Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-MS-2014-10

M.Sc. Thesis

Determining Performance Boundaries and Automatic Loop Optimization of High-Level System Specifications

Wouter van Teijlingen

Abstract

Designers are confronted with high time-to-market pressure and an increasing demand for computational power. As a result, they are required to identify as early as possible the quality of a specification for an intended technology. The designer needs to know if this specification can be improved, and at what cost. Specification trade-offs are often based on the experience and intuition of a designer, which in itself is not enough to make design decisions given the complexity of modern designs. Therefore, we need to identify the performance boundaries for the execution of a specification on an intended technology.

The degree of parallelism, required resources, scheduling constraints, and possible optimizations, etc. are essential in determining design trade-offs (e.g., power consumption, execution time, etc). However, existing tools lack the capability of determining relevant performance parameters and the option to automatically optimize high-level specifications to make meaningful design trade-offs.

To address these problems, we present in this thesis a new profiler tool, *cprof*. The Clang compiler front-end is used in this tool to parse high-level specifications, and to produce instrumented source code for the purpose of profiling. This tool automatically determines, from high-level specifications, the degree of parallelism of a given source code, specified in C and C++ programming languages. Furthermore, cprof estimates the number of clock cycles necessary to complete a program, it automatically applies loop optimization techniques, it determines the lower and upper bound on throughput capacity, and finally, it generates hardware execution traces. The tool assumes that the specification is executed on a parallel *Model of Computation* (MoC), referred to as a *Polyhedral Process Network* (PPN).

The proposed tool adds new functionality to existing technologies: the estimated performance by cprof of PolyBench/C benchmarks, as compared to realistic implementations in *Field-Programmable Gate Arrays* (FPGA) platforms, showed to be almost identical. Cprof is capable of estimating the lower and upper bound on throughput capacity, making it possible for the designer to make performance trade-offs based on real design points. As a result, only the high-level specification is used by cprof to assist in *Design Space Exploration* (DSE) and to improve design quality.

Determining Performance Boundaries and Automatic Loop Optimization of High-Level System Specifications

Profiling of Polyhedral Process Networks

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Engineering

by

Wouter van Teijlingen
born in Leiderdorp, The Netherlands

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Determining Performance Boundaries and Automatic Loop Optimization of High-Level System Specifications"** by **Wouter van Teijlingen** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: November 28, 2014

Chairman:

prof. dr. ir. A. J. van der Veen

Advisors:

dr. ir. T. G. R. M. van Leuken

dr. ir. A. C. J. Kienhuis

Committee Members:

dr. C. Galuzzi

dr. ir. J. S. S. M. Wong

Abstract

Designers are confronted with high time-to-market pressure and an increasing demand for computational power. As a result, they are required to identify as early as possible the quality of a specification for an intended technology. The designer needs to know if this specification can be improved, and at what cost. Specification trade-offs are often based on the experience and intuition of a designer, which in itself is not enough to make design decisions given the complexity of modern designs. Therefore, we need to identify the performance boundaries for the execution of a specification on an intended technology.

The degree of parallelism, required resources, scheduling constraints, and possible optimizations, etc. are essential in determining design trade-offs (e.g., power consumption, execution time, etc). However, existing tools lack the capability of determining relevant performance parameters and the option to automatically optimize high-level specifications to make meaningful design trade-offs.

To address these problems, we present in this thesis a new profiler tool, *cprof*. The Clang compiler front-end is used in this tool to parse high-level specifications, and to produce instrumented source code for the purpose of profiling. This tool automatically determines, from high-level specifications, the degree of parallelism of a given source code, specified in C and C++ programming languages. Furthermore, cprof estimates the number of clock cycles necessary to complete a program, it automatically applies loop optimization techniques, it determines the lower and upper bound on throughput capacity, and finally, it generates hardware execution traces. The tool assumes that the specification is executed on a parallel *Model of Computation* (MoC), referred to as a *Polyhedral Process Network* (PPN).

The proposed tool adds new functionality to existing technologies: the estimated performance by cprof of PolyBench/C benchmarks, as compared to realistic implementations in *Field-Programmable Gate Arrays* (FPGA) platforms, showed to be almost identical. Cprof is capable of estimating the lower and upper bound on throughput capacity, making it possible for the designer to make performance trade-offs based on real design points. As a result, only the high-level specification is used by cprof to assist in *Design Space Exploration* (DSE) and to improve design quality.

Acknowledgments

First of all, I would like to express my gratitude to my advisor, professor Rene van Leuken. Without your support, this work would never have come into existence. I would like to thank Carlo Galuzzi, for taking an interest in my work and for proofreading my thesis.

I am also very grateful to my second advisor, professor Bart Kienhuis. Thank you for introducing me into the world of polyhedral process networks, and for sharpening my writing skills. I hope to continue our collaboration in the future

I am indebted to my colleague Johan Peltenburg, for taking the time to read my thesis and for providing me with feedback. I would like to thank my colleagues at the Rotterdam University of Applied Sciences, for taking an interest in my work.

Noela, thank you for your love and unconditional support. I would like to thank my family, and in particular, my parents, Piet and Petra. Thank you for your limitless support throughout the years.

Wouter van Teijlingen
Delft, The Netherlands
November 28, 2014

Contents

Abstract v

Acknowledgments vii

List of Figures xiv

List of Tables xv

List of Algorithms xvii

List of Acronyms xix

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Problem statement . . . 3
- 1.3 Goals and contributions . . . 4
- 1.4 Synopsis and Outline . . . 4

2 Background 5
- 2.1 Models of Computation . . . 5
  - 2.1.1 Kahn Process Networks . . . 5
  - 2.1.2 Polyhedral Process Networks . . . 5
- 2.2 Deriving Polyhedral Process Networks . . . 6
  - 2.2.1 Linearization and Communication Models . . . 7
- 2.3 Static Affine Nested Loop Programs . . . 8
  - 2.3.1 Overview . . . 8
  - 2.3.2 Applied SANLPs . . . 8
  - 2.3.3 Iteration Domain and Dependencies . . . 9
  - 2.3.4 Transformations . . . 10
- 2.4 The LLVM/Clang Compiler Infrastructure . . . 10
- 2.5 Hierarchical Program Analysis . . . 11
- 2.6 High-Level Synthesis Tools . . . 11
- 2.7 Summary and Conclusions . . . 12

3 Related Work 13
- 3.1 Simulation . . . 13
- 3.2 Analytical Estimation . . . 13
- 3.3 Profiling . . . 14
  - 3.3.1 General-Purpose Profilers . . . 14
  - 3.3.2 Hardware Profilers . . . 14
  - 3.3.3 Parallel and Memory Profilers . . . 14
  - 3.3.4 Critical Path Analysis . . . 15
- 3.4 Summary and Conclusions . . . 16

4 Solution Approach 17
- 4.1 Concepts . . . 17
  - 4.1.1 Basic Calibration . . . 18
  - 4.1.2 Conditional Synchronization . . . 19
  - 4.1.3 Conditional Control Flow . . . 20
  - 4.1.4 Mutual Exclusion . . . 20

4.2 Performance Estimation . . . 20
4.2.1 Absolute Throughput Estimation . . . 20
4.2.2 Unbounded Throughput Estimation . . . 21
4.3 Case Studies . . . 21
4.3.1 Case Study: Absolute Throughput . . . 21
4.3.2 Case Study: Unbounded Throughput . . . 23
4.4 Shadow Variables . . . 26
4.5 Control Variables . . . 26
4.6 Statement Execution Profile . . . 27
4.7 Global Execution Profile . . . 27
4.8 Flow Dependencies . . . 28
4.9 Summary and Conclusions . . . 28

5 Design and Implementation **29**
5.1 Overview . . . 29
5.2 Input Processing . . . 30
5.2.1 Input Specification . . . 30
5.2.2 AST Construction . . . 31
5.3 Static Analysis and Instrumentation . . . 31
5.3.1 Static Analysis . . . 32
5.3.2 Instrumentation . . . 34
5.3.3 Source-to-Source Transformations . . . 34
5.4 Dynamic Analysis . . . 35
5.4.1 Compilation and Initialization . . . 35
5.4.2 Algorithms for Dynamic Analysis . . . 35
5.5 Performance Analysis . . . 39
5.5.1 Data Processing and Presentation . . . 39
5.5.2 Waveform Generation . . . 40
5.5.3 Program Profile Generation . . . 40
5.6 Optimization . . . 41
5.6.1 Methods . . . 41
5.6.2 Implementation of Optimizations . . . 42
5.7 Hierarchical Program Analysis . . . 43
5.7.1 Static Analysis . . . 43
5.7.2 Instrumentation . . . 44
5.7.3 Dynamic Analysis . . . 45
5.7.4 Performance Analysis . . . 45
5.8 The Cost of Profiling . . . 46
5.9 Summary and Conclusions . . . 46

6 Verification **47**
6.1 Verification Approach . . . 47
6.2 Verification of the Communication Models . . . 47
6.2.1 In-Order without Multiplicity (IOM-) . . . 48
6.2.2 In-Order with Multiplicity (IOM+) . . . 49
6.2.3 Out-of-Order without Multiplicity (OOM-) . . . 49
6.2.4 Out-of-Order with Multiplicity (OOM+) . . . 50
6.2.5 Results . . . 50
6.3 Verification of the Absolute and Unbounded Throughput Estimates . . . 51
6.3.1 The Predictor Program . . . 51
6.3.2 Optimization of Predictor . . . 53
6.3.3 Results . . . 55
6.4 Verification of Hierarchy Program Analysis . . . 56

6.4.1 The Hierarchy Program . . . 57
6.4.2 Results . . . 57
6.5 Summary and Conclusions . . . 58

7 Results **59**
7.1 Experimental Setup . . . 59
7.2 Absolute Throughput Estimates of PolyBench/C . . . 59
7.2.1 Execution Times . . . 59
7.2.2 The Average and Maximum Degree of Parallelism . . . 60
7.3 Unbounded Throughput Estimates of Polybench/C . . . 61
7.3.1 Execution Times . . . 61
7.3.2 The Average and Maximum Degree of Parallelism . . . 61
7.4 RTL Simulations . . . 62
7.5 Design Space Boundaries . . . 63
7.6 Optimization . . . 63
7.7 Scalability . . . 64
7.8 Summary and Conclusions . . . 66

8 Conclusions and Future Work **67**
8.1 Contributions . . . 68
8.2 Future Work . . . 68

Bibliography **73**

A Compiler Extension for Compaan DDE **75**
A.1 Introduction . . . 75
A.1.1 Design and Implementation of the Compiler Extension . . . 75

B Cprof Usage Instructions **79**
B.1 Introduction . . . 79
B.2 Installation . . . 79
B.3 Usage . . . 79

C Predictor Optimized Versions **81**
C.1 Inner Loop Unrolled . . . 81
C.2 Outer loop Unrolled . . . 81
C.3 Inner/Outer Loops Unrolled . . . 82
C.4 Inner/Outer/Sink Loops Unrolled . . . 83
C.5 Source/Inner/Outer Loops Unrolled . . . 84
C.6 Source/Inner/Outer/Sink Loops Unrolled . . . 85

D PolyBench/C 3.1 Benchmarks **89**

E Support of Control Flow Architectures in Cprof **91**
E.1 Modification of Algorithms . . . 91
E.1.1 Read Operations . . . 91
E.1.2 Write Operations . . . 92

F Verification Waveforms **93**
F.1 IOM- Waveforms . . . 94
F.2 IOM+ Waveforms . . . 95
F.3 OOM- Waveforms . . . 96
F.4 OOM+ Waveforms . . . 97

List of Figures

1.1 Exploring the Design Space, using the absolute and unbounded throughput estimation. 2
1.2 Traditional design flow in high-level synthesis. . . . 2
1.3 Reducing the feedback loop in the design flow in high-level synthesis. . . . 3

2.1 An example of a Polyhedral Process Network with FIFOs between processes. . . . 6
2.2 Derivation of Polyhedral Process Network. . . . 7
2.3 Communication models for Polyhedral Process Networks. . . . 8
2.4 Dependency analysis of Listing 2.2. . . . 9
2.5 Selection of independent computational tasks. . . . 10
2.6 LLVM Compiler Infrastructure. . . . 11

3.1 Level of accuracy and complexities in performance estimation. . . . 13

4.1 Mapping of a C program to a Polyhedral Process Network. . . . 17
4.2 Example IP block. . . . 18
4.3 The initiation interval and function latency of the execute stage. . . . 19
4.4 The Read, Write, and Execute units in a polyhedral process. . . . 19
4.5 Pipelined execution of the read, execute and write stages. . . . 19
4.6 Mutual Exclusion in Processes. . . . 20
4.7 Absolute and unbounded throughput. . . . 21
4.8 The shadow variables and their values for absolute throughput estimates. . . . 22
4.9 Control variables associated with each process in a polyhedral process network. . . . 22
4.10 Absolute throughput statement profiles. . . . 23
4.11 Example C Program. . . . 24
4.12 The shadow variables and their values for unbounded throughput estimates. . . . 24
4.13 Determiniation of control variables for unbounded throughput. . . . 25
4.14 Unbounded throughput statement profiles. . . . 26

5.1 Overview of the cprof profiler. . . . 29
5.2 Example of static analysis. . . . 32
5.3 The canonical declaration and its relation to variable references. . . . 33
5.4 Serialization of cprof objects. . . . 33
5.5 Source code after instrumentation by cprof. . . . 34
5.6 Overview of algorithms used in dynamic analysis. . . . 36
5.7 Overview of processing and presenting the output of dynamic analysis. . . . 40
5.8 Performance of the function bar after a closer inspection. . . . 40
5.9 Generation of program profile. . . . 41
5.10 Sample program to show optimizations. . . . 41
5.11 Modulo unfolding applied. . . . 42
5.12 Plane cutting aplied. . . . 42
5.13 Overview of the optimization flow in cprof. . . . 43
5.14 Example program with hierarchy used for HPA. . . . 44
5.15 The hierarchy function instrumented with support for HPA. . . . 45
5.16 Inter-procedural relations between variables. . . . 45

6.1 Communication models in PPNs. . . . 48
6.2 Example implementation of the IOM- communication model. . . . 48
6.3 Example implementation of the IOM+ communication model. . . . 49
6.4 Example implementation of the OOM- communication model. . . . 49
6.5 Example implementation of the OOM+ communication model. . . . 50

6.6 Execution times of the communication models measured by cprof and ISim. 51
6.7 The source code and derived PPN of predictor. 51
6.8 The dependency graph of the `transformer` function. 52
6.9 The absolute and unbounded throughput estimates for the predictor. 52
6.10 Iteration dependencies in the predictor. 53
6.11 Two possible optimizations of predictor. 53
6.12 Polyhedral process network of the predictor with the inner and outer loop unrolled. . . 54
6.13 Execution times of the predictor measured by cprof and by Xilinx ISim. 55
6.14 The average and maximum degree of parallelism available in the predictor. 56
6.15 The absolute and unbounded throughput estimates for hierarchy. 57
6.16 The execution finish times of hierarchy. 58

