseteq \{ c \} \land Q \subseteq \{ c \} \}
\]

That is, the result of the widening is intersected with all constraints in \( C \) which hold in both \( P \) and \( Q \). It can be shown that this is a widening operation for any set of constraints \( C \). The set \( C \) is typically selected strategically for each program. Specifically, for convex polyhedra:

\[
P_1 \nabla_C P_2 = (P_1 \nabla P_2) \cap \mathbb{P}_H(\{ c \in C | P_1 \subseteq \mathbb{P}_H(\{ c \}) \land P_2 \subseteq \mathbb{P}_H(\{ c \}) \})
\]

Our idea is to use a limited widening such that \( C \subseteq R(VAR_P) \). Our goal is to be able to intersect the result of the widening with at least \( R(VAR_P) \), to make the polyhedron fully bounded.

#### Placement of the Widening

Since our goal is to safely intersect the polyhedron with the base window, we can see from the definition of limited widening that both arguments to the widening operator have to be fully contained within the base window. Fortunately, there are certain points in the program where this is always true: whenever an explicit wrapping occurs, i.e., at conditionals.

SK requires that polyhedra are explicitly wrapped when a polyhedron is intersected by a linear constraint at linear conditionals. Wrapping guarantees that
\texttt{wrap}(P, b, A) \subseteq R(A), \text{ thus if } A = \text{VAR}_P \text{ the result will be fully contained in the base window. For this reason, our strategy is to make sure widening is done in conjunction with explicit wrappings. To avoid unnecessary loss of precision, the widening points can be distributed so that each cycle in the flow charts contains exactly one widening point.}

Specifically, this means that widening is applied where the control flow takes a conditional jump due to evaluating the constraint \( b \) to true (if program flow enters a cycle from evaluating a conditional to false, it can still be seen as evaluating its negation to true). We don’t impose a strategy for the exact placement of widening points in the cycles, we only require that the widening is done at conditionals and exactly once per cycle.

Let \( q \) be a program point before a conditional, and let \( q' \) represent the program point where control flow enters a cycle from evaluating the condition \( b \) to true. If \( b \) is non-linear, \( b \) is considered to be always be true. Classical polyhedral analysis just adds the condition as a new constraint to the \( \mathcal{H} \)-polyhedron (widening is typically done elsewhere): In BD, this is where widening is done, in particular limited widening:

\[
\widehat{\text{CS}}^{\text{BD}}(\widehat{s})(q') = \widehat{s}(q') \nabla_{R(\text{VAR}_P) \cup \{b\}}(\text{wrap}(\widehat{s}(q), b, \text{VAR}_P))
\] (7.5)

Note that all variables in \( \text{VAR}_P \) will be wrapped to ensure that the resulting polyhedron is contained within the base window. This differs from SK, where only the variables \( A_b \) that appear in the condition \( b \) are wrapped. This potentially gives a less precise result than wrapping with \( A_b \), but it will always result in a fully bounded polyhedron.

To see that (7.5) always results in a fully bounded polyhedron whenever it is used in abstract interpretation, let

\[
S_0^{\text{BD}} = \lambda q.\bot = \lambda q.\mathbb{P}_F((\emptyset, \emptyset, \emptyset))
\]

\[
S_n^{\text{BD}} = \widehat{\text{CS}}^{\text{BD}}(S_{n-1}^{\text{BD}})
\]

that is, \( S_n^{\text{BD}} \) represents the \( n:\text{th} \) iteration of the abstract interpretation. Then,

**Proposition 11.** Let \( q \) be a program point preceding a conditional with a linear condition, and let \( q' \) be a successor program point where control flow enters a cycle after evaluating the linear condition \( b \) to true. Then,

\[
S_n^{\text{BD}}(q') \subseteq P_{\mathcal{H}}(R(\text{VAR}_P) \cup \{b\})
\]

for all \( n \geq 0 \).
Again we note that the cycle may be entered either from evaluating $b$ or $\neg b$ to true, meaning that $q'$ may either be the true branch or the false branch as long as it enters a cycle.

**Proof.** First, let $Q_n = S_{n-1}^{BD}(q')$ and $U_n = \text{wrap}(S^{BD}_n(q), b, \text{VAR}_P)$, so we have

$$S^{BD}_n(q') = Q_n \nabla_C U_n$$

where $C = R(\text{VAR}_P) \cup \{b\}$. We will prove by induction over $n$ that the proposition holds. Let $n = 1$, then $Q_1 = S_0^{BD}(q') = \bot$. The wrapping operator guarantees that all variables $\text{VAR}_P$ are within their respective range. Thus, $\forall c \in R(\text{VAR}_P) : U_m \subseteq \mathbb{P}_H(\{c\})$ for all $m > 0$. Also we have that $U_n \subseteq \mathbb{P}_H(\{b\})$, since the wrapping operation applies the condition after wrapping. Now we have

$$Q_1 \nabla_C U_1 = \bot \nabla_C U_1$$

$$= (\bot \nabla U_1) \cap \mathbb{P}_H(\{c \in C | \bot \subseteq \mathbb{P}_H(\{c\}) \land U_1 \subseteq \mathbb{P}_H(\{c\}))$$

$$= U_1 \cap \mathbb{P}_H(\{c \in C | \bot \subseteq \mathbb{P}_H(\{c\}) \land U_1 \subseteq \mathbb{P}_H(\{c\}))$$

$$= U_1 \cap \mathbb{P}_H(C) \subseteq \mathbb{P}_H(C)$$

where the last equation comes from the fact that $\bot \subseteq \mathbb{P}_H(\{c\})$ for any $c$ and $U_1 \subseteq \mathbb{P}_H(\{c\})$ has already been established for any $c \in C$. Thus, the proposition holds for $n = 1$. Now assume that $Q_n \nabla_C U_n \subseteq \mathbb{P}_H(C)$ holds, then

$$Q_{n+1} \nabla_C U_{n+1} = (Q_{n+1} \nabla U_{n+1}) \cap \{c \in C | Q_n \subseteq \mathbb{P}_H(\{c\}) \land U_n \subseteq \mathbb{P}_H(\{c\})\}$$

The inductive hypothesis says that $Q_n \subseteq \mathbb{P}_H(\{c\})$ and $U_n \subseteq \mathbb{P}_H(\{c\})$ for all $c \in C$, so again we have that $\{c \in C | Q_n \subseteq \mathbb{P}_H(\{c\}) \land U_n \subseteq \mathbb{P}_H(\{c\})\} = C$. So,

$$(Q_{n+1} \nabla U_{n+1}) \cap \{c \in C | Q_n \subseteq \mathbb{P}_H(\{c\}) \land U_n \subseteq \mathbb{P}_H(\{c\})\}$$

$$= (Q_{n+1} \nabla U_{n+1}) \cap \mathbb{P}_H(C) \subseteq \mathbb{P}_H(C)$$

Chapter 8 contains a section where the implementation of Simon and King’s method as well as our bounded polyhedra have been implemented into SWEET. An evaluation of our and Simon and King’s method is found in Chapter 9.
Chapter 8

Implementation

8.1 Introduction

This chapter describes our implementation of the methods described in this thesis. The methods been implemented mainly to be able to evaluate them and continue to build upon them. The methods have been implemented mainly in two separate applications. The first is a prototype to test the feasibility of the parametric WCET method outlined in Chapter 3-5, the second is an implementation into the static analysis tool SWEET which is a more mature analysis tool.

8.2 First Prototype

The first prototype is implemented as a series of independent modules communicating via text-files. The modules were implemented using Haskell, C++ and third-party libraries/tools. The prototype uses a simple input language which is a subset of C for programs to analyse.

8.2.1 Input Language

The input language for the prototype is a very simple language which translates into the flow charts defined in Chapter 3. The language has the following BNF grammar:
Stmt $\rightarrow$ Var := Expr  
  | If Expr then Stmt else Stmt  
  | While BExpr do Stmt  
  | Skip  
  | Stmt ; Stmt

Expr $\rightarrow$ Num $*$ Var  
  | Num $*$ Var + Expr'  
  | NL

BExpr $\rightarrow$ Expr $\geq 0$ | Expr $== 0$

where Num and Var are the syntactic categories for integers and variables respectively. The non-terminal Expr' denotes Expr without the NL rule. Some syntactic sugar is layered on top of this, but the BNF grammar above illustrates the expressiveness of the input language. This structure is then translated into a flow chart data structure. Thus, the prototype does not consider pointers, arrays, structs or any other types than integers or Booleans. Function calls are supported but are implemented by inlining each function before translating into the BNF above. This means that recursion is not supported. Since the prototype is implemented for polyhedral analysis which can only handle linear assignments and conditionals, the language in the prototype is restricted to linear conditionals and assignments only. That is, all arithmetic expressions in the language either have the form $\sum_{i=0}^{n-1} a_i v_i$ or the special value NL denoting a non-linear expression. Parsing of the input language as well as translating it to a flow chart is implemented in Haskell.

8.2.2 Implemented Analyses

The prototype implements the phases outlined in Figure 5.1 on page 5.1. Each box is implemented as a Haskell program, a C++ program or a third-party module such as Piplib. The abstract interpretation phase is implemented in C++, using the NewPolka library \(^1\) for polyhedral operations. Symbolic counting

\(^1\)http://www.inrialpes.fr/pop-art/people/bjannet/newpolka/. However, the NewPolka library is discontinued and is now integrated in the APSON library: http://apron.cri.ensmp.fr/library/
was implemented using the method outlined in [96] adopted for convex polyhedra as described in Section 5.3.3. The symbolic counting was implemented in Haskell. The prototype does not implement any low-level analysis, instead, it assumes that all program points has a worst case cost of ten clock cycles. The reason for this is that our focus has not been on low-level analysis, however, to use the framework in a realistic setting, a proper low-level analysis would be needed. Two different parametric calculations have been implemented, one of them is parametric integer programming using the library Piplib. The other one is MPA which is explained in Chapter 6. MPA is implemented as a stand-alone C++ program. The prototype implements the SIMPLESLICE algorithm from [100] for preparing programs.