7.1 Absolute throughput estimates of the execution finish times of PolyBench/C kernels. . 60
7.2 Absolute throughput estimates of the average and maximum degree of parallelism in PolyBench/C kernels. 60
7.3 Unbounded throughput estimates of the execution finish times of PolyBench/C kernels. 61
7.4 Unbounded throughput estimates of the average and maximum degree of parallelism in PolyBench/C kernels. 62
7.5 Absolute throughput estimates of the execution finish time of PolyBench/C kernel versus the execution finish time of RTL implementations. 62
7.6 Design space boundaries: the average degree of parallelism found by the absolute and unbounded throughput estimates. 63
7.7 Optimization of the `atax` benchmark. 64
7.8 Estimates of the average and maximum degree of parallelism of the atax benchmark for various data sets. 65
7.9 Estimates of the execution time of the atax benchmark for various data sets. 65

A.1 Overview of the compiler extension. 75
A.2 Compiler Extension. 76

F.1 Waveforms representing the execution of the program implementing the IOM- communication model. 94
F.2 Waveforms representing the execution of the program implementing the IOM+ communication model. 95
F.3 Waveforms representing the execution of the program implementing the OOM- communication model. 96
F.4 Waveforms representing the execution of the program implementing the OOM+ communication model. 97

List of Tables

5.1 Profiling cost in space and time. Used notations: c is the number of dimensions of the read argument, r is the number times the read arguments are referenced throughout the program, d is the number of dimensions of the write argument, w is the number times the write arguments are referenced throughout the program, and m is the number of statement intervals. 46

6.1 Pipeline efficiency of the predictor. 52
6.2 Pipeline efficiency of the predictor after unrolling the inner or outer loop. 54
6.3 Resource cost of the predictor after unrolling the inner or outer loop. 56

7.1 PolyBench/C data set specifications. 64
7.2 Time spent by cprof on estimating the performance of atax, and by Compaan DDE to generate a hardware implementation of atax. 66

D.1 PolyBench/C Benchmarks. 89

List of Algorithms

1 Update shadow variables for read access. 37
2 Update the execute statement profiles. 37
3 Update shadow variables for write access. 38
4 Update algorithm for the statement execution profiles. 39
5 Substituting algorithm for inserting valid statements into the AST. 76
6 Update shadow variables for read access, with support for anti and output dependencies. 91
7 Update shadow variables for write access, with support for anti and output dependencies. 92

List of Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree
BRAM	Block Random Access Memory
CAS	Circuits and Systems
CFG	Control Flow Graph
CPA	Critical Path Analysis
CPN	C for Process Networks
DBA	Dynamic Binary Analysis
DBI	Dynamic Binary Instrumentation
DDE	Design Development Environment
DPN	Dataflow Process Network
DSE	Design Space Exploration
ESL	Electronci System Level
FF	Flip Flop
FIFO	First-In, First Out
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GPU	Graphics Processing Unit
HCPA	Hierarchical Critical Path Analysis
HDL	Hardware Design Language
HLS	High-Level Synthesis
HPA	Hierarhical Program Analysis
ILP	Integer Linear Programming
ILP	Instruction-Level Parallelism
IP	Intellectual Property
IR	Intermediate Representation
KPN	Kahn Process Network
LAURA	Leiden Architecture Research and Exploration Tool
LUT	Lookup Table
MAPS	MPSoC Application Programming Studio
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MoC	Model of Computation
MPSoC	Multile-Processor Systems-on-Chip
PLB	Parallel Block Vectors
PPN	Polyhedral Process Network
RAW	Read After Write
RHS	Right-hand Side
RTL	Register Transfer Level
SANLP	Static Affine Nested Loop Programs
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SSA	Static Single Assignment
UUID	Universally Unique Identifier
VCD	Value Change Dump
WAR	Write After Read
WAW	Write After Write
WPA	Whole Program Analysis
WPP	Whole Program Paths

XML	Extensible Markup Language

1 Introduction

Engineers are dealing with high time-to-market pressure and demanding design constraints. One area where these two issues are prominent is that of embedded systems. In 1997, the number of shipped embedded systems already matched that of personal computers [1]. Designing an embedded system is simplified by the use of high-level synthesis (HLS). HLS is a design process, which allows to specify systems at a higher level of abstraction, e.g., in the C programming language. This high-level specification is transformed into digital hardware and can be implemented in, for example, a *Field-Programmable Gate Array* (FPGA).

In this work, we use HLS to map sequential C code to *Polyhedral Process Networks* (PPN), a parallel *Model of Computation* (MoC) [2]. We consider a special type of C programs, called *Static Affine Nested Loop Programs* (SANLPs) that can automatically be converted by a compiler into a PPN. SANLPs are used for modeling time critical parts of audio/video stream-based and DSP applications [2].

We can significantly reduce the time required for design feedback, if it is possible to find limits on the performance of an embedded system, based on the C code only. Moreover, there is a reduction in risk, because engineers know from the beginning whether a design meets its specifications. This concept is shown in Figure 1.1. It shows that for a particular design, we can find a lower bound and an upper bound of the design space. The lower bound is the performance of the design without any specific optimizations. The upper bound is the performance of the design if there is an infinite amount of resources.

The shaded area indicates performance that is unattainable for this particular design. Most likely, the two design extremes are not the performance a designer wants. The designer probably wants a design somewhere in between these two points. Using design space exploration, a designer can establish the line between the two extremes and select a feasible design point. To establish the lower and upper bound in Figure 1.1 and the line between them, we present in this thesis the *cprof* profiler. Cprof estimates the performance of a system specified in C code and gives the performance of that code, when implemented as a PPN in hardware.

The two design extremes are referred to as the *absolute* and the *unbounded throughput* in Figure 1.1. The performance of a PPN is represented by the absolute and unbounded throughput, using the execution finish time, the average and maximum degree of parallelism, as metrics of performance. In Figure 1.1, the design space is bound by the average degree of parallelism found by the absolute and unbounded throughput estimates. Profiling is a technique for quickly determining the performance of programs during run-time. Cprof allows a designer to make modifications in the C-code of a system, and subsequently gives feedback on the performance of the system. This way, cprof assists in design space exploration.

This project is a joint effort of the Circuits and Systems (CAS) group at Delft University of Technology and Compaan. Compaan is a spin-off from Leiden University, and is a privately owned company. Compaan provides services and tools for automatic conversion of streaming algorithms to heterogeneous platforms. This introduction is organized as follows. The motivation for this work is presented in Section 1.1. We continue with the problem statement in Section 1.2. In Section 1.3 the goals and contributions of this thesis are discussed. Finally, Section 1.4 concludes this chapter and presents a synopsis and an outline of this work presented in the following chapters.

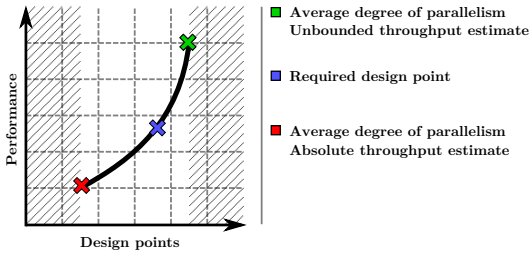


Figure 1.1: Exploring the Design Space, using the absolute and unbounded throughput estimation.

1.1 Motivation

The main motivations for this work are to boost engineering productivity, and to provide performance insights as early as possible, as a means of risk reduction. We can increase productivity by assisting in *Design Space Exploration* (DSE), and reduce risk by making design limitations explicit as early as possible. Specifying designs at the *Register Transfer Level* (RTL) is time-consuming and error-prone [3]. Instead, a higher level of abstraction is used to simplify the design process. This is what we call high-level synthesis. HLS leads to the design flow shown in Figure 1.2. In this work, we use Compaan [4] and Daedalus [5] for the design flow depicted in Figure 1.2. The figure shows that C-code is converted into a system-level specification that, via synthesis, is mapped onto either an FPGA or another platform.

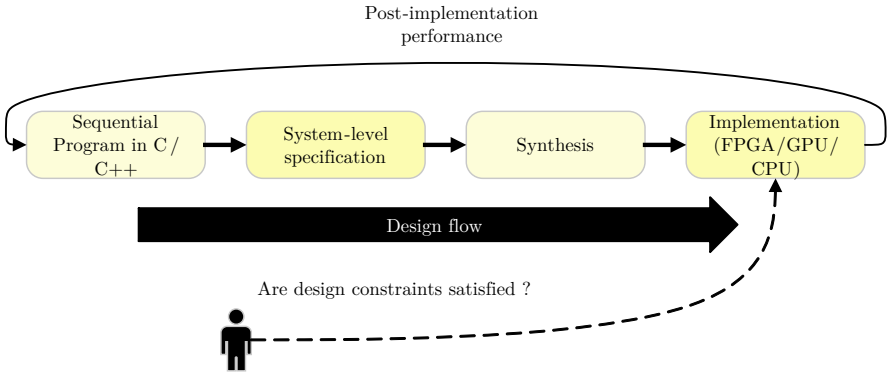


Figure 1.2: Traditional design flow in high-level synthesis.

A designer is not solely interested in converting C-code into a design that is implementable in hardware. He is interested in obtaining a system that meets specific design constraints. For example, a designer may need to process 25 *frames per second* (FPS) in the case of a video application. The problem with the traditional design flow, shown in Figure 1.2, is that it takes long time before the designer knows if the design meets the constraints. Furthermore, if the constraints are not met, the designer needs to know if it is actually possible to meet the constraints with the given C-code. If so, how should the designer modify the C-code in such a way that the system can process the 25 FPS on an FPGA?

In this thesis, we present a modified flow that uses profiling. Profiling is a technique used to analyze the behavior of programs during execution. This technique is very fast, and helps reducing the feedback loop, as depicted in Figure 1.3. In the modified design flow, a designer uses profiling to establish the design limits, as presented in Figure 1.1. This provides immediate feedback on whether the design can satisfy the constraints at all. If she needs 25 FPS, but the upper bound is at 20 FPS, it means that she can never satisfy the constraints as the 25 FPS is within the shaded area. If the upper bound is at 40 FPS, and the lower bound is 10 FPS, the designer knows that realizing a design capable of 25 FPS is indeed possible.

The next step will be to modify the C-code to increase parallelism in the application until the design reaches 25 FPS. Only at that time, the designer commits to the very time consuming design flow to make implementable hardware. The designer knows she will get a design in hardware that meets her constraints. Another important benefit of the modified design flow is that no specific hardware knowledge is needed to actually realize designs. Using cprof, a software designer can make design changes in the C-code and estimates its performance. Once a particular design point is obtained, the design is committed to a hardware flow. At that point, specific hardware knowledge is required.

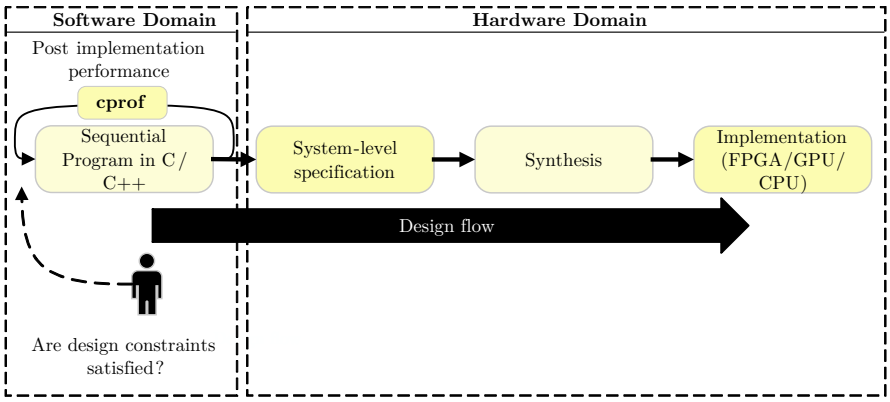


Figure 1.3: Reducing the feedback loop in the design flow in high-level synthesis.

1.2 Problem statement

The current design flow of Compaan and Deadalus is represented by the traditional design flow shown in Figure 1.2. The modified design flow presented in Figure 1.3 is the desired design flow. The desired design flow increases engineering productivity by assisting in DSE, and risk is reduced by making design limitations explicit at an early stage in the design process. Sven van Haastregt has presented in his thesis various techniques to obtain the performance of a PPN [2]. Cprof originated in his work, and he shows that the techniques used by cprof give a reliable performance estimate, with reduced cost in terms of time and effort in comparison to other techniques, such as simulation.

This technique is only shown for a small number of examples, and estimating the performance of programs with inter-procedural behavior is impossible. Furthermore, the optimization of designs is given as future work in [2]. It is unclear whether cprof can be applied to any SANLPs supported by

Compaan or Daedalus, and with what accuracy. The problem we address in this thesis is whether we can further develop cprof, such that it can handle any SANLP, and to assess the accuracy of the technique. Furthermore, we want to automatically optimize designs to assist in DSE.

1.3 Goals and contributions

The primary goal of this thesis is to develop a profiler, that is based on the principles presented by Sven van Haastregt [2]. We refer to the profiler as *cprof*. To show that cprof can handle complex SNALPs, we validated cprof against the PolyBench/C benchmarks [6]. This benchmark consists of 25 mathematical kernels that are used in data processing. We show that we can estimate the performance of each of these kernels, with on average, an overestimation of 0.44%. This shows that cprof is a very valuable tool to assess the performance of a SNALP very early. The use of the developed cprof profiler converts the traditional design flow of Compaan and Daedalus into the modified flow shown in Figure 1.3.

The main contributions of the work presented in this thesis are the following:

- The development of the cprof profiler for estimating the absolute and unbounded throughput in C-code.
- A compiler plugin based on LLVM/Clang for the transformation of unsupported statements into a form accepted by the profiler.
- The implementation of cprof in LLVM/Clang with support for the C and C++ programming languages.
- The optimization of source code to assist in DSE.
- The support for Hierarchical Program Analysis to estimate the behavior of systems with inter-procedural behavior.
- The validation of the results against hardware implementations of PolyBench/C benchmarks.

1.4 Synopsis and Outline

Engineers need tools to deal with the increasing complexity of hardware and software development. To reduce the risk in the design of embedded systems and to increase engineering productivity, we developed a profiler called cprof. With cprof, engineers can quickly determine the design space for the lower and upper bounds to parallelism, and apply source code transformations to explore the design space. *In this thesis, we show that cprof is capable of estimating the absolute and unbounded throughput of the PolyBench/C benchmarks.*

The remainder of this thesis has the following outline. In Chapter 2, the required background and precise definitions of the problem is presented. In Chapter 3, related work and state of art are discussed, including the proposed contributions and reflections on why current solutions are not sufficient. The solution approach is presented in Chapter 4. The design and implementation of the profiler is discussed in Chapter 5. In Chapter 6, various examples and cases to verify the functionality of the profiler are presented. In Chapter 7, case studies are discussed and the experimental results evaluated. Finally, conclusions are drawn and directions for future research are provided in Chapter 8.

2 Background

In this chapter, we present the required concepts and nomenclature used throughout this work. In Section 2.1, polyhedral process networks are defined. In Section 2.2, the derivation of polyhedral process networks is explained. Following in Section 2.3, a class of programs is introduced that can be transformed to polyhedral process networks. In Section 2.4, the compiler infrastructure used in this work is explained. In Section 2.5, there is an introduction to inter-procedural program analysis. In Section 2.6 a brief overview of high-level synthesis tools is provided. Summary and conclusions are presented in Section 2.7.

2.1 Models of Computation

Designers often specify algorithms in a sequential programming language. In such languages, it is not possible to express parallelism without special programming directives and libraries. However, the specification of algorithms for parallel computing is a difficult and time-consuming task, because humans tend to think sequentially. Designers are in need of model of computations that take as input a sequential specification, but that support the automatic derivation of parallel networks.

2.1.1 Kahn Process Networks

The *Kahn Process Network* (KPN) [7] is a distributed model of computation. Named after Dr. Gilles Kahn, who was responsible for introducing this model of parallel computation in 1974. Processes in a KPN communicate with each other via unbounded *First-In, First-Out* (FIFO) data channels. The behavior of process networks is deterministic, and is not disturbed by timing variations in computation and communication. Reads and writes in the process network are blocking and non-blocking, respectively. Process networks are deterministic, as the use of blocking read guarantees that the system behavior is always the same, whether a sequential schedule, fully parallel or a schedule in-between is used. Deterministic behavior is a desirable property in embedded applications, as it guarantees always the same system behavior.

2.1.2 Polyhedral Process Networks

The polyhedral process network is a dataflow-based MoC that is a specialization of KPNs. Programs specified as PPNs have static control flow; all loop bounds, conditions and array index expressions are such that they are represented by affine expressions. In PPNs, the computer program is represented by geometrical properties called polyhedra. Hence, the name polyhedral process networks. Polyhedra describe the program in a finite number of linear inequalities. Polyhedra are used for representing loops, which iterate over a finite and parameterized set of iterations found in the computer program. As a result, it is possible to analyze and optimize such objects with geometrical and combinatorial techniques, such as *Integer Linear Programming* (ILP).

In PPNs, each process is divided into three distinctive stages:

- **Read (R)**: in this stage, the process reads input data from the communication channel. If none is available, the process blocks.
- **Execute (E)**: in this stage, the process executes a computational function on the available data and produces data as output.

- **Write (W)**: in this stage, the process writes data to the outbound communication channels.

It is possible for processes to have a read or write phase, or both. In this thesis, we use the following definition for PPNs [2]:

Definition 2.1 (Polyhedral Process Network)**.** A *Polyhedral Process Network* (PPN) is modeled as a directed graph $(\mathcal{P}, \mathcal{E})$, where $\mathcal{P}$ is a finite set of vertices representing processes, and $\mathcal{E}$ is a finite set of edges representing communication channels. Each process $p_i \in \mathcal{P}$ is characterized by:

- a list of input ports responsible for reading all function input arguments from the inbound FIFO channels;
- a function responsible for processing the inputs and producing output arguments;
- a list of output ports responsible for writing all function output arguments to the outbound FIFO channels.

Each channel $c_i \in \mathcal{E}$ is characterized by:

- a source process;
- a destination process;
- a channel type;
- the buffer size.

Figure 2.1 shows an example of a PPN, which consists of 3 processes: P1, P2 and P3. Process P1

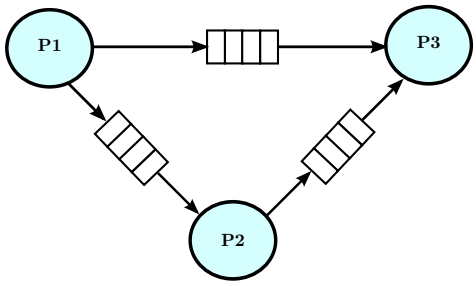


Figure 2.1: An example of a Polyhedral Process Network with FIFOs between processes.

communicates through FIFOs with processes P2 and P3. Process P3 reads the data from process P1 and P2. If one of the FIFOs connected to P3 is empty, the process P3 will block until data becomes available.

2.2 Deriving Polyhedral Process Networks

The PPNs are derived from sequential C or C++ code in a number of steps, shown in Figure 2.2. Statements in the program code are analyzed for dependencies, and a dependency graph is constructed. Linearization is the process of mapping higher-order data-structures to one-dimensional representations. Linearization is applied, as all memory access is mapped on FIFO channels.

After linearization, communication models are determined and a polyhedral process network is generated. After deriving the PPN it is possible to calculate communication channel sizes, and optimize the PPN to minimize the number of channels.

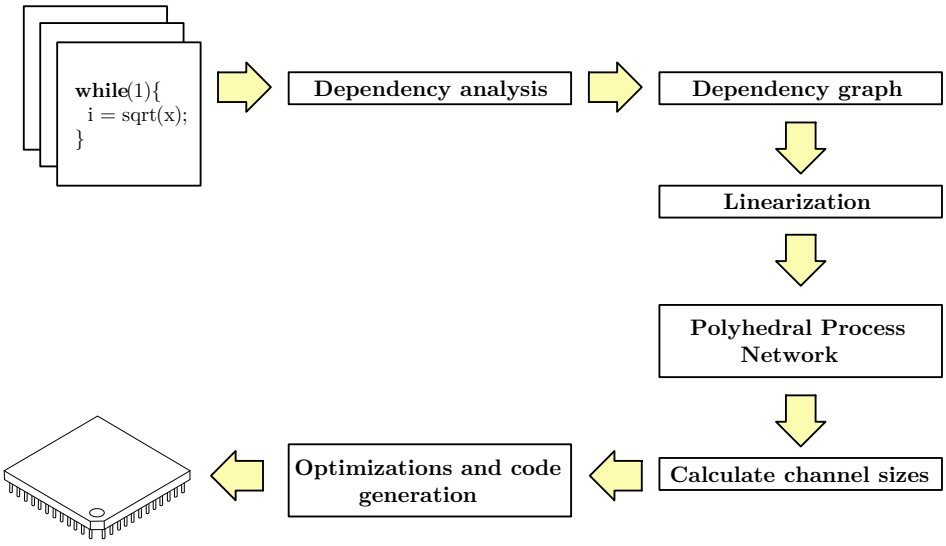


Figure 2.2: Derivation of Polyhedral Process Network.

2.2.1 Linearization and Communication Models

In the process of generating a PPN, the N-dimensional data-structures are mapped onto 1-dimensional streams in the linearization stage. The indexing of N-dimensional data-structures is replaced by two primitives, *Get* and *Put*, representing read and write operations on a FIFO buffer, respectively. In the linearization stage, each communication channel is identified as being one of the types given in [8]:

- *In-order without multiplicity* (IOM-): in this model, the *Get* and *Put* primitives are used on the FIFO buffer without considering the life-time of a communication token.
- *In-order with multiplicity* (IOM+): in this model, the life-time of a communication token is considered to address its multiplicity.
- *Out-of-order without multiplicity* (OOM-): in this model, the consumer process is equipped with reordering memory and a controller to reorder the memory.
- *Out-of-order with multiplicity* (OOM+): in this model, the life-time of a communication token is taken into account in the reordering controller. The controller is responsible for releasing memory if the life-time of a token has come to its end.

In Figure 2.3, the different communication models are shown.

In Figure 2.3(a), the IOM- model is shown. This model is the easiest to implement, as there is no need for reordering, and tokens are immediately used by one iteration only. In 2.3(b), the concept of multiplicity is shown. In this case, one token is used by multiple iterations. In Figure 2.3(c), the OOM- model is depicted. In Figure 2.3(d), the OOM+ model is shown. In this case, a reordering controller is necessary to solve the communication issues, as the tokens are not read in-order.

After deriving the communication models, the channel sizes for the FIFOs are calculated. The calculated buffer sizes should not be too small. If this is the case, one or more processes are blocked on write operations. On the other hand, if the buffer is too large, there is a penalty for the increase in memory and area usage. However, finding optimal deadlock-free buffer sizes is a non-trivial problem. In this process, tools such as PNGen [2] use greedy algorithms to compute adequate buffer sizes.

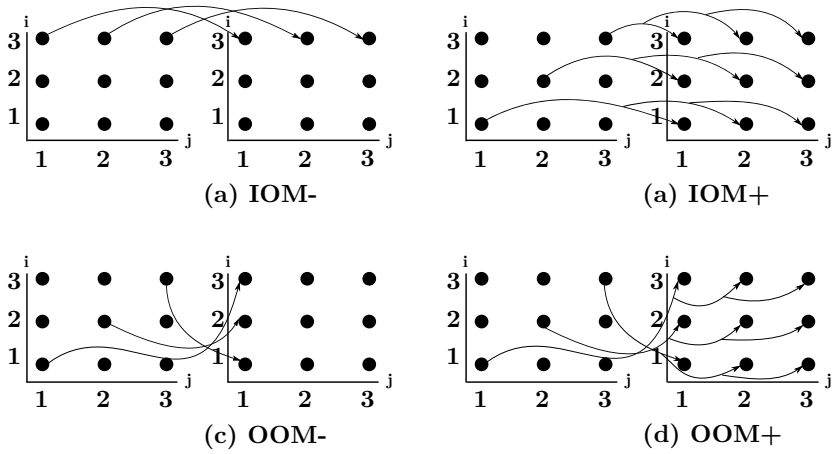


Figure 2.3: Communication models for Polyhedral Process Networks.

2.3 Static Affine Nested Loop Programs

SANLPs are programs with input restrictions, that can be automatically transformed into PPNs. SANLPs are used in a various domains, such as molecular biology, astronomy, and high performance computing. In particular, SANLPs are well suited to express time critical parts of audio/video streaming-based and DSP applications [9]. In the following, we discuss SANLPs in detail.

2.3.1 Overview

The SANLPs are specified in the C programming language. In this thesis, we use the following definition of SANLPs [2]:

Definition 2.2 (Static Affine Nested Loop Program)**.** A *static affine nested loop program* (SANLP) is a computer program that consists of statements, where:

- all statements are placed within one or more loops, and zero or more if-statements;
- all loop strides are constant;
- data exchange through hidden variables is forbidden;
- lower and upper loop bounds, array index expressions and if conditions are an affine function of non-dynamic, static program parameters and enclosing loop iterators.

An important property of SANLPs is that there are no restrictions for function calls, i.e., in the SANLP it is possible to call functions that do not conform to the definition given in 2.2.

2.3.2 Applied SANLPs

In Listing 2.1, an example of a SANLP written in the C programming language is shown.

```
1 for(i=0; i<N; i++)
2   for(j=0; j<N; j += 2)
3     if ( i+j <= N-1)
4       transform(&a[i][j], b[j][i]);
```

Listing 2.1: An example of a SANLP.

It is possible to place SANLP statements at any loop-level, as is shown in Listing 2.1. In line 3, an if-statement is used to guard the assignment statement. In this case, the logical condition is an affine combination of loop indices and constants. Then, in line 4 the `transform` function reads variable `b[j][i]` and writes to variable `a[i][j]`. The & operator is used to pass the address of the variable `a[i][j]`, making it possible for the `transform` function to write data to `a[i][j]`.

2.3.3 Iteration Domain and Dependencies

Each process in a polyhedral process network has a collections of elements representing its iteration domain. For example, consider the SANLP shown in Listing 2.2.

```
1 for(i=1; i<=M; i++)
2   for(j=2; j<=N; j++)
3       A[i][j] = F(A[i][j-1]);
```

Listing 2.2: Data dependencies in a SANLP.

To represent the iteration domain, the concept of parameterized polyhedrons [9] is introduced. Parameterized polyhedrons are used to represent loop nests that iterate over a set of finite iterations. These sets are geometrical representations of the SANLP. More details on the mathematics behind polyhedral process networks can be found in [2, 9]. For the derivation of the iteration domain, the information presented in this section is sufficient. Polyhedra can depend on a vector of parameters, denoted by $\mathbf{p}$, and we therefore define a parameterized polyhedron, $\mathcal{P}(\mathbf{p})$, as follows:

$$\mathcal{P}(\mathbf{p}) = \left\{ \mathbf{x} \in \mathbb{Q}^d \mid A\mathbf{x} \geq B\mathbf{p} + \mathbf{b} \right\}, \tag{2.1}$$

where A is an integral $m \times d$ matrix, B is an integral $m \times d$ matrix and b is an integral vector of size m [9]. It is possible to formally derive the iteration domain for the statement in line 3 of Listing 2.2, by applying Equation 2.1:

$$P(M,N) = \left\{ \{(i,j) \in \mathbb{Z}^2 | \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} * \begin{pmatrix} i \\ j \end{pmatrix} \geq \begin{bmatrix} 0 & 0 \\ -1 & 0 \\ 0 & 0 \\ 0 & -1 \end{bmatrix} * \begin{pmatrix} M \\ N \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \right\} \tag{2.2}$$

Before applying transformations to processes, it is important to know what data dependencies exist between successive calls of statement B in the inner loop. In this example, the assignment of A[i][j] is determined by the value of A[i][j-1]. The data dependencies for Listing 2.2 are shown in Figure 2.4. The arrows indicate the data dependencies between iterations.

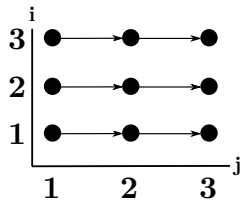


Figure 2.4: Dependency analysis of Listing 2.2.

To transform the program code in Listing 2.2, it is necessary to divide the iteration domain into points that are not dependent on each other. If there is no dependency between two points, it is possible to execute the statements of B in parallel.

2.3.4 Transformations

The dependencies shown in the previous section for the source code in Listing 2.2, allow for transformations to optimize the program code. In Figure 2.5, it is visible that operations are row dependent, i.e., there are no vertical dependencies between operations. In this example, the loop bounds M and N in Listing 2.2 are equal to 3.

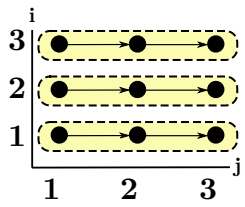


Figure 2.5: Selection of independent computational tasks.

```
1 for(j=2; j<=N; j++)
2   A[1][j] = F(A[1][j-1]);
3 for(j=2; j<=N; j++)
4   A[2][j] = F(A[2][j-1]);
5 for(j=2; j<=N; j++)
6   A[3][j] = F(A[3][j-1]);
```

Listing 2.3: Data dependencies in a SANLP

By applying loop unrolling (modulo unfolding), as shown in Listing 2.3, it is possible to map the statements in line 2, 5 and 8 to independent processes. Now, if a PPN of the C-code in Listing 2.3 were to be derived, such optimizations would be possible through algebraic manipulations [9]. This is one of the major strengths of PPNs. Examples of such algebraic manipulations are skewing, plane-cutting, and merging.