8.2.3 Discoveries and Experience

The prototype was implemented in order to get a first impression of how the parametric WCET analysis would work and to discover what could be improved and what works well. Chapter 9 contains results derived from this prototype, but below we give a list of observations and conclusions that could be drawn from using the prototype.

- It is possible to implement the parametric method outlined in Chapter 5 using independent modules and existing code libraries to obtain parametric WCET estimates on simple programs.

- For small programs, the bottleneck of the method is the parametric calculation. This is because the parametric calculation essentially solves a set of concrete calculation problems. As an example: ILP is in general NP-complete, and a method such as parametric integer programming which is generalising ILP is naturally even more complex.

- Parametric integer programming gives very complex solutions, even for small programs (as indicated by Figure 5.2. This means that simplification is definitely necessary. The composition part of the method introduces a lot of redundancy. This means that for practical use, simplification is needed both before parametric calculation and after.

- Piplib fails to produce solutions for larger programs, probably due to high complexity.
8.3 SWEET

SWEET (SWEdish Execution Time tool)\(^2\) is a research prototype tool for static analysis in general, and WCET analysis in particular, developed at Mälardalen Real-Time Research Centre, Mälardalen University.

SWEET analyses programs in the intermediate program language format ALF\(^{[53, 52]}\). ALF stands for Artist Flow Analysis language; it is a general intermediate program language format, especially developed for flow analysis. ALF code can be generated from different sources, like C code and assembler code, and a number of translators are available.

SWEET is not in itself a complete WCET tool, but represents the flow analysis phase of WCET analysis, and is designed to collect as much information about a program automatically. This is done via two methods, the first method is classical abstract interpretation, the second one is a more advanced analysis called *abstract execution* \(^{[41]}\) which does a more detailed simulation of the execution. Publications about SWEET and what it can do include \(^{[54, 50]}\).

For the low-level analysis and calculation phase of WCET analysis, there is another tool called low-SWEET which supports the processors NECV850E and ARM9 and three calculation methods: a fast path-based method, a global IPET method, and a hybrid clustered method.

8.3.1 Implemented Analyses

SWEET has been enhanced with the polyhedral domain in order to implement the parametric WCET method and to be able to evaluate our fully bounded polyhedral analysis. Before the polyhedral domain was implemented, SWEET

\(^{2}\)http://www.mrtc.mdh.se/projects/wcet/sweet/index.html
8.3 SWEET

SWEET (SWEdish Execution Time tool) is a research prototype tool for static analysis in general, and WCET analysis in particular, developed at Mälardalen Real-Time Research Centre, Mälardalen University.

SWEET analyses programs in the intermediate program language format ALF\[53, 52\]. ALF stands for Artist Flow Analysis language; it is a general intermediate program language format, especially developed for flow analysis. ALF code can be generated from different sources, like C code and assembler code, and a number of translators are available.

SWEET is not in itself a complete WCET tool, but represents the flow analysis phase of WCET analysis, and is designed to collect as much information about a program automatically. This is done via two methods, the first method is classical abstract interpretation, the second one is a more advanced analysis called abstract execution\[41\] which does a more detailed simulation of the execution. Publications about SWEET and what it can do include \[54, 50\].

For the low-level analysis and calculation phase of WCET analysis, there is another tool called low-SWEET which supports the processors NECV850E and ARM9 and three calculation methods: a fast path-based method, a global IPET method, and a hybrid clustered method.

8.3.1 Implemented Analyses

SWEET has been enhanced with the polyhedral domain in order to implement the parametric WCET method and to be able to evaluate our fully bounded polyhedral analysis. Before the polyhedral domain was implemented, SWEET only supported non-relational domains. Handling a relational domain is very different since a relational domain must be aware of the full memory while a non-relational domain is just abstracting each value individually. This also requires a solution to some of the problems outlined in Section 7.1 of Chapter 7, since ALF is an intermediate level language. Especially the problem of determining “what is a variable” is an issue here. ALF treats memory as a set of non-overlapping frames (see [52]), where each frame represent a memory segment of any size. If the ALF code is generated from a C-program, then normally each variable is assigned to one frame, except for dynamically allocated memory and arrays. For this reason, we have implemented the polyhedral domain so that each frame represent a dimension in the polyhedron.

We have used the Parma Polyhedra Library (PPL)\(^3\) to use polyhedral operations in the abstract interpretation. PPL implements a wide set of features including Simon and King’s wrapping operation (see 7.4 in Chapter 7).

To be able to evaluate our bounded analysis and Simon and King’s method we have implemented these two domains as well. That means that it is possible to choose from the domains poly\_cp, poly\_sk and poly\_bd when using SWEET corresponding to the classical polyhedral domain, Simon and King’s domain and the bounded polyhedra respectively. It is also possible to run SK and BD simultaneously in order to be able to compare the results easier.

The state counting phase of the parametric WCET analysis is not implemented directly into SWEET; instead SWEET can provide textual output of the results of the abstract interpretation and the tool ISCC (see Section 5.3.3) is then used to count the number of integer points symbolically. Likewise, the parametric calculation is done outside of SWEET, either by parametric integer programming or MPA. The combination phase is also done outside of SWEET. The simplification phase is yet to be implemented for the SWEET version of the parametric WCET analysis.

8.3.2 Discoveries and Experiences

The SWEET implementation is still a work in progress, but the basic functionality is implemented. The main thing that has been discovered through this implementation is:

- The Census method is very sensitive to variables that are not relevant for the timing of a program. If a the values of a single irrelevant variable is

\(^3\)http://bugseng.com/products/ppl/
“counted” in the state counting phase, this can lead to very large over-approximations. As an example, if a 32-bit variable \( x \) with no known relation to other variables is counted even though it has nothing to do with the timing, then the census method will over-estimate the number of times a state can be visited by a factor \( 2^{32} - 1 \). This means that it is very important to do aggressive slicing as well as removing other irrelevant variables.

- Struct and array variables have to be handled in a different way. Our current implementation uses frames as variables, which is not a sufficient representation whenever a frame represents more than one variable.

- The problems associated with low-level analysis outlined in Chapter 7 are becoming relevant when analysing code on the intermediate format ALF. As the previous bullet suggested, the problem of “what is a variable” is important, especially in the Census method. This may be even more important in the cases where ALF-code has been generated from object code rather than source code.
Chapter 9

Evaluation

9.1 Introduction

This chapter summarises the experimental results that we have obtained from our prototype implementation and our implementation into SWEET. The evaluations presented in this chapter are:

- An evaluation of the two suggested parametric calculation methods: parametric integer programming and MPA. These two methods are evaluated both in terms of efficiency and precision.

- An evaluation of the trade-off in restricting the tree-depth in the MPA algorithm.

- A comparison between Simon and King’s method and our bounded polyhedral analysis.

- Experiences with the implementation of the parametric WCET analysis.

9.1.1 Benchmarks

All evaluations in this chapter have been performed by analysing programs from the Mälardalen benchmark suite [49]. These benchmarks have become somewhat of a standard in WCET analysis, and have been used in evaluation for instance in [72, 95, 42, 94], in addition it was used in the WCET challenge in 2006 [48]. The source code for these benchmarks are available \(^1\). The scale

\(^1\)http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
of these programs are quite small (ranging between 40-4000 lines of code) but are sufficient to use in experiments to get indicators. Table 9.1 and 9.2 gives a description of each of these benchmarks.

Table 9.1 Benchmark programs (part 1)

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>Adaptive pulse code modulation algorithm.</td>
<td>Completely well-structured code.</td>
</tr>
<tr>
<td>bs</td>
<td>Binary search for the array of 15 integer elements.</td>
<td>Completely structured.</td>
</tr>
<tr>
<td>bsort100</td>
<td>Bubblesort program.</td>
<td>Tests the basic loop constructs, integer comparisons, and simple array handling by sorting 100 integers.</td>
</tr>
<tr>
<td>cnt</td>
<td>Counts non-negative numbers in a matrix.</td>
<td>Nested loops, well-structured code.</td>
</tr>
<tr>
<td>compress</td>
<td>Compression using lzw.</td>
<td>Adopted from SPEC95 for WCET-calculation. Only compression is done on a buffer (small one) containing totally random data.</td>
</tr>
<tr>
<td>cover</td>
<td>Program for testing many paths.</td>
<td>A loop containing many switch cases.</td>
</tr>
<tr>
<td>crc</td>
<td>Cyclic redundancy check computation on 40 bytes of data.</td>
<td>Complex loops, lots of decisions, loop bounds depend on function arguments, function that executes differently the first time it is called.</td>
</tr>
<tr>
<td>duff</td>
<td>Using “Duff’s device” to copy 43 byte array.</td>
<td>Unstructured loop with known bound, switch statement</td>
</tr>
<tr>
<td>edn</td>
<td>Finite Impulse Response (FIR) filter calculations.</td>
<td>A lot of vector multiplications and array handling.</td>
</tr>
<tr>
<td>expint</td>
<td>Series expansion for computing an exponential integral function</td>
<td>Inner loop that only runs once, structural WCET estimate gives heavy overestimate.</td>
</tr>
<tr>
<td>fdct</td>
<td>Fast Discrete Cosine Transform.</td>
<td>A lot of calculations based on integer array elements.</td>
</tr>
<tr>
<td>fftl</td>
<td>1024-point Fast Fourier Transform using the Cooly-Turkey algorithm.</td>
<td>A lot of calculations based on floating point array elements.</td>
</tr>
<tr>
<td>fibcall</td>
<td>Iterative Fibonacci, used to calculate fib(30).</td>
<td>Parameter-dependent function, single-nested loop</td>
</tr>
<tr>
<td>fir</td>
<td>Finite impulse response filter (signal processing algorithms) over a 700 items long sample.</td>
<td>Inner loop with varying number of iterations, loop-iteration dependent decisions.</td>
</tr>
<tr>
<td>insertsort</td>
<td>Insertion sort on a reversed array of size 10.</td>
<td>Input-data dependent nested loop with worst-case of ((n^2)/2) iterations (triangular loop).</td>
</tr>
</tbody>
</table>
### Table 9.2 Benchmark programs (part 2)