2.4 The LLVM/Clang Compiler Infrastructure

The LLVM Compiler Infrastructure is the result of a research project by the University of Illionois released in 2003. The goal of the project was to develop a modern compiler with support for the *Static Single Assignment (SSA)* form. SSA requires that each variable in the program code is defined before it is used, and that each variable is assigned only once [10]. If the compiler supports SSA, various compiler optimizations, e.g., dead code elimination and constant propagation, are easier to implement. In LLVM all scalar register values are represented in the SSA form. Furthermore, most production-grade compilers are not easily integrated in other applications. That is, given the aging code base and decade old techniques in such compilers, it is often impossible to reuse code in other programs. LLVM tries to overcome these issues by providing a framework that is extensible and reusable.

Traditional compilers are often monolithic, whereas LLVM is build as a set of modular components, as shown in Figure 2.6. In this work, a compiler front-end is required that supports C and C++ programs, as we need to instrument source code for profiling. A compiler front-end for LLVM supporting C, C++, Objective C and Objective C++ is Clang. Clang provides a complete set of libraries for source-to-source transformations, which can be integrated in customized applications.

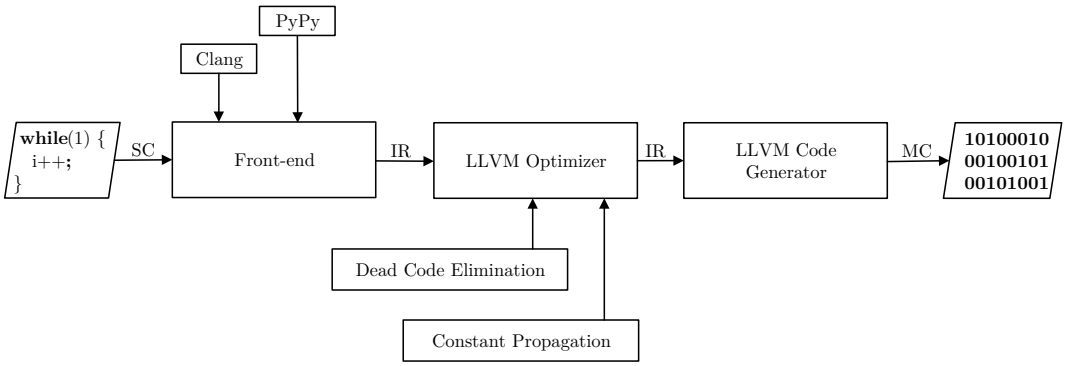


Figure 2.6: LLVM Compiler Infrastructure.

2.5 Hierarchical Program Analysis

In modern compilers, *Whole Program Analysis* (WPA) [11] is used for the analysis of large software systems. In WPA, a *Control Flow Graph* (CFG) is constructed for individual procedures, and a collection of algorithms is used to capture inter-procedural behavior of software. Other specializations of WPA are available as well. For example, Larus [12] introduced a technique called *Whole Program Paths* (WPP). It applies both static and dynamic analysis to capture a complete CFG in a compact form, using specialized compression algorithms.

In this work, inter-procedural behavior is analyzed, a CFG is not used for this purpose. That is, the profiler annotates code to detect relations between variables in software systems at run-time. In this way, it is possible to keep track of read and writes to variables in programs that have a hierarchical design. Therefore, it would be incorrect to define the approach used in this work as WPA. We use the term *Hierarhical Program Analysis* (HPA) to describe the inter-procedural relations. HPA is defined as follows:

Definition 2.3 (Hierarchical Program Analysis)**.** *Hierarchical Program Analysis* (HPA) is a technique for relating inter-procedural variables, where all variables within procedures are related using their full memory address.

2.6 High-Level Synthesis Tools

For polyhedral code generation, numerous tools are available. The *MPSoC Application Programming Studio* (MAPS) [3] is one example. It provides a framework for programming MPSoCs. MAPS supports the extraction of parallel specifications from sequential programs. Furthermore, it provides a specialized programming language called *C for Process Networks* (CPN), and the language is aimed at designers who specify specifications in block diagrams. Another well known polyhedral compiler is CLooG [13]. Daedalus [5] provides an open source framework for MPSoC programming. Daedalus provides means to transform high-level specifications in the C programming language to gate-level implementations.

In this work, Compaan *Design Development Environment* (DDE) [4] is used for the automatic conversion of C-based streaming algorithms to heterogeneous platforms. The Compaan-compiler uses *Leiden Architecture Research and Exploration Tool* (LAURA) as back-end for mapping PPNs onto hardware. Examples of hardware platforms supported by Compaan are *Field Programmable Gate Arrays* (FPGAs), and *Graphics Processing Units* (GPUs).

2.7 Summary and Conclusions

In this chapter, we presented the necessary background for the work presented in the next chapters. The concepts of PPNs were introduced, as well as SANLPs, which are automatically mapped onto PPNs. After that, we gave a brief overview of the compiler architecture used as framework for cprof. Finally, the tools used to transform high-level specifications to hardware implementations were introduced in the last section. We showed that deriving process networks requires a significant amount of effort. As a result, engineers and scientists will benefit from cprof, as it provides means to estimate behavior of SANLPs, without actually deriving the PPNs.

3 Related Work

In this chapter, we present the state-of-the-art in performance estimation of software and hardware designs. In Section 3.1, performance estimation using simulation is discussed. Then, in Section 3.2, follows a brief summary of analytical performance estimation. In Section 3.3, tools and methods for profiling are evaluated. The summary and conclusions are presented in Section 3.4.

3.1 Simulation

Simulation at the RTL level is the most precise method to estimate throughput performance. The accuracy is very high, as the RTL for implementation is also used for simulation. However, to obtain this level of accuracy, the complete design flow is traversed, as shown in Figure 3.1.

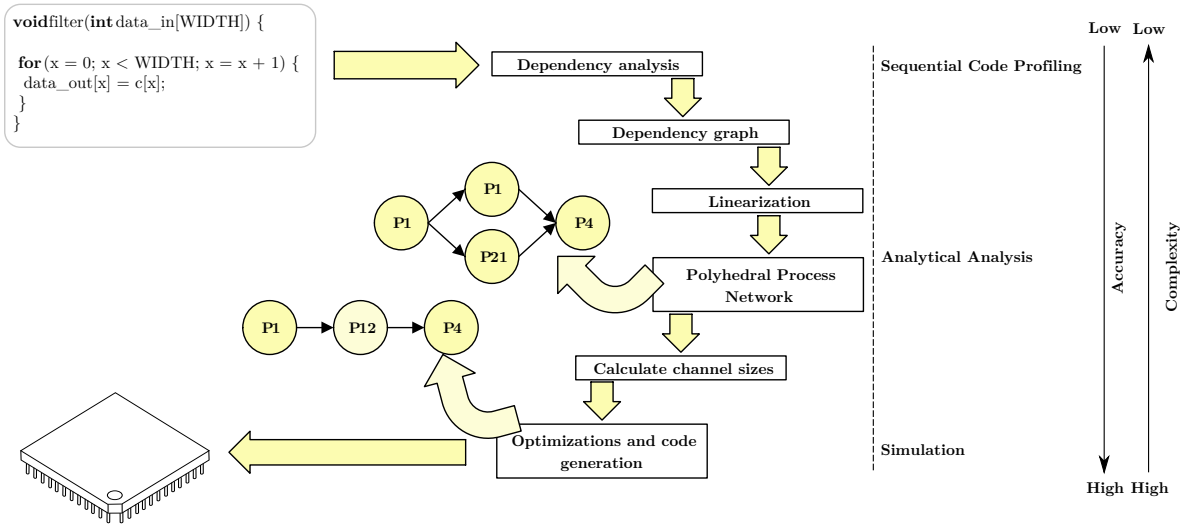


Figure 3.1: Level of accuracy and complexities in performance estimation.

The advantage of a low-level simulation is that designers know about actual implementation details, such as resource cost. However, this level of details is often not necessary to estimate the performance in early stages of designing systems. Cprof is not designed to replace simulation as a means of performance estimation. Instead, it is a tool which can help developers postpone gate-level simulation to the last stage of the design flow.

3.2 Analytical Estimation

Given the mathematical definition of PPNs, there have been efforts to analytically predict the performance. For example, in [9], the authors propose estimation techniques for process networks on microprocessor-based systems. However, the application of these techniques is limited, as only a subset of polyhedral process networks is supported.

One advantage of analytical estimation is that it is independent of workload, whereas simulation and profiling approaches are directly affected by the workload. Nonetheless, a complete derivation of

the process network is required for analytical estimation, which we want to avoid in the first place.

Haastregt [2] introduced four new concepts for the performance estimations of PPNs. With his concepts, it is possible to profile a wide-range of PPNs. However, to estimate the performance of the PPN, the derivation of the complete process network is necessary. Therefore, the proposed solutions are equally or more expensive than the methods introduced in [9].

3.3 Profiling

Profilers inspect the dynamic behavior of computer programs. This gives insight into the behavior of computer programs by measuring memory consumption, duration, time complexity, and other quantifiable metrics of a program during execution. To collect data, profilers employ a variety of techniques. Such techniques include, but are not limited to, static code analysis and code instrumentation. Modern profilers apply a combination of techniques to collect the performance profile.

3.3.1 General-Purpose Profilers

For the purpose of dynamic performance estimation, a collection of both free and commercially general-purpose profilers is available. A well-known profiler developed in the 1980's is gprof [14]. Gprof is used on daily basis by software engineers to collect performance metrics of computer programs. However, profiling for hardware platforms is not the same as profiling for general-purpose processors, as the system architectures vary greatly. That is, decisions based on general software profilers, like gprof, may give directions that are not applicable to hardware implementations. For example, let us take the time required to finish a computer program. The profiler gprof delivers a number that is imprecise, as it relies on program counter sampling. The sampling rate affects the accuracy of performance measurements, making it an inexact method for performance evaluation. The profiling results are only valid for the platform on which the profiling is performed. As a result, tools such as gprof are not a reliable instrument for estimating performance of PPNs.

3.3.2 Hardware Profilers

Hardware profilers support different processor architectures, and are designed to estimate the performance of programs executing on different processors. The Valgrind profiler [15] provides a framework for dynamic instrumentation. Valgrind provides advanced tools for implementing support for different hardware architectures.

For run-time analysis of code, Valgrind implements *Dynamic Binary Analysis* (DBA). Valgrind's implementation of DBA is based on *Dynamic Binary Instrumentation* (DBI). DBI injects code into the computer program during run-time. Developers do not have to recompile or relink their code to collect measurements. However, Valgrind is strongly intertwined with the architecture of the computing platform. Implementing support for polyhedral process networks and the LAURA architecture in Valgrind requires a significant amount of work.

The profiler TotalProf [16] was designed to support multiple processor architectures. TotalProf works with the LLVM IR code for profiling. The LLVM intermediate representation is instrumented by TotalProf, and executable code is generated for the host processor. The framework supports profiling for MPSoCs. However, the LAURA architecture is not supported, and TotalProf is, therefore, not suitable for profiling PPNs.

3.3.3 Parallel and Memory Profilers

In [17], a tool for finding pipeline parallelism in sequential programs is proposed. It applies static and dynamic data flow analysis to find patterns of parallelism in applications. The tool does not support PPNs, and is targeting general-purpose computers.

ParaProf [18] is a tool for parallel performance analysis. It supports various programming languages, including C and C++. However, the tool only supports programs already specified in parallel frameworks, such as OpenMP, MPI or pthreads.

Harmony [19] is a tool for finding *Parallel Block Vectors* (PLB) in computer programs. A PLB describes the relation between the static blocks in a multi-threaded application, and the degree of parallelism available during each invocation of a block. However, Harmony only supports pthread applications, which is not applicable to this project.

Intel's Parallel Advisor [20] is a profiler that identifies regions of program code that may benefit from parallelism. However, the parallel advisor requires a significant amount of effort from the programmer. This happens as fine-grained annotations need to be manually inserted into the program code. Furthermore, its application domain is shared-memory systems, based on a control-flow architecture, making it unfit for PPNs.

The Q^2 profiling framework has its roots in the *Delft Workbench* (DWB) [21]. DWB targets heterogeneous platforms with support for reconfigurable computing. It describes the complete design flow from high-level specification to synthesis. The Q^2 framework profiles programs in two steps. The first step is static profiling, to collect code characteristics from the computer program. The characteristics are used to make estimates of FPGA area requirements. The second step is dynamic profiling, to collect data about the run-time behavior of the computer program.

The next step is the execution of the general-purpose profiler gprof, to produce the call graph of the application. The call graph is used by the profiler *Memory Access Intensity Profiler* (MAIP) for accurate measurements, as gprof is sample based and, therefore, inaccurate. The data produced by MAIP is then used by a program to estimate parallelism in the profiled data.

The static analysis part is modeled by the Quipu [22] approach. Quipu gives estimates on the area usage of a C code application implemented in hardware. Early Quipu predictions have an error of 10% to 20% percent. However, it gives, in an early stage of the design process, insight into area performance of a given implementation.

The memory access behavior of the application is fully described by QUAD [23]. This tool is implemented with the Pin framework [24]. The profiling tools developed for the Q^2 provides developers with valuable information for HW/SW co-design. However, the Q^2 profiling framework, and its related tools and architectures target a general shared-memory HW/SW co-design model, whereas our scope is limited to PPNs.

In [25], the author proposes a method for memory access and operator usage estimation. Memory usage is a key factor in application performance. With this estimation method, it is possible to view memory access and operator usage on function or loop basis. With this information, developers can optimize code, and configure HLS tools to increase circuit throughput. However, the tool is targeted towards traditional control-flow architectures, and developers have to hand-craft optimizations to explore the design space.

3.3.4 Critical Path Analysis

Critical Path Analysis (CPA) is a technique used for identifying parallel code regions in a program [26]. CPA is used to model the synchronization and communication dependencies between processes in a program. Kumar proposed the tool COMET [27], for measuring the degree of parallelism found in Fortran programs by applying CPA. The assumption is that the program is executed on an ideal machine. This ideal machine has an unbounded number of resources. Furthermore, the ideal machine has no synchronization, communication and scheduling issues. After instrumenting the source code, COMET dynamically collects data to determine the absolute amount of parallelism.

Kremlin [28, 29] is profiling tool similar to gprof, but is designed to discover parallelization in sequential programs. Its purpose is to identify which parts of a program to parallelize. Kremlin uses a technique called *Hierarchical Critical Path Analysis* (HCPA), which adds hierarchy to critical path analysis. With CPA, parallelism is measured within the complete program, whereas HCPA considers separate program regions. Kremlin delivers the upper bound on parallelism in the program

under inspection. With Kremlin's parallelization planner, it is possible to calculate speedup after parallelizing a region.

However, there are certain limitations to Kremlin's approach. It is a tool targeting general-purpose processors, whereas cprof targets implementations based on PPNs. Moreover, it is not possible to automatically transform the code in such a way that the theoretical maximum degree of parallelism can be achieved.

Another tool applying HCPA is Parkour [30]. This tool provides parallel speedup estimates for unmodified serial programs. Parkour consists of two phases: HCPA and speedup prediction. HCPA is covered by the Kremlin profiler. Kismet [31] extends Parkour and Kremlin with sequence regions. With this extension, Kismet is able to detect various forms of parallelism within the program code. In particular, it is possible to separate *instruction-level parallelism* (ILP) from other classes of parallelism.

Li et al. [32] present DiscoPoP, short for *Discovery of Potential Parallelism*. The profiler is based on the Kremlin profiler. DiscoPoP is set out to find possible parallel regions within unstructured code, whereas Kremlin is designed to work on regions between two specified endpoints. However, this additional feature still makes it an unfit choice for profiling SANLPs.

Haastregt [2] transformed Kumar's paper into an idea called cprof. His approach for profiling PPN performance is based on the COMET profiler. With this idea, it is possible to measure parallelism in PPNs. First, cprof can measure the parallel performance of the program on an ideal machine. In this ideal machine, each iteration of a statement is mapped onto its own processing element. Second, cprof can evaluate the parallel performance of the program, while restricting all iterations of a statement to one process.

The original cprof profiling tool developed by Haastregt [2], has no support for complex dynamic data structures, and all data is kept on the stack, thereby limiting the number of processes that can be profiled. Moreover, it has no support for selecting kernels of interest. The original cprof is a basic proof-of-concept, not capable of profiling real-world applications. Another issue is that there is no consideration of hierarchy. Without hierarchy, it is impossible to estimate performance of systems with inter-procedural behavior. In the original idea, the absolute throughput and maximum degree of parallelism are collected and presented. However, it is not possible to automatically apply transformations based on these numbers to explore the design space. In this thesis, we continue with the work of Haastregt presented in [2].

3.4 Summary and Conclusions

In this chapter, various profilers and other tools for static and dynamic performance estimation were discussed. None of the profilers, but the original cprof support the PPN architecture. In the following chapter, we continue with the solution approach we adopted in order to estimate the performance of C code when implemented as a PPN in hardware.

4 Solution Approach

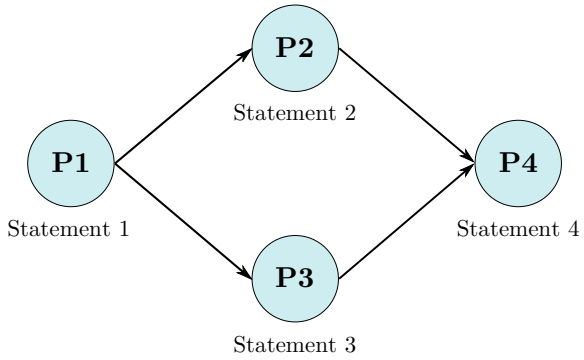
In this chapter, we present cprof. Cprof profiles sequential C programming code, and determines the performance of the program when implemented as a PPN in hardware. In Section 4.1, the concepts related to profiling PPNs are introduced. The definition and examples of performance estimates are given in Section 4.2. In Section 4.4 and 4.5, the solutions for modeling the behavior of processes are discussed. The definitions of the execution profiles are given in Section 4.6 and 4.7. Finally, in Section 4.9, a summary is given and conclusions are drawn.

4.1 Concepts

The goal of cprof is to determine the performance of applications specified in the C programming language, that are implemented as a PPN in hardware. The performance we determine in cprof relates to the run-time of a given application in hardware, measured on a global time scale. The global time scale is used to track start and stop times of statements in sequential computer programs. In this section, we introduce the relevant concepts and properties related to the profiling of polyhedral process networks.

```
1  for(i=0; i<4; i++) {
2  st1: a[i] = source();
3  }
4
5  for(i=0; i<4; i++) {
6  st2: x[i] = foo(a[i]);
7  st3: y[i] = bar(a[i]);
8  }
9
10 for(i=0; i<4; i++) {
11 st4: sink(x[i], y[i]);
12 }
```

(a) SANLP.



(b) Derived PPN.

Figure 4.1: Mapping of a C program to a Polyhedral Process Network.

In the whole discussion, we follow the basic principle used by Compaan and Daedalus to model PPNs. The principle is that each statement is mapped to one process. In Figure 4.1(a) and 4.1(b), we show a C-program and the resulting PPN, respectively. The figure shows that the functions `source`, `foo`, `bar` and `sink` are mapped to the processes P1, P2, P3 and P4, respectively. The relationships between variables are mapped to edges. For example, the variable `a[i]` from `source` to `foo` becomes an edge in Figure 4.1(b).

From the work of van Gemund [33], we know that the performance of any parallel system is determined by four properties [33]: **basic calibration**, **conditional synchronization**, **conditional control flow**, and **mutual exclusion**. To understand these 4 properties, we look at each of them in context of the program given in Figure 4.1.

4.1.1 Basic Calibration

The functions in the C-programs we analyze will be implemented in hardware. It turns out that these functions can be modeled using only two parameters: the initiation interval and the function latency. Look for example at the functions `foo` and `bar` in Figure 4.1(a). The two parameters are enough to calibrate a function in C-code for performance analysis. If we look at a modern HLS tool, such as Vivado HLS, we see that Vivado HLS also characterizes a function using the initiation interval and the function latency. For example, in line 3 of Listing 4.1, Vivado HLS determines that the initiation interval and function latency of `source` are 1 and 5, respectively.

```
1 [exec][SCHED-11] Starting scheduling ...
2 [exec][SCHED-61] Pipelining function 'source'.
3 [exec][SCHED-61] Pipelining result: Target II: 1,Final II: 1,Depth: 5.
4 [exec][SCHED-11] Finished scheduling.
```

Listing 4.1: Example output of Vivado HLS.

No matter what the complexity is of the IP block that implements a computational function, the initiation interval and function latency are enough to model the performance of any function, as shown in Figure 4.2.

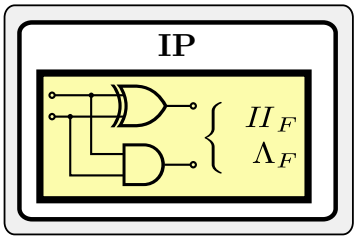


Figure 4.2: Example IP block.

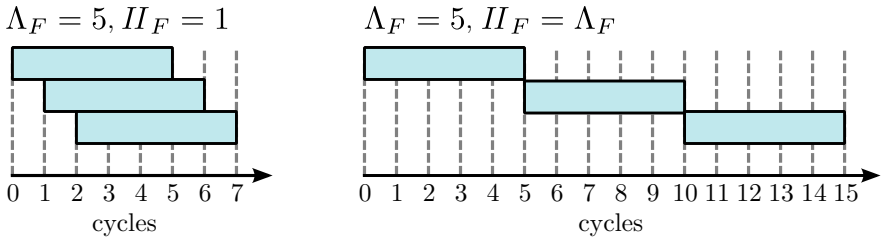
The initiation interval determines the throughput. It indicates the number of cycles required between successive starts. The function latency determines the number of cycles it takes to finish the execution of an IP block:

- the initiation interval II_F,
- a function latency Λ_F,

where $II_F \in \mathbb{N}^+$ is the initiation interval in clock cycles, and $\Lambda_F \in \mathbb{N}^+$ is the input-to-output latency in clock cycles [2]. The initiation interval and function latency are explained using the `source` function in line 2 of Figure 4.1(a). We implemented the `source` function in Vivado HLS, and this gave 5 cycles for the function latency and an initiation rate of 1 cycle. In this case, the initiation interval $II_F < \Lambda_F$. The result is that the execution is pipelined, and throughput is increased, as shown in Figure 4.3(a). On the other hand, in 4.3(b), $II_F = \Lambda_F$. In this case, pipelining is not possible.

The integration of IP blocks in PPNs derived by Daedalus and Compaan uses the LAURA Virtual Processor model [2]. In Compaan and Daedalus, each process consists of a read, execute, and write stage, as shown in Figure 4.4. The read and write step take care of the distribution of data in a process, as a result of the parallelization of the C-code by the data-flow analysis. The read or write stage is optional, depending on the requirements of the execute stage. For example, the statement in Line 2 of 4.1(a) skips the read stage, as the `source()` function is only producing, and not consuming, data. The execute stage integrates the IP block representing a function in the C-code.

The use of the LAURA processor leads to the introduction of two properties relevant to the implementation of a process into hardware.



(a) Pipelined execution. **(b) Non-pipelined execution.**

Figure 4.3: The initiation interval and function latency of the execute stage.

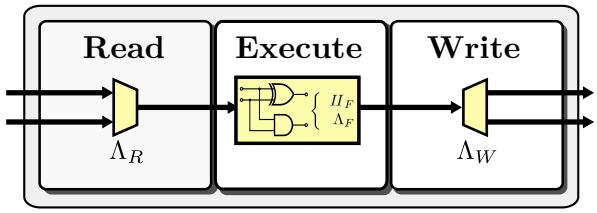


Figure 4.4: The Read, Write, and Execute units in a polyhedral process.

- the read latency Λ_R,
- the write latency Λ_W,

where $\Lambda_R \in \mathbb{N}^+$ is the latency in clock cycles for reading input tokens, and $\Lambda_W \in \mathbb{N}^+$ is the latency in clock cycles for writing output tokens. In this thesis, we assume $\Lambda_R = \Lambda_W = 1$, as all input and output tokens are read or written in one cycle [2]. The execute stage typically implements a pipelined IP core, as shown in Figure 4.5. The read operation is blocked until data becomes available, implementing the behavior of PPNs. As a result, the pipeline is stalled in iteration 2, as indicated by the "–". The execution is delayed, and it is resumed later on, as depicted in Figure 4.5.

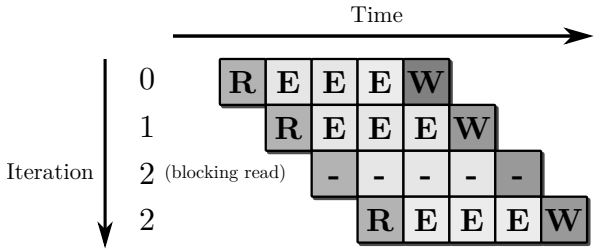


Figure 4.5: Pipelined execution of the read, execute and write stages.

4.1.2 Conditional Synchronization

Within a PPN, we assume that a function only executes when all its data arguments are present. The same happens in sequential C-code. Consider the function `sink` in line 11 of Figure 4.1(b). The function consumes the variables `x[i]` and `y[i]`. The execution of the function `sink` is allowed only if both data dependencies are satisfied.

Now, assume that variable `x[0]` is available at time $t_{x[0]} = 3$, and variable `y[0]` is available at time $t_{y[0]} = 4$, in the first iteration of the loop in line 10 of Figure 4.1(a). Even though `x[0]` was available at $t = 3$, the function has to wait until `y[0]` becomes available at $t = 4$. This affects the performance of a system, and is called conditional synchronization.

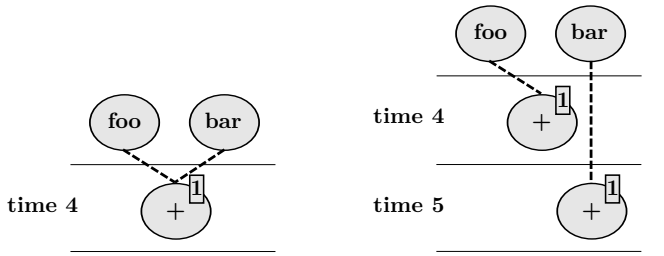
4.1.3 Conditional Control Flow

Statements, like `a[i] < 3` and `i > N`, are used for modeling conditional control flows. The sequence in a regular program is influenced by conditional control flow. However, cprof assumes that the code is specified as a SANLP (see Section 2.3). The loop bounds and conditional predicates of a SANLP are affine functions of enclosing loop indices and parameters. As a result, a SANLP has only static control parts. There is always a single-entry, single-exit region.

4.1.4 Mutual Exclusion

Mutual exclusion is about the fact that two or more tasks are not allowed to execute their critical sections simultaneously. The critical section of a task accesses a shared resource, which can be used by one task at a time. To illustrate the concept of mutual exclusion, let us consider the functions `foo` and `bar` in Figure 4.1(a). Both functions are mapped to the processes `P2` and `P3`, as shown in Figure 4.1(b). The input `a[i]` is available to both `P2` and `P3` at the same time. The functions `foo` and `bar` are, therefore, able to start their execution at the same time.

However, assume that the system has only one adder and `foo` and `bar` require, each one, one adder to execute. As a result, the simultaneous execution of `foo` and `bar` results in a resource conflict, as shown in Figure 4.6(a). High level synthesis tools, such as Vivado and OpenCL, will try to come up with a feasible schedule, in order to resolve the resource conflict. Well-known techniques for solving resource conflicts are retiming [34], and changing the performance parameters of IP blocks, for example using an initiation interval of 2. A possible conflict-free schedule is shown in Figure 4.6(b). In this example, the execution of `bar` is delayed.



(a) Resource conflict. **(b)** Resource conflict resolved.

Figure 4.6: Mutual Exclusion in Processes.

However, the fine-grained details necessary to solve resource conflicts is not available to cprof. The reason is that cprof has no knowledge about the hardware resources used by an IP block. That is, functions are not restricted by the definition of SANLPs (see Section 2.3), and can take any form as long as the implementation of the function is supported by the high-level synthesis tool. We assume a one-to-one mapping of functions to processes. As a result, there cannot be any resource contention.

4.2 Performance Estimation

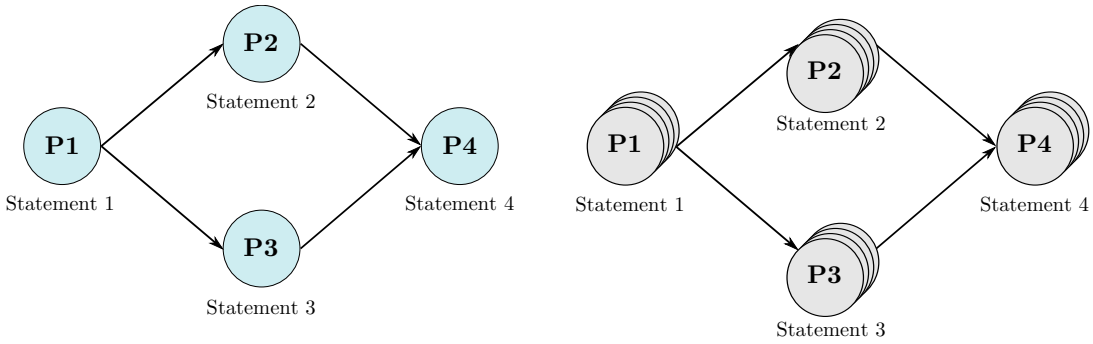
In this section, we define two modes of performance estimation that both relate to the run-time of PPNs in hardware: the absolute throughput estimation and the unbounded throughput estimation.

4.2.1 Absolute Throughput Estimation

The absolute throughput assumes that all iterations of a statement are mapped onto the same processing resource. We define absolute throughput as follows.