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>janne.complex</td>
<td>Nested loop program.</td>
<td>The inner loops number of iterations depends on the outer loops current iteration number.</td>
</tr>
<tr>
<td>jfdctint</td>
<td>Discrete-cosine transformation on 8x8 pixel block.</td>
<td>Long calculation sequences (i.e., long basic blocks), single-nested loops.</td>
</tr>
<tr>
<td>lcdnum</td>
<td>Read ten values, output half to LCD.</td>
<td>Loop with iteration-dependent flow.</td>
</tr>
<tr>
<td>lms</td>
<td>LMS adaptive signal enhancement.</td>
<td>A lot of floating point calculations.</td>
</tr>
<tr>
<td>ludcmp</td>
<td>LU decomposition algorithm.</td>
<td>A lot of calculations based on floating point arrays with the size of 50 elements.</td>
</tr>
<tr>
<td>matmult</td>
<td>Matrix multiplication of two 20x20 matrices.</td>
<td>Multiple calls to the same function, nested function calls, triple-nested loops.</td>
</tr>
<tr>
<td>minver</td>
<td>Inversion of floating point matrix.</td>
<td>Floating value calculations in 3x3 matrix. Nested loops (3 levels).</td>
</tr>
<tr>
<td>ndes</td>
<td>Complex embedded code. A lot of bit manipulation, shifts, array and matrix calculations.</td>
<td>A lot of bit manipulation, shifts, array and matrix calculations.</td>
</tr>
<tr>
<td>ns</td>
<td>Search in a multi-dimensional array.</td>
<td>Return from the middle of a loop nest, deep loop nesting (4 levels).</td>
</tr>
<tr>
<td>nsichneu</td>
<td>Simulate an extended Petri net.</td>
<td>Automatically generated code with more than 250 if-statements.</td>
</tr>
<tr>
<td>qsort-exam</td>
<td>Non-recursive version of quick sort algorithm.</td>
<td>The program sorts 20 floating point numbers in an array. Loop nesting of 3 levels.</td>
</tr>
<tr>
<td>qurt</td>
<td>Root computation of quadratic equations.</td>
<td>The real and imaginary parts of the solution are stored in arrays.</td>
</tr>
<tr>
<td>recursion</td>
<td>A simple example of recursive code.</td>
<td>Both self-recursion and mutual recursion are used.</td>
</tr>
<tr>
<td>select</td>
<td>A function to select the Nth largest number in a floating point array.</td>
<td>A lot of floating value array calculations, loop nesting (3 levels).</td>
</tr>
<tr>
<td>sqrt</td>
<td>Square root function implemented by Taylor series.</td>
<td>Simple numerical calculation.</td>
</tr>
<tr>
<td>st</td>
<td>Statistics program.</td>
<td>This program computes for two arrays of numbers the sum, the mean, the variance, and standard deviation, and the correlation coefficient between the two arrays.</td>
</tr>
<tr>
<td>statemate</td>
<td>Automatically generated code.</td>
<td>Generated by the STAtechart Real-time-Code generator STARC.</td>
</tr>
</tbody>
</table>
For our first prototype implementation a selection of the benchmarks have been hand-translated to the simple input language presented in Chapter 8. In this case only benchmarks which have properties that conforms with the limitations of the input language (such as completely structured flow, no use of pointers or recursion) and where array elements are ignored (i.e., they are treated as variables with no known information on their values).

9.1.2 A Note on Input Parameters

Chapter 3 shows how the input parameters to a program are determined. However, when analysing benchmarks, we will often use constant macros as input parameters. While they are not technically a variable (all constant-macros are replaced by a constant during preprocessing), they can be seen as constant program variables and can therefore be consider input parameters, just as our definition of input variables. Another common input parameter type when analysing a single function, is to use the value of the argument as input parameter. Both these uses represent values that are invariant during the execution of the analysed program and is therefore consistent with our definition of input parameter.

9.1.3 Experimental set-up

All experiments are performed in Windows XP Professional SP3 on an Intel core duo 2.4 GHz with 2.39GB RAM and a 6MB L2-cache. Most experiments have been run under Cygwin, and any timings have been obtained by the Unix command time.

9.2 Evaluating Parametric Calculation

In the first prototype implementation it became clear that the bottle-neck of the parametric WCET analysis we have suggested was in the parametric calculation. In this implementation each statement (not each basic block) was considered to be a program point, leading to a huge set of parameters in the parametric calculation, even for relatively small programs. The tool Piplib started to take a lot of time and in some cases fail to deliver a solution. MPA was developed to improve on this, and this section compares parametric integer programming with MPA for the purposes of parametric calculation. The comparison was made with our first prototype.
Both Piplib and our C++ implementation of MPA were compiled with GCC 3.4.4 under Cygwin. Since the prototype tool lacks a proper low-level analysis, all program points are assumed to have a constant WCET of 10 clock cycles in this evaluation. The experiments have been performed by analysing some benchmarks using both parametric integer programming and MPA.

In this evaluation, one function at a time has been analysed and the name of the function is given as second name in the benchmark column (i.e., edn/fir denotes that the function “fir” from the benchmark edn is analysed). When a benchmark is marked with x2, x3 etc, it means that the particular function has been called repeatedly and is thus inlined multiple times. This is just to see how the methods scale when the number of program points are increased.

Table 9.3 shows the evaluation, and the columns are explained below.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Pps</th>
<th>PIP</th>
<th>MPA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Size</td>
</tr>
<tr>
<td>edn/fir</td>
<td>11</td>
<td>0.1s</td>
<td>3</td>
</tr>
<tr>
<td>edn/latsynth</td>
<td>7</td>
<td>0.1s</td>
<td>1</td>
</tr>
<tr>
<td>edn/latsynth x2</td>
<td>12</td>
<td>0.1s</td>
<td>2</td>
</tr>
<tr>
<td>edn/latsynth x4</td>
<td>25</td>
<td>0.1s</td>
<td>10</td>
</tr>
<tr>
<td>cnt/initialize</td>
<td>12</td>
<td>0.1s</td>
<td>3</td>
</tr>
<tr>
<td>cnt/initialize x2</td>
<td>23</td>
<td>0.2s</td>
<td>83</td>
</tr>
<tr>
<td>cnt/initialize x3</td>
<td>34</td>
<td>2.6s</td>
<td>1782</td>
</tr>
<tr>
<td>cnt/sum</td>
<td>16</td>
<td>0.3s</td>
<td>80</td>
</tr>
<tr>
<td>cnt/sum x2</td>
<td>31</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>jcomplex</td>
<td>23</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>matmult/Initialize</td>
<td>12</td>
<td>0.1s</td>
<td>3</td>
</tr>
<tr>
<td>matmult/Initialize x2</td>
<td>23</td>
<td>0.3s</td>
<td>83</td>
</tr>
</tbody>
</table>

**Program points** Labeled as #Pps in the table. This is the number of arcs in the flow chart. In this chart, however, each statement rather than each basic block is considered a node.

**Execution time** The execution time of the running algorithms. The cases where the time is not given means that Piplib failed to solve the problem due to a too high complexity of the solution.

**Size** The size of the solution, given in KB. The measurements come from the
file sizes of the solutions textual representations. Note that Piplib does not scale well, especially not in solution size.

It should be noted that we later have discovered that the library isl [113], which includes an implementation of parametric integer programming, seems to be able to handle certain cases better than Piplib. For instance, it was possible to analyse jcomplex in 6 seconds using this tool, however, cst/sum x2 was still a problem. No full evaluation have yet been done using isl.

9.2.1 Evaluation of Precision

The precision of MPA compared to PIP is hard to measure since the solutions of MPA and PIP looks so different. The precision has been compared by evaluating WCET for some input values. As an example, edn/fir has been evaluated by choosing instances of the input parameters N and ORDER. Table 9.4 shows the estimated WCETs of instantiated variables from the two parametric calculation methods. The input parameters have been chosen so they have a parametric behaviour and are instantiated with values somewhat close to their original values in the benchmark programs. The last two columns show how much the MPA solution differs from the PIP solution in that particular instantiation. As can be seen, MPA gives slightly less precise result compared to PIP. An imprecision of up to 32.3% compared to PIP has been observed (on cnt/sum), but in most cases it is less than one percent.

9.2.2 Evaluation of Upper Bounds on Min-Tree Depth

As mentioned in Section 6.3.4, the first levels of a Min-Tree are the ones most likely to contribute to the final solution. This can be demonstrated by running MPA with different max-depths of the Min-Trees. Interestingly enough, for most of the evaluated programs, it is sufficient to have a maximum depth of one to achieve the precision presented in Table 9.4. To be precise, the following programs gives the same precision as in Table 9.4 when the maximum depth is set to one: edn/fir, edn/latsynth, edn/latsynth x2, edn/latsynth x4, cnt/initialize, cnt/sum and matmult/initialize.