Definition 4.1 (Absolute Throughput)**.** Absolute throughput assumes that all iterations of a statement are mapped onto the same processing resource, and assumes an unbounded number of hardware resources.

For example, let us consider process P1, representing statement 1, in Figure 4.7(a). In absolute throughput estimation, the 4 iterations of statement 1 are mapped onto the process P1. If we assume that all executions of a statement are mapped onto the same processing resource, then the next invocation of a process should take the initiation interval (II_F) into account. The PPN of the source code in Figure 4.1(a) is shown in Figure 4.7(a).



(a) Absolute throughput. **(b)** Unbounded throughput.

Figure 4.7: Absolute and unbounded throughput.

4.2.2 Unbounded Throughput Estimation

The unbounded throughput assumes an unbounded number of processing resources. That is, each execution of an iteration of a statement is mapped onto its own dedicated processing resource. We define unbounded throughput as follows.

Definition 4.2 (Unbounded Throughput)**.** Unbounded throughput assumes that the execution of an iteration of a statement is mapped onto its own dedicated processing resource, and assumes an unbounded number of hardware resources.

For example, statement 1 in Figure 4.1(a) is mapped to four processing resources, instead of one. As a result, each process created for statement 1 executes as soon as the input data is available. The PPN used for unbounded throughput estimation is shown in Figure 4.7(b).

4.3 Case Studies

To illustrate how to determine the absolute throughput and the unbounded throughput estimation, we will now look again at the program shown in Figure 4.1(a).

4.3.1 Case Study: Absolute Throughput

To successfully collect the performance profiles of computer programs without implementing them as a PPN in hardware, data dependencies related to the execution of processes in a PPN must be tracked to model **conditional synchronization**. For this purpose, we introduce **shadow variables** [2]. For each variable declaration, a shadow variable is used to store the finish time of write operations, as is shown in Figure 4.8(a). For process P1, $II_F = 2$ and $\Lambda_F = 1$. Process P2 and P3 have an initiation interval and function latency of $II_F = \Lambda_F = 1$. Process P3 is configured with $II_F = 1$ and $\Lambda_F = 2$.

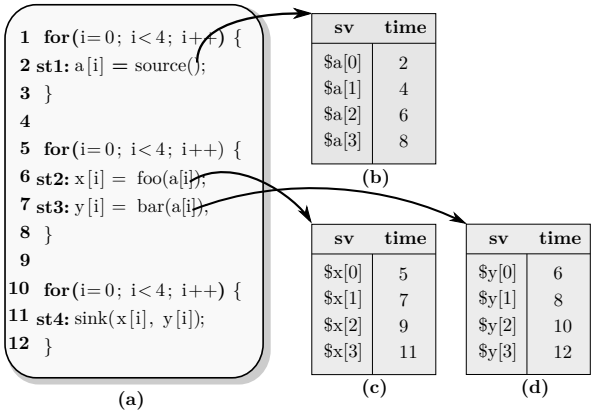


Figure 4.8: The shadow variables and their values for absolute throughput estimates.

In Figure 4.8(b), the left column shows the name of the shadow variable, `$a`. The shadow variable `$a` is associated with the original variable `a`. The second column lists the timestamp at which time the data produced for `$a[i]` is available. The same applies to shadow variables `$x` and `$y`. In this example, statement 4 in line 11 has no shadow variable, as no data is written by the process.

The measurements for both absolute and unbounded throughput make use of **control variables** [2] to calculate the valid start times of the processes. Control variables are associated with each statement to determine whether a process can execute, as shown in Figure 4.9. The start time of a statement is determined by taking the maximum over the input variables and the control variable. In the case of absolute throughput estimation, the initiation interval is added to the control variable, after successfully executing a process.

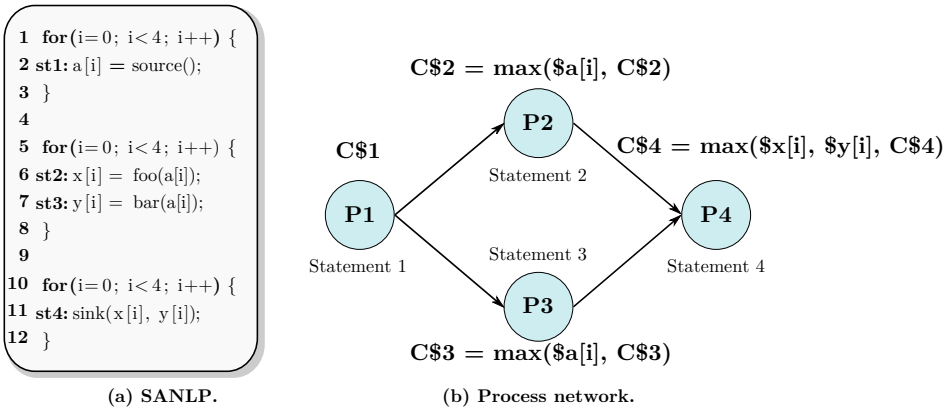
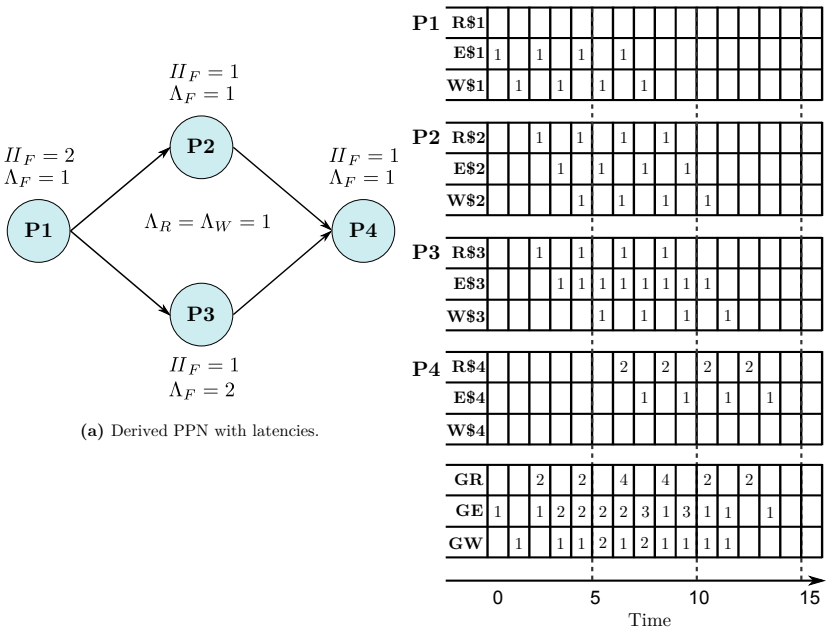


Figure 4.9: Control variables associated with each process in a polyhedral process network.

Shadow and control variables are used to facilitate the construction of execution profiles. What follows is an explanation of how this information is used for the absolute throughput estimate. In Figure 4.10(a), the polyhedral process network including the basic calibration is shown. The associated source code is shown in 4.8(a). The read and write latencies are defined as $\Lambda_R = \Lambda_W = 1$.

The statement execution profiles capture the behavior of the processes. In Table P1, in Figure 4.10(b), we can see that the first execute operation starts at time 0. The associated write operations start at time 1, as the function latency is one cycle. This information directly relates to the tables shown in 4.8(b). In table 4.8(b), the shadow variable `$a[0]` is available at time 2. This is correct, as at time 1 the variable is being written. Now, the next execute operation starts at time 2, as the initiation interval is 2 cycles. For process P2, P3 and P4 similar tables are constructed. In the



(a) Derived PPN with latencies.

(b) Statement execution profile.

Figure 4.10: Absolute throughput statement profiles.

execution statement profile of P4 in Table 4.10, the last execute operation is at time 14.

The global execution profile sums up the number of reads, executes, and writes active at a certain point in time, found in the statement execution profiles. As a result, the global execution profile describes the complete behavior of the process network. The global execution profile is subsequently used to determine the average and maximum degree of parallelism. We define the average and maximum degree of parallelism as follows.

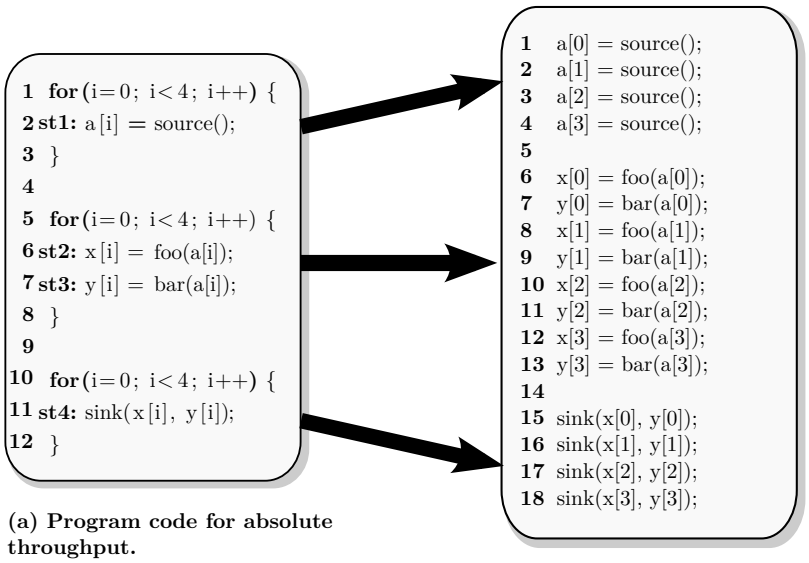
Definition 4.3 (Average Degree of Parallelism)**.** The average degree of parallelism is the sum of all execute operations in the global execution profile, divided by the number of execute operations in the global execution profile.

Definition 4.4 (Maximum Degree of Parallelism)**.** The maximum degree of parallelism is the maximum number of simultaneously active execute operations in the global execution profile, at a given time t.

The time at which the process network is finished executing is available as well. In this example, the process network is finished at time 14. In this case, the sum of all execute operations equals 20. The average degree of parallelism is, therefore, $\frac{20}{14} = 1.4$. This means that on average 1.4 processes are active. The maximum degree of parallelism is the number representing the peak parallel performance. In this example, the peak parallel performance is seen at time 8 and 10 in the global execution profile in Figure 4.8(b). The peak parallel performance of this PPN is 3.

4.3.2 Case Study: Unbounded Throughput

For unbounded throughput, we assume the existence of an unbounded number of processing elements. We can emulate this by fully unrolling the loops. In Figure 4.11(a), the original source code is shown, whereas in Figure 4.11(b) the fully unrolled version is shown. Fully unrolled means that each statement is mapped onto its own process resource, e.g. `foo(a[0])` is a different resource than `foo(a[1])` and `foo(a[2])`. The fully unrolled process network now consists of 16 processes instead of 4, as shown in Figure 4.14(b).

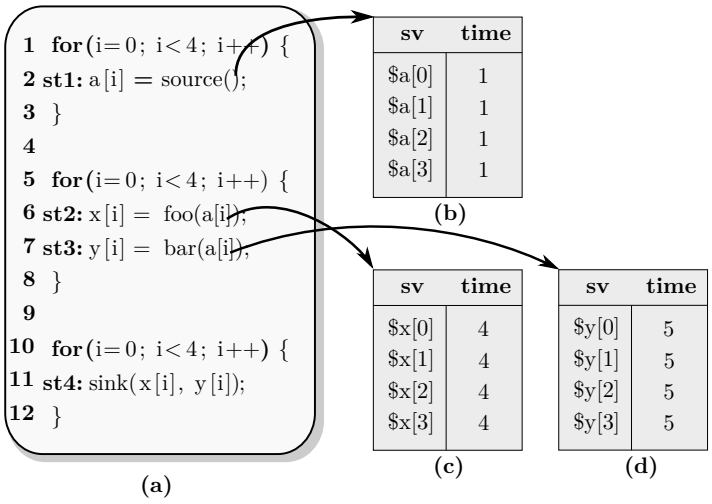


(a) Program code for absolute throughput.

(b) Possible program code for unbounded throughput.

Figure 4.11: Example C Program.

In the fully unrolled version, a statement starts executing as soon as the input data is available. In this example, the latencies and initiation intervals are the same, as in the case study of the absolute throughput estimation. In Figure 4.12(b), the times at which the variable `a[i]` is finished writing are listed. The same table is shown for statement 2 and 3 in Figure 4.12(b) and (c), respectively.



(a) **(b)** **(c)** **(d)**

Figure 4.12: The shadow variables and their values for unbounded throughput estimates.

The start time of a process in unbounded throughput mode is determined by the maximum over the input variables. A process starts executing as soon as the input data is available, as shown in Figure 4.13. For absolute throughput, the control variable includes the initiation interval of the process. However, for the unbounded throughput, this is not necessary. Each statement can immediately process the data as each execution is mapped onto its own processing resource.

For the unbounded throughput estimate, the associated statement profiles are shown in Figure

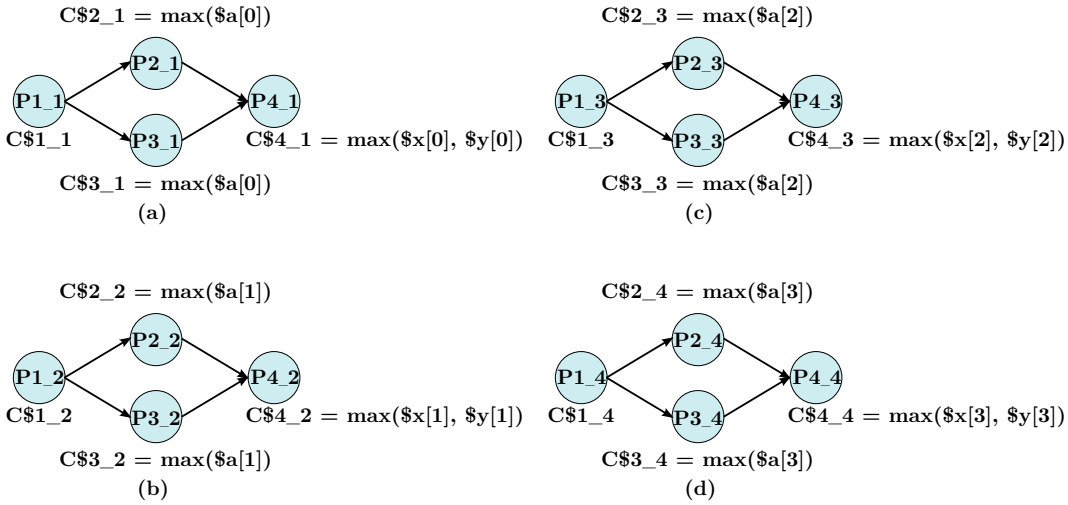
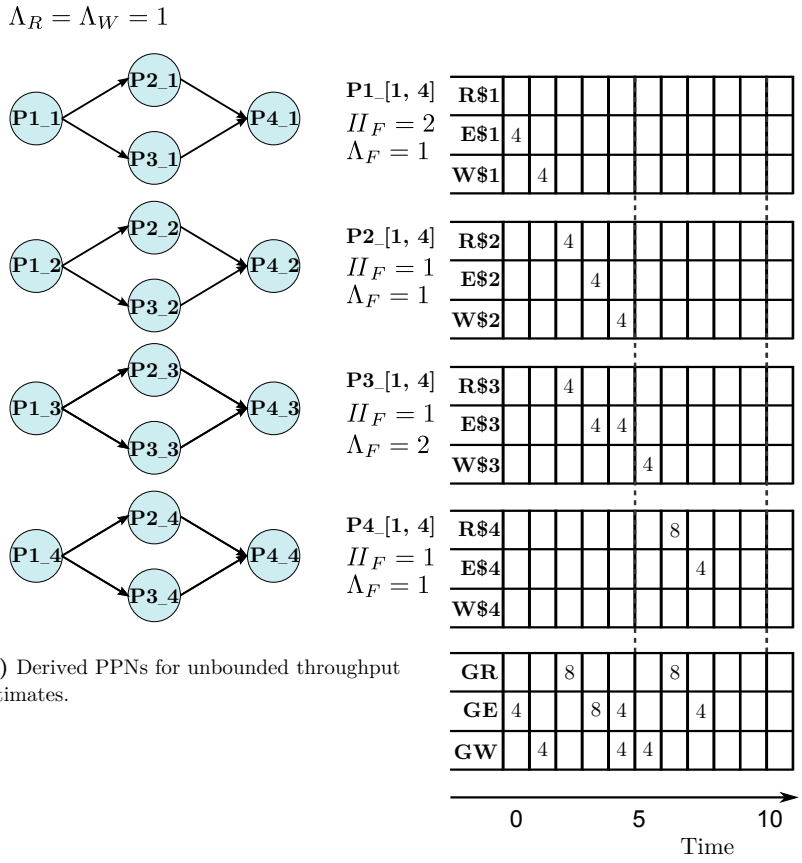


Figure 4.13: Determiniation of control variables for unbounded throughput.

4.14(b). At time 0, the process P1_1, P1_2, P1_3 and P1_4 are executing. Then, at time 2, the processes [P2_1, P2_4] and [P3_1, P3_4] read input data. Their start is delayed, as processes [P1_1, P1_4] write data at time 1. Then, processes [P4_1, P4_4] start executing at time 6, as the input dependencies are solved at that time.

The full execution of the process network takes, in total, 8 time units. The sum of all execute operations is shown in the table of the global execution profile. The average degree of parallelism equals $\frac{20}{8} = 2.5$. The maximum degree of parallelism is 8. To achieve the minimum execution time, a system with 8 processors is required. However, the average degree of parallelism provides a more realistic design point that is close to the maximum degree of parallelism [35].



(a) Derived PPNs for unbounded throughput estimates.

(b) Statement execution profile.

Figure 4.14: Unbounded throughput statement profiles.

4.4 Shadow Variables

To keep track of when a variable is written, we introduced the concept of shadow variables. The shadow variables are used for keeping track of the point in time in which a variable `v` is written. The timestamp written to the shadow variable `$v` is the time of the write operations, including the cost of a write operation Λ_W. Shadow variables are used to model the conditional synchronization in PPNs. Cprof incorporates the conditional synchronization aspect by performing a `max` on the timestamps stored in the shadow variables.

The SANLPs, which are by definition sequential, are processed in textual order, and one statement execution at a time. Moreover, before executing a statement, all data dependencies must be resolved. As long as the data dependencies are satisfied, many different execution orders may exist [9]. PPNs use this property to execute processes in parallel.

4.5 Control Variables

Control variables, denoted as `C$s`, are used to store the timestamp of the point in time in which a given process can execute. These control variables are constructed for each statement. The variable is used to make sure that the conditional synchronization aspect is taken into account, as well as the initiation interval (II_F).

This control variable `C$s` is based upon the availability of input data and is updated in the read

stage of a process. That is, for all variables associated, it takes the maximum timestamp at which one of the variables is written. If the absolute throughput is measured, the initiation interval (II) of the function is taken into account. As a result, it is possible to model the system for execution on a single resource.

For execution on an ideal machine, we assume an unbounded number of processing resources. Given an unbounded number of processing elements, there is no need to determine whether a resource is available. In effect, the control variable only considers the availability of data, and it takes the maximum over all shadow variables only.

4.6 Statement Execution Profile

To be able to calculate meaningful performance metrics, each statement `s` has three one-dimensional arrays `R$s`, `E$s`, and `W$s`. These three arrays make up the statement execution profile, which contains the behavior of the read, execute and write stages of a statement over time. For example, if `R$s[16] = 1`, it means that at time 16 there is one read operation active. Each array is initialized with zeros, and is incremented at each operation in an interval $[t_s, t_f)$, where t_s is the starting time and $t_f = t_s + \Lambda$ is the finish time of the operation with latency Λ.

The statement execution profile provides the following information.

- The total time spent on read, execute, and write operations, which is the sum of all elements in the `R$s`, `E$s` and `W$s` arrays.
- The start time of a process $s(p)$, which is the first non-zero element in `R$s`. If there are no read arguments, the first non-zero element in `E$s` is used.
- The finish time of a process $f(p)$, which is the last non-zero element in `W$s`. If no data is written, the first non-zero element in `E$s` is used.
- The number of unused read, execute, or write slots, which are found by looking for zeros in the `R$s`, `E$s`, and `W$s` arrays for the interval $[s(p), f(p))$.
- The maximum number of concurrent executed processes, which is obtained by finding the maximum value in `E$s`.
- The *flat execution profile*, which is defined as the number of process iterations executed simultaneously at a given time t. This profile is calculated using equation 4.1:

$$\texttt{R\$s}[t] + \texttt{E\$s}[t] + \texttt{W\$s}[t]. \tag{4.1}$$

4.7 Global Execution Profile

By using the statement execution profile, we can determine the global execution profile `G$s`, as defined in Equation 4.2:

$$\texttt{G\$}[\texttt{k}] = \sum_{i=0}^{|P|-1} \texttt{R\$}i[\texttt{k}] + \texttt{E\$}i[\texttt{k}] + \texttt{W\$}i[\texttt{k}], \tag{4.2}$$

$$0 \leq \texttt{k} < \max\{\forall p \in P \mid f(p)\},$$

where P is the set of all processes and k is the maximum index of a statement profile for a given process p. This definition is taken from [2].

Besides finding the finish time of a PPN, we are interested in the performance of an IP block in the execute stage of a process. The execute unit determines the efficiency of an IP block. That is, the

read and write operations are fundamental requirements of the computational part of the process. To capture this information, we define the following equation:

$$\texttt{GE\$[k]} = \sum_{i=0}^{|P|-1} \texttt{E\$}i\texttt{[k]} \tag{4.3}$$
$$0 \leq \texttt{k} < \max\left\{\forall p \in P \mid f(p)\right\},$$

where P is the set of all processes and k is the maximum index of a statement profile for a given process p. The following information is extracted from `GE$s`:

- The *execution time* of an IP block, which is equal to the number of elements in `GE$s`.
- The *average degree of parallelism*, as defined in Definition 4.3.
- The *maximum degree of parallelism*, as defined in Definition 4.4

4.8 Flow Dependencies

In sequential programs, the following data dependencies exist [36]:

- *Read After Write* (RAW): this may happen if a variable `v` is being read before it is written.
- *Write After Read* (WAR): this happens if a variable `v` is written after a variable is read and, thus, poses a problem in concurrent systems.
- *Write After Write* (WAW): this may happen if two statements write data to the same variable `v`. The timestamp of the first write operation is stored and is then overwritten by the timestamp of the second operation.

The flow dependency (RAW) is addressed in cprof as follows. The read operation takes the availability of a variable into account, by checking the shadow variable for the last write time. Therefore, a read cannot start before the write operation has finished and, as a result, this dependency is accurately modeled. Anti-dependencies in SANLPs do not affect the performance of PPNs, as each token is stored and accessed in and from a private storage location [2]. In effect, different values for the same variable are stored in dedicated storage areas. One of the properties of a PPN is that the storage location of a data token is never rewritten after utilization. Output dependencies cannot exist in PPNs, as the models guarantees that each produced token is consumed at least once [2]. This means that if a token is not used, it is never communicated. The result is that it is impossible to have output dependencies in PPNs.

The anti-dependencies (WAR) and the output dependencies (WAW) are not modeled in COMET [27], and in Haastregt's cprof [2]. As a result, it is impossible to model the performance of a computer program if it contains anti- (WAR) or output (WAW) dependencies. In this work, we extended (see Appendix E) cprof to support WAR and WAW hazards, broadening the application domain of the profiler.

4.9 Summary and Conclusions

In this chapter, we have shown that it is possible to estimate the performance of applications specified in the C programming language, when implemented as a PPN in hardware, using the four key aspects of performance modeling of parallel systems. Two modes of performance estimation were presented. The absolute throughput estimate assumes that all iterations of a statement are mapped onto the same processing resource, whereas the unbounded throughput estimate assumes that the execution of an iteration of a statement is mapped onto its own dedicated processing resource.

For both the absolute and unbounded throughput, cprof estimates the run-time of a polyhedral process network when implemented in hardware, and determines the average and maximum degree of parallelism available in PPNs.

5 Design and Implementation

In this chapter, we present the design and implementation of cprof, based on the concepts and solutions discussed in the previous chapters. The architectural overview of the profiler is presented in Section 5.1. The parts of cprof are shown in Figure 5.1. In the following sections, each part of the profiler is discussed in detail. The cost of profiling is presented in Section 5.8. Finally, in Section 5.9, a summary is presented and conclusions are drawn.

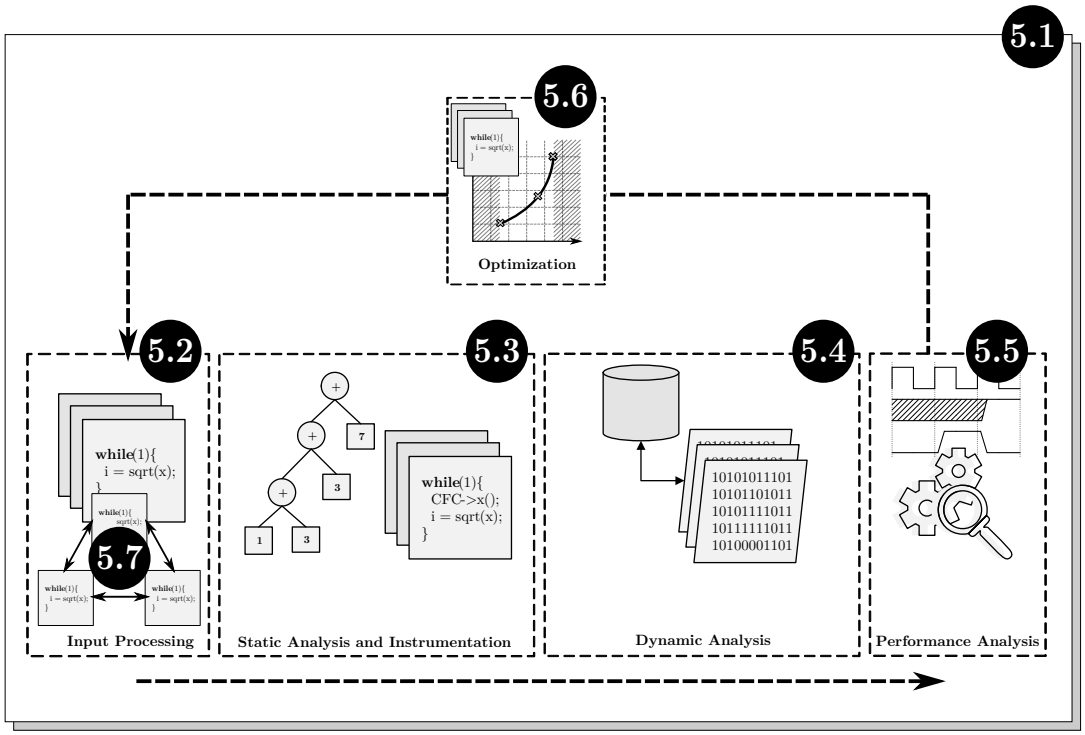


Figure 5.1: Overview of the cprof profiler.

5.1 Overview

The cprof profiler is divided into five systems, as shown in Figure 5.1. The first system is responsible for **input processing**, and will be discussed in Section 5.2. The next system deals with **static analysis** and **code instrumentation**. Static analysis is used to identify relevant code regions in the source code, represented using an *abstract syntax tree* (AST). This information is used to construct a database with information about variables, function calls and code structure in the program. Code instrumentation is used to insert cprof specific statements into the abstract syntax tree, and the result is an instrumented C++ program. Static analysis and code instrumentation are discussed in Section 5.3.

The next system is used for **dynamic analysis**. Dynamic analysis is used for profiling the behavior of the instrumented C++ program during run-time. Dynamic analysis is presented in Section 5.4. The

collected data is processed during the **performance analysis**, and is presented in Section 5.5. In this part of the profiler, the average and maximum of degree of parallelism are calculated, as well as the pipeline efficiency. A program profile is generated, which may be used for optimizing the source code.

The last part of the profiler is responsible for **optimization**. The relevant program parameters are stored in a database, and the user has the option to select kernels for optimization. Optimization is discussed in Section 5.6.1. The profiler supports **hierarhical program analysis**. Support for HPA is integrated throughout the profiler. The integration of HPA in cprof is explained in Section 5.7.

5.2 Input Processing

The input restrictions that apply to cprof are discussed in Section 5.2.1. The construction of an abstract syntax tree is discussed in Section 5.2.2.

5.2.1 Input Specification

The profiler supports a subset of C and C++ programs. The programs must be valid SANLPs (see Section 2.3), and statements must obey to the following rules:

- each computational statement is modeled as a function call;
- write arguments are passed as pointers to the function;
- read arguments are passed by value to the function;
- it is valid to have a single write argument as the left-hand side of the assignment operator, as long as the right-hand side is a function call;
- the **int**, **float**, **double**, **long** and **struct** (data) types are supported;
- use braces `{, }` for all `if` and `for` statements.

In Listing 5.1, a valid example program is shown:

```
1 #pragma cprof procedure filter
2 void filter() {
3   int a[N], x[N], y[N];
4   for(int j=0; j<N; j++) {
5     a[i] = source();
6   }
7   for(int j=0; j<N; j++) {
8     x[i] = foo(a[i]);
9     y[i] = bar(a[i]);
10  }
11  for(int j=0; j<N; j++) {
12    sink(x[i], y[i]);
13  }
14 }
```

Listing 5.1: Valid program in the C programming language.

In lines 5, 8, 9 and 12, examples of supported statements are shown. Variables are either declared as fundamental or compound type (arrays). There are no restrictions on the dimensions used by arrays. In line 5, the variable `a[i]` is modeled as the LHS of the assignment operator.