The rest of the benchmarks lost precision when the depth was set to one. Table 9.5 shows how the precision changes when changing the max depth. As can be seen, in all tested programs, the best possible precision can be achieved with the maximum depth set to 4, but in most cases it is sufficient to set it even lower. Note that in these small examples, the execution time of MPA is still
9.2 Evaluating Parametric Calculation

neglectable, so there are no obvious benefits on setting a maximum depth for them. However, in larger programs, as will be seen in Section 9.6, the benefits are obvious.
Table 9.4 Precision Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parameters</th>
<th>PIP result</th>
<th>MPA result</th>
<th>Diff</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>edn/fir</td>
<td>N = 100, ORDER = 25</td>
<td>60790</td>
<td>60810</td>
<td>20</td>
<td>0.03%</td>
</tr>
<tr>
<td></td>
<td>N = 100, ORDER = 50</td>
<td>78040</td>
<td>78060</td>
<td>20</td>
<td>0.03%</td>
</tr>
<tr>
<td></td>
<td>N = 100, ORDER = 75</td>
<td>57790</td>
<td>57810</td>
<td>20</td>
<td>0.03%</td>
</tr>
<tr>
<td></td>
<td>N = 200, ORDER = 25</td>
<td>141790</td>
<td>141810</td>
<td>20</td>
<td>0.01%</td>
</tr>
<tr>
<td></td>
<td>N = 200, ORDER = 50</td>
<td>234040</td>
<td>234060</td>
<td>20</td>
<td>&lt;0.01%</td>
</tr>
<tr>
<td></td>
<td>N = 200, ORDER = 75</td>
<td>288790</td>
<td>288810</td>
<td>20</td>
<td>&lt;0.01%</td>
</tr>
<tr>
<td></td>
<td>N = 300, ORDER = 25</td>
<td>390040</td>
<td>390060</td>
<td>20</td>
<td>&lt;0.01%</td>
</tr>
<tr>
<td></td>
<td>N = 300, ORDER = 50</td>
<td>519790</td>
<td>519810</td>
<td>20</td>
<td>&lt;0.01%</td>
</tr>
<tr>
<td>edn/latsynth</td>
<td>n = 50</td>
<td>1520</td>
<td>1520</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>n = 100</td>
<td>3020</td>
<td>3020</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>n = 200</td>
<td>6020</td>
<td>6020</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>edn x2</td>
<td>n = 50</td>
<td>3030</td>
<td>3060</td>
<td>30</td>
<td>0.99%</td>
</tr>
<tr>
<td></td>
<td>n = 100</td>
<td>6030</td>
<td>6060</td>
<td>30</td>
<td>0.5%</td>
</tr>
<tr>
<td></td>
<td>n = 200</td>
<td>12030</td>
<td>12060</td>
<td>30</td>
<td>0.25%</td>
</tr>
<tr>
<td>edn x4</td>
<td>n = 50</td>
<td>6050</td>
<td>6160</td>
<td>110</td>
<td>1.82%</td>
</tr>
<tr>
<td></td>
<td>n = 100</td>
<td>12050</td>
<td>12160</td>
<td>110</td>
<td>0.91%</td>
</tr>
<tr>
<td></td>
<td>n = 200</td>
<td>24050</td>
<td>24160</td>
<td>110</td>
<td>0.46%</td>
</tr>
<tr>
<td>cnt/initialize</td>
<td>MAXSIZE=10</td>
<td>4640</td>
<td>4660</td>
<td>20</td>
<td>0.4%</td>
</tr>
<tr>
<td></td>
<td>MAXSIZE=20</td>
<td>17240</td>
<td>17260</td>
<td>20</td>
<td>0.1%</td>
</tr>
<tr>
<td></td>
<td>MAXSIZE=30</td>
<td>37840</td>
<td>37860</td>
<td>20</td>
<td>0.05%</td>
</tr>
<tr>
<td>cnt x2</td>
<td>MAXSIZE=10</td>
<td>9270</td>
<td>9810</td>
<td>540</td>
<td>5.83%</td>
</tr>
<tr>
<td></td>
<td>MAXSIZE=20</td>
<td>34470</td>
<td>35510</td>
<td>1040</td>
<td>3.02%</td>
</tr>
<tr>
<td></td>
<td>MAXSIZE=30</td>
<td>75670</td>
<td>77210</td>
<td>1540</td>
<td>2.04%</td>
</tr>
<tr>
<td>cnt x3</td>
<td>MAXSIZE=10</td>
<td>13900</td>
<td>15460</td>
<td>1560</td>
<td>11.22%</td>
</tr>
<tr>
<td></td>
<td>MAXSIZE=20</td>
<td>51700</td>
<td>54760</td>
<td>3060</td>
<td>5.92%</td>
</tr>
<tr>
<td></td>
<td>MAXSIZE=30</td>
<td>113500</td>
<td>118060</td>
<td>4560</td>
<td>4.02%</td>
</tr>
<tr>
<td>cnt/sum</td>
<td>MAXSIZE=10</td>
<td>-</td>
<td>17810</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>MAXSIZE=20</td>
<td>-</td>
<td>67510</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>MAXSIZE=30</td>
<td>-</td>
<td>149210</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>jcomplex</td>
<td>a = 1, b = 1</td>
<td>-</td>
<td>80</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>a = 1, b = 15</td>
<td>-</td>
<td>120</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>a = 1, b = 30</td>
<td>-</td>
<td>110</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>a = 15, b = 1</td>
<td>-</td>
<td>80</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>a = 15, b = 15</td>
<td>-</td>
<td>80</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>a = 15, b = 30</td>
<td>-</td>
<td>30</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>a = 30, b = 1</td>
<td>-</td>
<td>80</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>a = 30, b = 15</td>
<td>-</td>
<td>80</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>a = 30, b = 30</td>
<td>-</td>
<td>30</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>matmult/Initialize</td>
<td>UPPERLIMIT = 100</td>
<td>406040</td>
<td>406060</td>
<td>20</td>
<td>&lt;0.01%</td>
</tr>
<tr>
<td></td>
<td>UPPERLIMIT = 150</td>
<td>909040</td>
<td>909060</td>
<td>20</td>
<td>&lt;0.01%</td>
</tr>
<tr>
<td></td>
<td>UPPERLIMIT = 200</td>
<td>1612040</td>
<td>1612060</td>
<td>20</td>
<td>&lt;0.01%</td>
</tr>
<tr>
<td>matmult x2</td>
<td>UPPERLIMIT = 100</td>
<td>812070</td>
<td>817110</td>
<td>5040</td>
<td>0.62%</td>
</tr>
<tr>
<td></td>
<td>UPPERLIMIT = 150</td>
<td>1818070</td>
<td>1825610</td>
<td>7540</td>
<td>0.41%</td>
</tr>
<tr>
<td></td>
<td>UPPERLIMIT = 200</td>
<td>3224070</td>
<td>3234110</td>
<td>10040</td>
<td>0.31%</td>
</tr>
</tbody>
</table>
### Table 9.4 Precision Comparison

<table>
<thead>
<tr>
<th>Benchmark Parameters</th>
<th>Benchmark Parameters</th>
<th>MPA Result</th>
<th>Diff</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>edn/fir N = 100, ORDER = 25</td>
<td>edn/fir N = 100, ORDER = 25</td>
<td>60790</td>
<td>20</td>
<td>0.03%</td>
</tr>
<tr>
<td>edn/fir N = 100, ORDER = 50</td>
<td>edn/fir N = 100, ORDER = 50</td>
<td>78040</td>
<td>20</td>
<td>0.03%</td>
</tr>
<tr>
<td>edn/fir N = 100, ORDER = 75</td>
<td>edn/fir N = 100, ORDER = 75</td>
<td>57790</td>
<td>20</td>
<td>0.03%</td>
</tr>
<tr>
<td>edn/latsynth n = 50</td>
<td>edn/latsynth n = 50</td>
<td>1520</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>edn x2 n = 50</td>
<td>edn x2 n = 50</td>
<td>3030</td>
<td>30</td>
<td>0.99%</td>
</tr>
<tr>
<td>edn x4 n = 50</td>
<td>edn x4 n = 50</td>
<td>6050</td>
<td>110</td>
<td>1.82%</td>
</tr>
<tr>
<td>cnt/initialize MAXSIZE=10</td>
<td>cnt/initialize MAXSIZE=10</td>
<td>4640</td>
<td>20</td>
<td>0.4%</td>
</tr>
<tr>
<td>cnt/initialize MAXSIZE=20</td>
<td>cnt/initialize MAXSIZE=20</td>
<td>17240</td>
<td>20</td>
<td>0.1%</td>
</tr>
<tr>
<td>cnt/initialize MAXSIZE=30</td>
<td>cnt/initialize MAXSIZE=30</td>
<td>37840</td>
<td>20</td>
<td>0.05%</td>
</tr>
<tr>
<td>cnt x2 MAXSIZE=10</td>
<td>cnt x2 MAXSIZE=10</td>
<td>9270</td>
<td>540</td>
<td>5.83%</td>
</tr>
<tr>
<td>cnt x2 MAXSIZE=20</td>
<td>cnt x2 MAXSIZE=20</td>
<td>34470</td>
<td>1040</td>
<td>3.02%</td>
</tr>
<tr>
<td>cnt x2 MAXSIZE=30</td>
<td>cnt x2 MAXSIZE=30</td>
<td>75670</td>
<td>1540</td>
<td>2.04%</td>
</tr>
<tr>
<td>cnt x3 MAXSIZE=10</td>
<td>cnt x3 MAXSIZE=10</td>
<td>13900</td>
<td>1560</td>
<td>11.22%</td>
</tr>
<tr>
<td>cnt x3 MAXSIZE=20</td>
<td>cnt x3 MAXSIZE=20</td>
<td>51700</td>
<td>3060</td>
<td>5.92%</td>
</tr>
<tr>
<td>cnt x3 MAXSIZE=30</td>
<td>cnt x3 MAXSIZE=30</td>
<td>113500</td>
<td>4560</td>
<td>4.02%</td>
</tr>
<tr>
<td>cnt/sum MAXSIZE=10</td>
<td>cnt/sum MAXSIZE=10</td>
<td>6640</td>
<td>2020</td>
<td>30.4%</td>
</tr>
<tr>
<td>cnt/sum MAXSIZE=20</td>
<td>cnt/sum MAXSIZE=20</td>
<td>25240</td>
<td>8020</td>
<td>31.8%</td>
</tr>
<tr>
<td>cnt/sum MAXSIZE=30</td>
<td>cnt/sum MAXSIZE=30</td>
<td>55840</td>
<td>18020</td>
<td>32.3%</td>
</tr>
<tr>
<td>matmult/Initialize UPPERLIMIT = 100</td>
<td>matmult/Initialize UPPERLIMIT = 100</td>
<td>406040</td>
<td>&lt;0.01%</td>
<td></td>
</tr>
<tr>
<td>matmult/Initialize UPPERLIMIT = 150</td>
<td>matmult/Initialize UPPERLIMIT = 150</td>
<td>909040</td>
<td>&lt;0.01%</td>
<td></td>
</tr>
<tr>
<td>matmult/Initialize UPPERLIMIT = 200</td>
<td>matmult/Initialize UPPERLIMIT = 200</td>
<td>1612040</td>
<td>&lt;0.01%</td>
<td></td>
</tr>
<tr>
<td>matmult x2 UPPERLIMIT = 100</td>
<td>matmult x2 UPPERLIMIT = 100</td>
<td>812070</td>
<td>5040</td>
<td>0.62%</td>
</tr>
<tr>
<td>matmult x2 UPPERLIMIT = 150</td>
<td>matmult x2 UPPERLIMIT = 150</td>
<td>1818070</td>
<td>7540</td>
<td>0.41%</td>
</tr>
<tr>
<td>matmult x2 UPPERLIMIT = 200</td>
<td>matmult x2 UPPERLIMIT = 200</td>
<td>3224070</td>
<td>10040</td>
<td>0.31%</td>
</tr>
</tbody>
</table>

### Table 9.5 Comparison of precision with different max-depths

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parameters</th>
<th>Result with max-depth</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnt/initialize x2</td>
<td>MAXSIZE = 10</td>
<td>∞</td>
<td>9810</td>
<td>9810</td>
<td>9810</td>
<td></td>
</tr>
<tr>
<td>cnt/initialize x3</td>
<td>MAXSIZE = 10</td>
<td>∞</td>
<td>17840</td>
<td>15460</td>
<td>15460</td>
<td></td>
</tr>
<tr>
<td>cnt/sum x2</td>
<td>MAXSIZE = 10</td>
<td>∞</td>
<td>17810</td>
<td>17810</td>
<td>17810</td>
<td></td>
</tr>
<tr>
<td>jcomplex</td>
<td>a=30,b=30</td>
<td>90</td>
<td>80</td>
<td>70</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>matmult/initialize x2</td>
<td>UPPERLIMIT=100</td>
<td>∞</td>
<td>817110</td>
<td>817110</td>
<td>817110</td>
<td></td>
</tr>
</tbody>
</table>

### 9.2.3 Scaling Properties

Since the input language to this prototype is somewhat limited (requiring source code to be translated by hand), the scalability of MPA is also evaluated by running it in isolation on a larger set of benchmarks. This is possible since parametric calculation can be performed over the structure (control flow graph) and does not need the actual code.

Since the translated benchmarks used in previous section are small, they don’t show well how MPA scales with program size. In order to investigate how MPA scales in more realistic cases, we have run the algorithm in isolation (independent of the parametric framework and the prototype) on the full benchmark Mälardalen benchmark suite. In this case we have used SWEET to obtain control flow graphs for the benchmarks. A control flow graph (CFG) is a graph where each node is a basic block and where there are no explicit merge or conditional nodes. Note that the MPA is equally valid on control flow graph as on a flow chart, since the premises of the constraints are the same (each node is upper bounded by inflow and outflow). The CFGs obtained from SWEET are on the full programs, that is, it includes all functions and all function calls. In contrast to the evaluation in Section 9.2, the CFGs obtained from SWEET are not inlined; each function call is an edge from the caller to the callee, and each return is an edge from the exit of a function, back to the caller. Since the algorithm in this experiment is run in isolation (i.e., no state counting phase is done), we cannot examine the precision of MPA in this test, just how resource consuming it is.
Table 9.6 Scalability Properties of MPA

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Pps</th>
<th>MD</th>
<th>Iterations</th>
<th>Calls</th>
<th>Time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>884</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>bs</td>
<td>39</td>
<td>2</td>
<td>249</td>
<td>1538</td>
<td>0.04s</td>
<td>6633</td>
</tr>
<tr>
<td>bsort</td>
<td>66</td>
<td>2</td>
<td>1750</td>
<td>9610</td>
<td>0.14s</td>
<td>44152</td>
</tr>
<tr>
<td>cnt</td>
<td>93</td>
<td>2</td>
<td>1537</td>
<td>11547</td>
<td>0.16s</td>
<td>48022</td>
</tr>
<tr>
<td>cover</td>
<td>1593</td>
<td>121</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>compress</td>
<td>380</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>crc</td>
<td>127</td>
<td>2</td>
<td>10543</td>
<td>69017</td>
<td>0.85s</td>
<td>316212</td>
</tr>
<tr>
<td>duff</td>
<td>390</td>
<td>9</td>
<td>5937</td>
<td>121369</td>
<td>1.22s</td>
<td>475556</td>
</tr>
<tr>
<td>edn</td>
<td>342</td>
<td>2</td>
<td>7202</td>
<td>95585</td>
<td>1.04s</td>
<td>421438</td>
</tr>
<tr>
<td>expint</td>
<td>88</td>
<td>2</td>
<td>1028</td>
<td>7983</td>
<td>0.11s</td>
<td>32407</td>
</tr>
<tr>
<td>fac</td>
<td>36</td>
<td>2</td>
<td>260</td>
<td>1298</td>
<td>0.08s</td>
<td>5959</td>
</tr>
<tr>
<td>fdct</td>
<td>147</td>
<td>2</td>
<td>973</td>
<td>21565</td>
<td>0.22s</td>
<td>79457</td>
</tr>
<tr>
<td>fft1</td>
<td>266</td>
<td>4</td>
<td>72572</td>
<td>482486</td>
<td>6.18s</td>
<td>2390541</td>
</tr>
<tr>
<td>fibcall</td>
<td>29</td>
<td>2</td>
<td>75</td>
<td>702</td>
<td>0.03s</td>
<td>2567</td>
</tr>
<tr>
<td>fir</td>
<td>77</td>
<td>2</td>
<td>779</td>
<td>6828</td>
<td>0.10s</td>
<td>27315</td>
</tr>
<tr>
<td>insertsort</td>
<td>39</td>
<td>2</td>
<td>175</td>
<td>1313</td>
<td>0.03s</td>
<td>5310</td>
</tr>
<tr>
<td>jcomplex</td>
<td>48</td>
<td>2</td>
<td>792</td>
<td>3289</td>
<td>0.06s</td>
<td>16763</td>
</tr>
<tr>
<td>jfdctint</td>
<td>122</td>
<td>2</td>
<td>1038</td>
<td>14726</td>
<td>0.16s</td>
<td>55563</td>
</tr>
<tr>
<td>lcdnum</td>
<td>158</td>
<td>17</td>
<td>4042</td>
<td>40164</td>
<td>0.46s</td>
<td>168927</td>
</tr>
<tr>
<td>lms</td>
<td>262</td>
<td>4</td>
<td>90864</td>
<td>586176</td>
<td>7.44s</td>
<td>2959463</td>
</tr>
<tr>
<td>ludcmp</td>
<td>181</td>
<td>3</td>
<td>5583</td>
<td>35101</td>
<td>0.47s</td>
<td>169103</td>
</tr>
<tr>
<td>matmult</td>
<td>97</td>
<td>2</td>
<td>1351</td>
<td>9441</td>
<td>0.13s</td>
<td>40263</td>
</tr>
<tr>
<td>minmax</td>
<td>109</td>
<td>3</td>
<td>1926</td>
<td>18881</td>
<td>0.23s</td>
<td>74619</td>
</tr>
<tr>
<td>ndes</td>
<td>445</td>
<td>9</td>
<td>1235359</td>
<td>11593218</td>
<td>2m19s</td>
<td>54938649</td>
</tr>
<tr>
<td>ns</td>
<td>46</td>
<td>2</td>
<td>562</td>
<td>2838</td>
<td>0.06s</td>
<td>13245</td>
</tr>
<tr>
<td>nsichneu</td>
<td>3313</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>prime</td>
<td>114</td>
<td>3</td>
<td>11425</td>
<td>79060</td>
<td>0.95s</td>
<td>356992</td>
</tr>
<tr>
<td>qsort-exam</td>
<td>153</td>
<td>2</td>
<td>15861</td>
<td>104870</td>
<td>1.30s</td>
<td>501762</td>
</tr>
<tr>
<td>qurt</td>
<td>135</td>
<td>4</td>
<td>27658</td>
<td>178578</td>
<td>2.17s</td>
<td>821808</td>
</tr>
<tr>
<td>select</td>
<td>136</td>
<td>2</td>
<td>32418</td>
<td>165320</td>
<td>2.25s</td>
<td>842275</td>
</tr>
<tr>
<td>sqrt</td>
<td>49</td>
<td>3</td>
<td>896</td>
<td>4717</td>
<td>0.08s</td>
<td>21758</td>
</tr>
<tr>
<td>statemate</td>
<td>1287</td>
<td>47</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ud</td>
<td>150</td>
<td>2</td>
<td>2770</td>
<td>15938</td>
<td>0.23s</td>
<td>78277</td>
</tr>
</tbody>
</table>

Table 9.6 shows the result of the tests. The first column is the benchmark name, second column (#Pps) is the number of program points. The third col-
umn (MD) is the maximum degree of a node, i.e. the maximum number of outgoing or incoming edges from a node. As seen in Table 9.6 this property strongly affects the time consumed by MPA. The fourth column (iterations) is the global number of iterations of MPAs main loop (rows 4-21 in Algorithm 1). The fifth column (Calls) is the global number of calls (including recursive calls) to Algorithm 1. The sixth column (Time) is the real time of the algorithm running. Finally, the seventh column (Size) is the size of the solution text file in bytes.

Note that MPA in this case runs without imposing a limit on the Min-Tree depth. This caused five of the programs to fail the analysis. The reason seems to be the combination of many program points and a high vertex degree on the nodes, resulting in MPA taking a high number of recursive branches. However, by imposing an upper bound on the depth of the produced Min-Trees, most of these programs can be analysed. Since we do not run the whole framework in these tests, we are not able to see how much precision is lost from doing so, but the tests in the previous section indicated that a maximum depth of four was enough for small benchmarks. For the five programs that failed the unbounded Min-Tree depth we have analysed them with different upper bounds to see how they would scale. The result is shown in Table 9.7. The program nsichneu fails to be analysed even for maximum depth 2 and is the only benchmark which completely fails to be analysed. As seen, the other benchmarks can be analysed, but may or may not be over-approximated.

### Table 9.7 scaling properties with max-depth

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Execution Time of MPA with Min-Tree max-depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>adpcm</td>
<td>3s</td>
</tr>
<tr>
<td>compress</td>
<td>0.8s</td>
</tr>
<tr>
<td>cover</td>
<td>1m9s</td>
</tr>
<tr>
<td>state mate</td>
<td>11s</td>
</tr>
<tr>
<td>nsichneu</td>
<td>-</td>
</tr>
</tbody>
</table>

### 9.3 The Reason for Over-Estimation

As seen in Table 9.4, MPA often over-estimates the results compared to parametric integer programming. This occurs in the cases where MPA fails to derive the tightest possible bounds for a Min-Tree. There are cases where the
Figure 9.1: A program causing over-estimation in MPA

bounds (6.3), (6.4) and (6.1) are not sufficient to express the tightest possible bound. Consider Figure 9.1 which depicts a nested loop (note that MPA is not concerned with the semantics of the program, so the boxes are intentionally left empty). The Min-Tree of $q_7$ and $q_8$ (they are equal) is shown in Figure 9.2. By inspection of Figure 9.1 we can see that the execution of $q_3$ implies execution of $q_7$ and $q_8$, thus $p_3$ is an upper bound of $q_7$ (since, by (6.1), $q_3$ can not execute more than $p_3$ times, which implies that neither can $q_7$). As seen in Figure 9.2, MPA does not derive $p_3$ as an upper bound, instead, the bounds $p_3 + \min(p_5, p_6)$ and $p_3 + p_2$ are derived. This is because $q_3$ is one out of two outgoing arcs from $q_2$ as well as one out of two incoming arcs for $q_4$, giving rise to the constraints $x_4 \leq x_3 + x_6$ (by (6.3)) and $x_1 \leq x_2 + x_3$ (by (6.4)). In summary, the upper bound $p_3$ cannot be determined for $q_7$ only by the constraints (6.3),(6.4) and (6.1) only, and hence MPA fails to find it.

As an example, we will show that for a certain instantiation of the parameters $p_0, \ldots, p_8$, that none of the bounds (i.e., branches) in Figure 9.2 are sufficiently tight. Assume that the parameters are instantiated as follows:

$$p_0, \ldots, p_8 = [1, \infty, 1, 1, \infty, 3, 3, \infty, \infty].$$

Again, by inspection of Figure 9.1 we can conclude that $q_7$ can maximally be executed once, since $p_3 = 1$. But the bounds derived from Figure 9.2 are (from left to right): $p_3 + \min(p_5, p_6) = 1 + \min(3, 3) = 4$, $p_1 = \infty$, $p_4 = \infty$, 

Figure 9.2: The Min-Tree for the program points $q_7$ and $q_8$. 

9.4 Evaluation of Bounded Polyhedra

The evaluation of bounded polyhedra and Simon and King's method that was described in Chapter 7, we have used SWEET which implements both these techniques. This means that the analysed programs have been translated to ALF and that programs are analysed from a CFG format. This also means that each program point here corresponds to a basic block. In the evaluation, we have applied widening in loops at the exits of their header basic blocks (the entry point of the loop) that do not exit the loop. This ensures that each path in the loop has exactly one widening point. All of the programs that we have analysed in our evaluation have structured loops (meaning that each loop has exactly one header), so it is possible to place widening in this way. Other placements of widening points are also possible, but this has not been investigated.
$p_7 = \infty$, $p_8 = \infty$ and $p_3 + p_2 = 1 + 1 = 2$. Thus, the tightest bound (the minimum of the above) given from the Min-Tree is 2, which clearly is an over-approximation.

9.4 Evaluation of Bounded Polyhedra

The evaluation of bounded polyhedra and Simon and King’s method that was described in Chapter 7, we have used SWEET which implements both these techniques. This means that the analysed programs have been translated to ALF and that programs are analysed from a CFG format. This also means that each program point here corresponds to a basic block. In the evaluation, we have applied widening in loops at the exits of their header basic blocks (the entry point of the loop) that do not exit the loop. This ensures that each path in the loop has exactly one widening point. All of the programs that we have analysed in our evaluation have structured loops (meaning that each loop has exactly one header), so it is possible to place widening in this way. Other placements of widening points are also possible, but this has not been investigated.
9.4.1 Comparison between SK and BD

Our aim is to compare the precision of the polyhedra resulting when analysing programs with SK and BD. Our evaluation consists of analysing a set of benchmark programs with both SK and BD. Abstract interpretation yields results for all points in the analysed program, in our case each basic block is associated with a polyhedron. For each of basic block $q$ of a program, we associate a polyhedron $\hat{CS}_{SK}(q)$ as the result of abstract interpretation with Simon and King’s method and $\hat{CS}_{BD}(q)$ as the result of abstract interpretation with fully bounded polyhedra.

While the usefulness of the results should ideally be compared considering a particular application, we have settled on two ways of comparison. The first one is to investigate the relation between the sets $\gamma_{SK}(\hat{CS}_{SK}(q))$ and $\gamma_{SK}(\hat{CS}_{BD}(q))$ in terms of inclusion. However, this does not show how much better the result is, just if it is better or not. For this reason we also measure the precision of a result by the number of integer points it contains. This measurement is highly relevant in the Census method.

9.4.2 Computing the Set of Integers of a Polyhedron

As concluded Chapter 7, $\gamma_{SK}(P)$ does not necessarily form a convex polyhedron. This means we cannot use conventional techniques to compare and count elements in convex polyhedra when dealing with these kind of sets. However, if $P$ is a fully bounded polyhedron, then $\gamma_{SK}(P)$ is the union of a finite set of convex polyhedra within the base window. The Barvinok library and the tool iscc provides techniques for manipulating and counting sets of integers, including unions of convex polyhedra. Thus, as long as the polyhedra are fully bounded, we can use Barvinok to compare them inclusion-wise as well as count the size of these sets.

If $P$ is unbounded, we can no longer see $\gamma_{SK}(P)$ as a finite union of convex polyhedra and we have to use another technique. We can form the polyhedron $\text{wrap}(P, \emptyset, \text{VAR}_P) \subseteq R(\text{VAR}_P)$ which is fully contained within the base window. Moreover, $\gamma_{SK}(\text{wrap}(P, \emptyset, \text{VAR}_P))$ always forms a convex polyhedron, which means that straightforward counting techniques for convex polyhedra can be used. However, $\text{wrap}(P, \emptyset, \text{VAR}_P)$ is potentially less precise than $P$. Furthermore, both SK and BD are designed to avoid explicit wrapping as much as possible to avoid imprecision. Still, to give an indication of the improvement of BD compared to SK in the cases where SK results in unbounded polyhedra, we use this comparison.
In the experiment, six benchmarks programs were analysed with SK and BD. In two of the benchmarks, namely bs and bsort100, $\overline{\mathbb{S}}_{SK}(q)$ is unbounded for some basic blocks $q$. For these two benchmarks, the results are compared using $\text{wrap}(\overline{\mathbb{S}}_{SK}(q), \emptyset, \text{VAR}_P)$. This is also indicated in the tables.

### 9.4.3 The Results

#### Set Relations.

Table 9.8 shows the percentage of program points $q$ in which the set $\gamma_{SK}(P_{BD}(q))$ is a subset respective proper subset to $\gamma_{SK}(P_{SK}(q))$.

The column $PPs$ shows how many program points (in this case basic blocks) the benchmark has, and consequently, how many pairs of polyhedra are compared. As can be seen, BD is at least as precise as SK in terms of inclusion in all observed cases, often strictly better.

#### Number of Integer Points.

Table 9.9 shows statistics when comparing $|\gamma_{SK}(P_{BD}(q))|$ to $|\gamma_{SK}(P_{SK}(q))|$, except for bs and bsort where $|\gamma_{SK}(\text{wrap}(P_{BD}(q), \emptyset, \text{VAR}_P))|$ is compared to $|\gamma_{SK}(\text{wrap}(P_{SK}(q), \emptyset, \text{VAR}_P))|$. We explain the table below:

**BBs** The number of basic blocks in the benchmark.

**Best** This column shows the highest percentage-decrease in number of integer points when comparing BD to SK observed in this benchmark.

**Avg str**. This shows the average decrease in number of integer points when comparing BD to SK, in the cases where there is a strict improvement.
The percentage of polyhedra with strict improvement can be seen in Table 9.8.

**Avg imp.** This shows the average decrease in number of integer points inside polyhedra when comparing BD to SK, including cases where there was no strict improvement.

In total 450 SK polyhedra have been compared to 450 BD polyhedra. As seen in Table 9.8, BD is at least as good as SK in all cases, and in 29% of the total cases strictly better. The amount of improvement varies from less than one percent, to 75% in some polyhedra. As we will see in the following section, this improvement comes with little cost.

To give an idea of what is being improved and why, we briefly analyse the results of the six benchmark programs.

**BS** As can be seen in Table 9.9, our method does not seem to strictly improve the results. For this program, using SK results in unbounded polyhedra for eight out of eleven basic blocks, which means that we cannot count integer points and thus not conclude whether we have a strict improvement or not. However, by wrapping the results, we see a slight improvement in three cases. In all three cases SK derives no information (covering the complete base window) while BD maintains some brief relational information.

**BSort100** A couple of program points yields unbounded polyhedra for SK. The best case shows a 25% improvement, where BD seem to be able to retain a relation between the indices of two nested loops where SK fails. When the results are wrapped to include the previously unbounded polyhedra, we observe a strict improvement in 40% of the cases.
**fibcall** In this program, SK derives bounded polyhedra in all cases, so the results are completely comparable without wrapping. The best case here shows as much as a 75% reduction in comparison with SK. In the cases where a reduction is shown, SK derives no information except that one variable has a constant value, while BD gives a complex polyhedron stretching outside the base window.

**insertsort** shows improvement in some points where SK derives no information. The improvement however, is very slight.

**jcomplex** This benchmark is designed to have an intricate relationship between an inner and outer loop. The BD method does preserve an important loop condition inside the loop, due to the limited widening, leading to a 75% increase of precision at one basic block. This preserved relationship seems to carry out to several other places giving a general increase of precision.

**loop3** The final benchmark loop3 contains a number of similar loops with different bounds. An improvement in 49% of the program points is observed, with the best improvement 25%.

### 9.4.4 On Efficiency

To be able to compare the precision of the analyses, the SK and BD methods were run simultaneously in the above experiments. To also give an impression of the complexity of the two approaches, we have run the analyses in isolation on the two biggest benchmarks (in terms of program points): loop3 and bsort100. We compare them with two criteria: the running times and the number of iterations before termination. Note that the analysis implementation has not been optimised and contains a lot of run-time checks and debugging information. The running times are just intended to give a rough idea of the difference in performance.

Table 9.10 compares the running times and iterations of three methods, the classical polyhedral analysis (CP), Simon and King’s method (SK) and our bounded polyhedral method (BD).

As can be seen, BD takes a few more iterations to terminate than SK due to the more complex polyhedra resulting. This also shows that SK takes more iterations than classical polyhedral analysis. It can be concluded that the increased cost of running BD compared to SK is quite limited.
9.5 Demonstration of Parametric WCET Analysis

The SWEET implementation of the parametric WCET analysis is, as mentioned, still in an early phase of development and is considered a work in progress. While most of individual phases on Figure 5.1 are implemented (except for the simplification phase) in SWEET and as external components, the implementation is not capable of a full comparative evaluation. However, we want to demonstrate that the implementation in SWEET has a basic functionality and that some benchmarks can be analysed and produce a parametric result.

Since the formulae resulting from analysing these benchmarks are not simplified, they are typically very long, but for the small benchmark fibcall (see Figure 9.3 for source code), the derived formula is manageable, and we display it here. In fibcall, the argument \( a \) is passed to the function computing the fibonacci number. If we remove the assignment \( a = 30 \) from fibcall, then \( a \) becomes an input variable, and our method yields the formula:

\[
PWCET_{\text{fibcall}}(a) = 10 + 10\min(\infty, 1, \min(Z(a), Z(a)) + \min(\infty, 1)) + 10\min(Z(a), Z(a)) + 10\min(\infty, 1, \min(\infty, 1) + \min(Z(a), Z(a))) + 10\min(Z(a), Z(a)) + 10
\]

where

\[
Z(a) = a \geq 2 ? a - 1 : 0
\]
9.5 Demonstration of Parametric WCET Analysis

Table 9.10

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Running Time</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP SK BD</td>
<td>1m 2.81s</td>
<td>1m 14.84s</td>
</tr>
<tr>
<td>CP SK BD</td>
<td>1m 24.62s</td>
<td>878</td>
</tr>
<tr>
<td>bsort100</td>
<td>4.64s</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>4.98s</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>5.33s</td>
<td>48</td>
</tr>
</tbody>
</table>

9.5 Demonstration of Parametric WCET Analysis

The SWEET implementation of the parametric WCET analysis is, as mentioned, still in an early phase of development and is considered a work in progress. While most of the individual phases in Figure 5.1 are implemented (except for the simplification phase) in SWEET and as external components, the implementation is not capable of a full comparative evaluation. However, we want to demonstrate that the implementation in SWEET has a basic functionality and that some benchmarks can be analysed and produce a parametric result.

Since the formulae resulting from analysing these benchmarks are not simplified, they are typically very long, but for the small benchmark `fibcall` (see Figure 9.3 for source code), the derived formula is manageable, and we display it here. In `fibcall`, the argument `a` is passed to the function computing the fibonacci number. If we remove the assignment `a = 30` from `fibcall`, then `a` becomes an input variable, and our method yields the formula:

\[
P_{\text{WCET}}(\text{fibcall}(a)) = 10 + \min(\infty, 1, \min(Z(a), Z(a))) + \min(\infty, 1) + \min(\infty, 1, \min(\infty, 1) + \min(Z(a), Z(a))) + 10
\]

where

\[
Z(a) = \begin{cases} a \geq 2 ? & 20(1 + a) : 40 \end{cases}
\]

This was achieved using classical polyhedral analysis, MPA as parametric calculation and assuming that each basic block takes 10 clock cycles. We see how the formula reflects the fact that if an argument below 2 is given, then the loop will not execute. We also see that the execution time is linear in the input `a`.

\[
\begin{align*}
\text{int fib(int n)} \\
& \{ \\
& \quad \text{int i, Fnew, Fold, temp, ans; } \\
& \quad \text{Fnew = 1; Fold = 0; } \\
& \quad \text{for ( i = 2; i <= n; i++) } \\
& \quad \{ \\
& \quad \quad \text{temp = Fnew; } \\
& \quad \quad \text{Fnew = Fnew + Fold; } \\
& \quad \quad \text{Fold = temp; } \\
& \quad \} \\
& \quad \text{ans = Fnew; } \\
& \quad \text{return ans; } \\
& \}
\end{align*}
\]

\[
\begin{align*}
\text{int main()} \\
& \{ \\
& \quad \text{int a; } \\
& \quad a = 30; \\
& \quad \text{fib(a); } \\
& \quad \text{return a; } \\
& \}
\end{align*}
\]

Figure 9.3: Source code of benchmark `fibcall`. 
Chapter 10

10.1 Introduction

In our research we have formalised, implemented and evaluated a method for performing parametric WCET analysis. This method is based on the observation that the number of states that can be visited at each point in a program is limited and if this number can be bounded in terms of the input to the program, then the WCET can also be bound in terms of the input. We have discovered the main strength and weaknesses of this method and have provided a number of solutions to issues that has come up, including a new efficient parametric calculation algorithm, reducing complexity by reducing the number of variables and making analyses sound and precise for fixed size integers.

10.2 Research Process

The main research conducted in this thesis has been around one method of investigating parametric WCET analysis. The method is based on Björn Lisper's original publication [76] and the research have been conducted by testing, implementation, experimenting and deductive methods. The general process have been to implement the method, discover issues and then attempt to solve those issues, which has incrementally resulted in a number of concrete contributions that takes us closer to a fully functioning and automatic parametric WCET analysis.
Chapter 10

Conclusions

10.1 Introduction

In our research we have formalised, implemented and evaluated a method for performing parametric WCET analysis. This method is based on the observation that the number of states that can be visited at each point in a program is limited and if this number can be bounded in terms of the input to the program, then the WCET can also be bound in terms of the input. We have discovered the main strength and weaknesses of this method and have provided a number of solutions to issues that has come up, including a new efficient parametric calculation algorithm, reducing complexity by reducing the number of variables and making analyses sound and precise for fixed size integers.

10.2 Research Process

The main research conducted in this thesis has been around one method of investigating parametric WCET analysis. The method is based on Björn Lisper’s original publication [76] and the research have been conducted by testing, implementation, experimenting and deductive methods. The general process have been to implement the method, discover issues and then attempt to solve those issues, which has incrementally resulted in a number of concrete contributions that takes us closer to a fully functioning and automatic parametric WCET analysis.
10.3 Main Results

10.3.1 Parametric WCET Analysis

Chapter 3 and 5 provided a thorough formalisation of our method along with proofs of its correctness (contribution \textbf{C1} in the introduction). This formalisation displays a very detailed model of how the method can be used which should make it simple to reproduce the results given in this thesis. These chapters also clearly state which assumptions the method use, although many of the assumptions are just to simplify the presentation and could easily be relaxed.

Our two implementations of the method described in Chapter 8 have helped to give a good understanding of the method and its properties. Using these implementation and the result of the experiments in Chapter 9 has led us to the following conclusions (contribution \textbf{C2} and \textbf{C5}):

- The method is able to get very detailed parametric formulae for programs independent of loop structure and program flow since it is based on general techniques such as abstract interpretation.

- The method is very flexible and can be used with various relational abstract domains, parametric calculation methods and methods of counting states.

- The method is very sensitive to which variables that matters to the timing when states are counted. It is very important to remove irrelevant variables from the analysed program.

- The final formula tends to be very large and complex even for small programs and usually it contains a lot of redundancy. A lot of simplification is needed in order to use it in practice.

- The bottleneck of the method is the parametric calculation. Careful trade-off between precision and efficiency has to be considered.

- In any relational abstract interpretation it is important to keep the number of variables to a minimum. Modelling individual elements in large arrays as variables in a relational domain is not feasible in practice.

- It is important to have a good and sound definition of what is a variable when analysing code on a low-level or immediate level code format such as ALF.
Our method is intended to be used on smaller programs, functions or components in order to get a good understanding of the timing behaviour in terms of the input and we conclude that this method, given some work really can provide this in a very detailed manner. So in conclusion to our research question

**Q1** Can Lisper’s method be used to efficiently find accurate parametric WCET estimates on program segments? What are the practical issues with the method, and does it need to be adjusted?

we think that the method has potential to be very useful. There still are some issues to be solved, especially when it comes to how variables are modelled, but overall the method is powerful. We have identified the main issues and properties of the method and what needs to be considered when implementing it.

### 10.3.2 Parametric Calculation

Since the bottleneck of the method was identified as the parametric calculation, a new, more efficient algorithm for parametric calculation is presented in Chapter 6 and evaluated in Chapter 9 (contribution C3). This new algorithm, MPA, is based on propagating structural constraints through the flow chart of the program in order to make sure that the implications of the constraints can be applied at all points in the program. MPA scales much better than the previously suggested calculation that uses parametric integer programming, while being slightly less precise and flexible. However, these costs in precision and flexibility is clearly worth, since MPA makes parametric calculation feasible. Comparing parametric integer programming with MPA has given the following insights:

- It is possible to reduce the number of variables in the parametric integer programming calculation, however it does not seem to reduce the complexity enough.
- There are alternative tools to Piplib, such as *isl* that seem to scale better than Piplib, but which still gets problems when the programs get larger. This suggests that the problem is in the method rather than in the tool.
- Parametric Calculation is feasible and scalable when using MPA.
- If efficiency is a problem, MPA can be made more efficient by restricting the tree-depth of the result. This usually does not affect precision too much (see Table 9.7 for details).
We draw the conclusion that we can give a positive answer to the research question

\textbf{Q2} Is it possible to use Lisper’s method more efficiently by replacing the parametric calculation with something better?

\subsection*{10.3.3 Analysis of Fixed Size Integers}

Since classical polyhedral analysis is not sound when wrap-around semantics is used. Wrap-around semantics is the most common semantics when analysing programs designed for embedded real-time systems, and due to the fact that processors with short word-length are often used, wrap-arounds may very well be present in analysed programs. For this reason we have adopted the polyhedral analysis used in the parametric WCET analysis to be sound even in the presence of wrap-arounds, presented in Chapter 7. We have done this by improving upon a previously suggested method by Simon & King (contribution \textbf{C4}). We have implemented both the previous method and our own fully bounded polyhedral analysis (see Chapter 8) and compared them. Our comparison of 450 different polyhedra shows that our method is always at least as precise as the previously suggested method, and showing strict improvement in 29\% of our tests (see Section 9.4). The observed improvements have been as high as 75\% less integer points inside a polyhedron in one case. Our implementation and experiments have shown that

\begin{itemize}
  \item There is limited research on making relational abstract interpretation sound for wrapping integers, and is a very interesting field for future research.
  \item Simon & King’s method introduces a lot of imprecision by explicit wrapping, especially for unbounded polyhedra.
  \item Our approach has the nice side effect that each polyhedron is fully bounded during abstract interpretation. This might have additional benefits.
\end{itemize}

In summary to the research question:

\textbf{Q3} Can the flow analysis used in \cite{76} be adapted or changed so that it is safe to use with integers with wrap-around behaviour without losing precision?

we would like to conclude that it is possible to use the parametric WCET method just as described and make it sound just by replacing the classical polyhedral domain with our fully bounded analysis.
10.4 Future Work

From the results we have got during our research, we have discovered several interesting paths that is worth exploring as well as issues that could be improved by further research. This section presents ideas of what areas could be interesting to conduct additional research in.

10.4.1 Evaluation

Our implementation of parametric WCET analysis in SWEET is still a work in progress, and it is in its current stage not mature enough to be used in an comparative evaluation or a case-study. It would, however, be interesting to see how practical the method is in a realistic industrial setting. In addition, it would be interesting to do a comparison to classical WCET analysis to see how the results differ, how much the “cost” of the extra complexity that parametric analysis is, and to find practical guidelines on how to use the method in practice.

10.4.2 Analysis on Low-level Code

There are several issues that are associated with analysing code on low-level or intermediate level. In our case, with the SWEET implementation, analysis is done on the intermediate code format ALF. Since ALF can be generated both from source code and object code, the issue of exactly what to treat like a variable in relational abstract interpretation is critical. Our initial approach to use frames as variables is a good start, but it fails to properly model individual elements of arrays and structs. In addition, this approach works best when the code is generated from source. If the code is generated from object code, the issue becomes even more complicated.

Another interesting thing is to develop more relational abstract domains that make analysis sound with wrapping semantics on integers. In this thesis, we have investigated the polyhedral domain. However, there are other relational abstract domains that could also be altered so that they are based on the assumption that overflows results in wrap-arounds. In our studies, we have discovered that such research is limited.
10.4.3 Slicing Techniques

We have seen during our experiments that the state count phase of our method sometimes can make gross over-approximations in the cases where the following two conditions simultaneously happen:

- No relevant information about the value of variable $x$ could be obtained from abstract interpretation at program point $q$.
- Variable $x$ does not directly or indirectly affect the number of times $q$ is visited.

It should be noted that this situation is reduced by program slicing (which eliminates the second condition), however, as mentioned, slicing is not exact and may leave non-control variables in the analysed program. Moreover, a variable may affect the program flow, but still not affect the number of times $q$ is visited. An example is a loop invariant variable, which does not affect the number of times the program point $q$ is affected when it is in a loop, but may affect program flow elsewhere.

The solution to this problem may be to introduce a more aggressive slicing, especially tailored for to be used with the Census method. This would be very interesting to investigate in order to make the Census method more practical.

10.4.4 Abstract Domains

There are many abstract domains available in literature which can capture different interesting property. To be able to capture non-unit strides in loops it is possible to combine the polyhedral domain with the linear congruence domain to obtain sparse convex polyhedra. This is implemented in the library Polylib [88], but has not yet been explored in this context.

10.4.5 The Minimum Propagation Algorithm

While the minimum propagation algorithm works quite well in practice, there are still things that can be done to improve it. For instance, more (memory) efficient data types could possibly be invented. In addition, we plan to investigate the possibility of adding additional constraints (such as infeasible path information) to MPA. Finally, future work is to see if the over-estimation can be reduced.
10.4.3 Slicing Techniques
We have seen during our experiments that the state count phase of our method sometimes can make gross over-approximations in the cases where the following two conditions simultaneously happen:

• No relevant information about the value of variable \( x \) could be obtained from abstract interpretation at program point \( q \).
• Variable \( x \) does not directly or indirectly affect the number of times \( q \) is visited.

It should be noted that this situation is reduced by program slicing (which eliminates the second condition), however, as mentioned, slicing is not exact and may leave non-control variables in the analysed program. Moreover, a variable may affect the program flow, but still not affect the number of times \( q \) is visited. An example is a loop invariant variable, which does not affect the number of times the program point \( q \) is affected when it is in a loop, but may affect program flow elsewhere.

The solution to this problem may be to introduce a more aggressive slicing, especially tailored for to be used with the Census method. This would be very interesting to investigate in order to make the Census method more practical.

10.4.4 Abstract Domains
There are many abstract domains available in literature which can capture different interesting properties. To be able to capture non-unit strides in loops it is possible to combine the polyhedral domain with the linear congruence domain to obtain sparse convex polyhedra. This is implemented in the library Polylib [88], but has not yet been explored in this context.

10.4.5 The Minimum Propagation Algorithm
While the minimum propagation algorithm works quite well in practice, there are still things that can be done to improve it. For instance, more (memory) efficient data types could possibly be invented. In addition, we plan to investigate the possibility of adding additional constraints (such as infeasible path information) to MPA. Finally, future work is to see if the over-estimation can be reduced.

Bibliography


[29] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze and transform scientific programs. In *ICS ’96: Proceedings of the 10th international*


[92] Polylib website, 2009. icps.u-strasbg.fr/polylib/.