5.2.2 AST Construction

In Section 5.2.2.1, the compiler front-end Clang is introduced. The design and implementation of the pragma directive for kernel selection is discussed in Section 5.2.2.2

5.2.2.1 The Compiler Front-end

The Clang framework is a compiler front-end (see Section 2.4). The main reason for using Clang is that the framework provides accessible libraries for building and modifying the AST. The clang framework is part of cprof, and the cprof is responsible for setting up Clang. After initializing various objects, the `CompilerInstance` is responsible for creating the AST. Each node in the AST can be visited by implementing the `RecursiveASTVisitor` interface. Nodes in the AST are not modifiable. To modify nodes, a `Rewriter` object is required. This object is associated with the source code, and keeps its own copy of the AST in memory that can be modified. The profiler supports both C and C++. The main difference in constructing the AST for C or C++ programs is that Clang uses different classes and objects to model the AST of C++ programs.

5.2.2.2 Pragma Support

Pragmas are defined in both the C and C++ standard and provide additional information to the compiler. Cprof needs to know which kernels to profile, and pragmas are used for that purpose. Clang provides no easy integration of new pragmas. Therefore, the parts responsible for parsing source code and the semantic analysis were modified. The `#pragma` shown in line 1 of Listing 5.1 allows to select a kernel for profiling. The first part of the `#pragma` is a directive for the preprocessor to launch a `pragmahandler`. The second part, `cprof_procedure`, is the name of the pragma. After naming the pragma, a name for the cprof procedure is specified. In the case of the example it is `filter`.

5.3 Static Analysis and Instrumentation

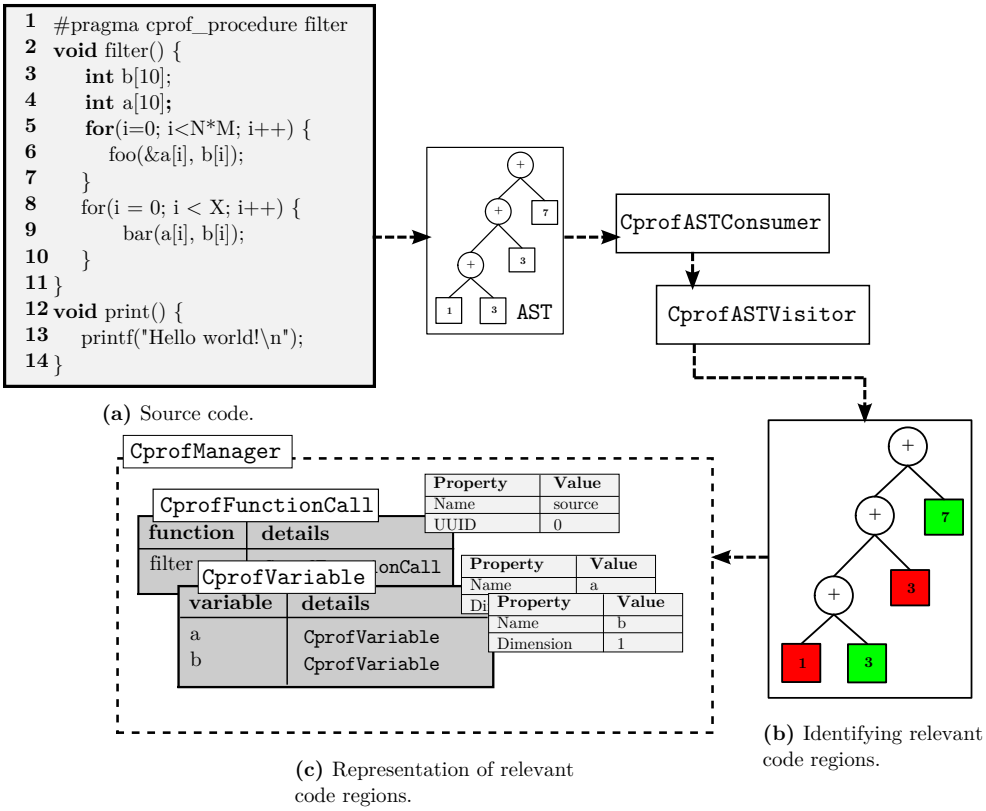
During static analysis, the relevant code regions are identified, and is discussed in Section 5.3.1. Following in Section 5.3.2, the process of code instrumentation is explained.

5.3.1 Static Analysis

In the following, we present the basic process of identifying code regions in the AST. After static analysis, a database with program information is generated and subsequently used during dynamic analysis. In Section 5.3.1.2, the process of storing the database on disk is explained.

5.3.1.1 Kernel and Statement Identification in the AST

The process of identifying and collecting relevant code regions is explained in the context of Figure 5.2. The source code is listed in 5.2(a). The AST constructed during input processing is consumed by the `CprofASTConsumer`, which is responsible for identifying relevant kernels. In this case, it accepts the kernel `filter` and ignores `print`. The reason is that `print` is not annotated with a `#pragma`.



(a) Source code.

(b) Identifying relevant code regions.

(c) Representation of relevant code regions.

Figure 5.2: Example of static analysis.

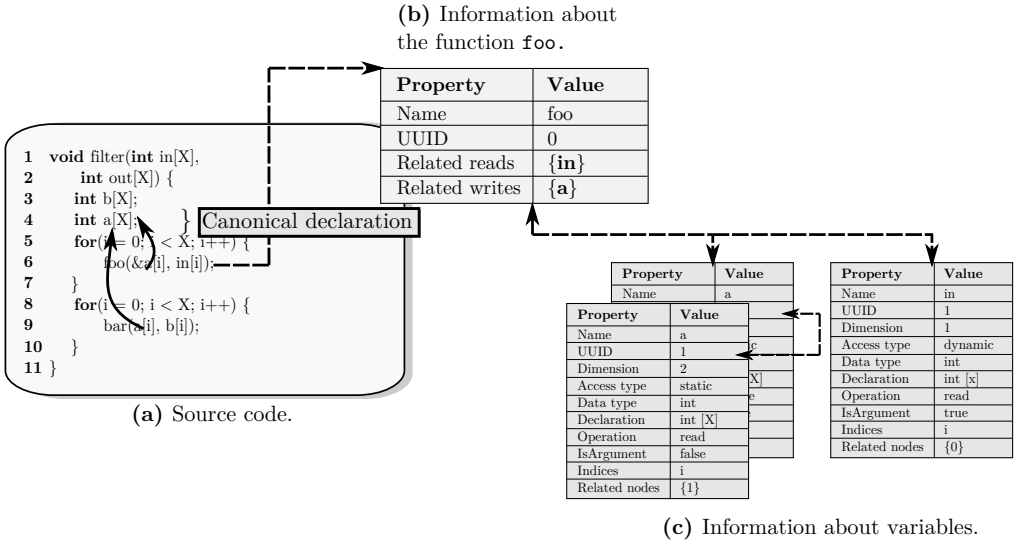
The AST of `filter` is inspected for relevant code regions. The class `CprofASTVisitor` is used for this purpose. In this example, the visitor finds the function call `foo` and `bar`, and the two variables `a` and `b` in line 6 and 9 of Figure 5.2(a), respectively. The marking of code regions is shown in Figure 5.2(b). The colors red and green represent code regions, which are not marked and marked, respectively.

The kernel `filter` is represented by a `CprofManager`. This object keeps a collection of function calls used in the kernel, in this example `foo` and `bar`. These function calls are modeled with `CprofFunctionCall` objects. For each variable, a `CprofVariable` is created with type information about the variable.

To relate variables in the source code, variable `a` must know about its canonical declaration. The canonical declaration is the initial declaration in the C code, in this example line 4 of Figure 5.2(a). For the purpose of illustration, we present an example of relating variables in Figure 5.3(a). The

function `foo` writes to variable `a` and the function `bar` consumes variable `a`.

During static analysis, the `CprofASTVisitor` detects that the canonical declaration of variable `a` is in line 4. Furthermore, the `CprofASTVisitor` knows that variable `a` is written in line 6, and read in line 9. The function `foo` keeps a list of related reads and writes, as shown in Figure 5.3(b). Figure 5.3(b) shows that variable `a` keeps a list of related nodes, shown in Figure 5.3(c). All the related nodes share the same shadow and control variables, thereby reducing memory usage.



(a) Source code.

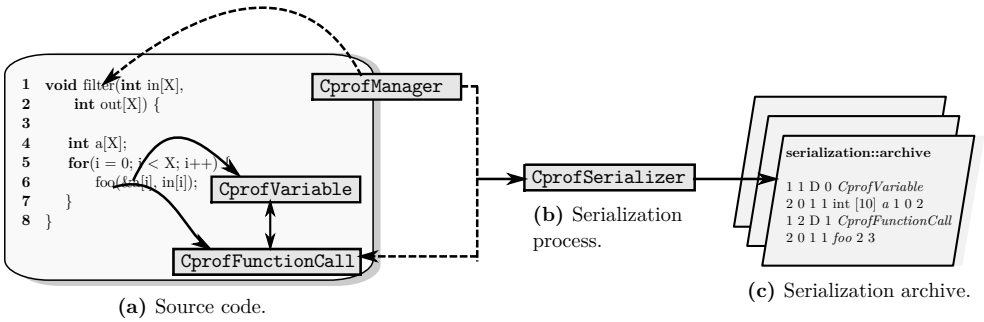
(b) Information about the function `foo`.

(c) Information about variables.

Figure 5.3: The canonical declaration and its relation to variable references.

5.3.1.2 Serialization of Objects

In computer science, in the context of transmission and data storage, serialization is the process of breaking down an arbitrary set of data structures into a sequence of bytes [37]. The serialization of object is required, as it stores a relational model of the program. This model is used during dynamic analysis to relate the variables. In Figure 5.4(a), variable `a[i]` is associated with the function `foo`. The function `foo` is represented by a `CprofFunctionCall` object, and is associated with a `CprofManager` object. The manager is responsible for the relation between the kernel and the code regions of interest.



(a) Source code.

(b) Serialization process.

(c) Serialization archive.

Figure 5.4: Serialization of cprof objects.

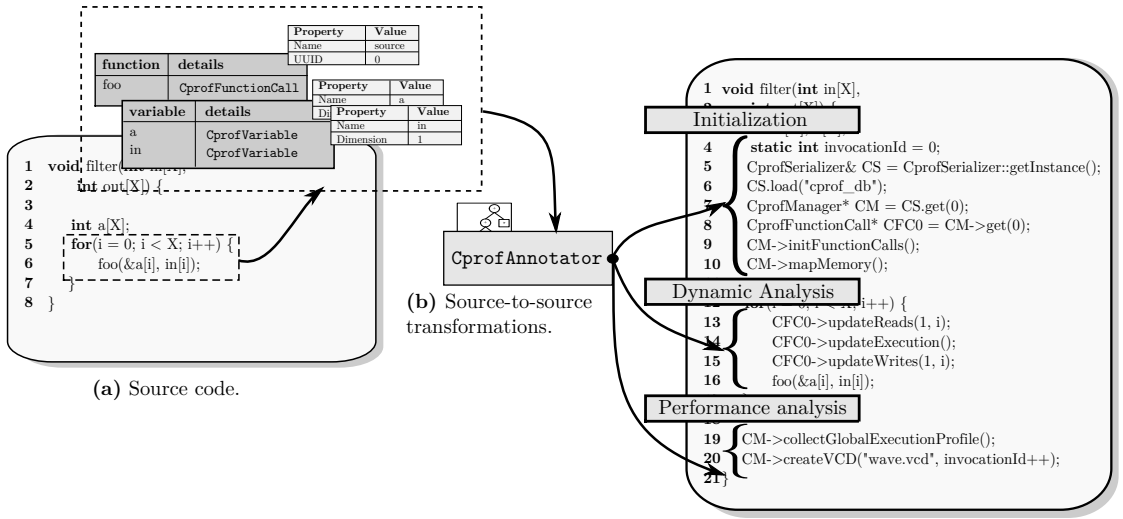
The information collected during static analysis is forwarded to the `CprofSerializer`, as shown in Figure 5.4(b). The `CprofSerializer` is responsible for serializing objects. Each class being processed by the `CprofSerializer` implements the Boost serializing interface [37]. In this case, the data is stored in a text file, and the result is shown in 5.4(c).

5.3.2 Instrumentation

After identification, objects were created to model the kernels, function calls, and variables. Section 5.3.3 explains how these objects relate to the instrumentation of source code.

5.3.3 Source-to-Source Transformations

The source code in Figure 5.5(a) is used to explain the process of instrumentation. The information stored during the serialization process is used by the `CprofAnnotator` object, shown in Figure 5.5(b). The `CprofAnnotator` creates a new source file with instrumented code. The instrumentation of code is discussed in the three separate sections: initialization (Section 5.3.3.1), dynamic analysis (Section 5.3.3.2), and performance analysis (Section 5.3.3.3).



(a) Source code.

(b) Source-to-source transformations.

(c) Instrumented source code.

Figure 5.5: Source code after instrumentation by cprof.

5.3.3.1 Code Instrumentation for Initialization

The `CprofAnnotator` inserts the initialization code to make dynamic analysis possible. The code in Figure 5.5(c) is used throughout this section. The code for initialization is always placed above all other statements in the kernel, as it assures that subsequent cprof statements use initialized objects.

In line 4, an invocation number is declared. For example, let us consider a program in which the kernel `filter` is called multiple times. In this case, the data associated with each invocation is stored in separate files.

In line 5 to 6, the `CprofSerializer` object is retrieved. After loading the data, the `CprofManager` with identification number zero is initialized, as shown in line 7. In line 8, the manager initialized the function calls.

The variables referenced by their canonical declaration use the same shadow variable for storing information about write operations. Since all memory used for the cprof statements is dynamically allocated, the memory mapping procedure called in line 10 is executed during run-time. The result is that the variables use the same shadow and control variables if they are related.

5.3.3.2 Code Instrumentation for Dynamic Analysis

During dynamic analysis, the statement execution profiles and shadow variables (see Chapter 4) are built and keep track of the behavior of the function calls. For each `CprofVariable`, the associated

index with the read operation is evaluated during run-time. The code in line 13 is inserted to make this evaluation possible. In line 14, the execution part of the function call is updated. Then, in line 15, the write operation is evaluated. In this case, the variable `a[i]` is written and the associated index `i` is evaluated during run-time.

5.3.3.3 Code Instrumentation for Performance Analysis

The `CprofManager` object is responsible for collecting all the execution profiles at the end of the program. Therefore, the code for performance analysis is inserted either before the last `return` statement in a function, or before the last brace if the `return` statement is missing. The code for collecting the statement profiles is shown in line 19. At the end of the program, an objects is responsible for creating a waveform, as shown in line 20.

5.4 Dynamic Analysis

To determine the performance, information is collected during dynamic analysis. The source file generated during code instrumentation is used for this purpose. In Section 5.4.1, the compilation and initialization of the instrumented code is discussed. Then, in Secton 5.4.2, the algorithms used for dynamic analysis and data collection are presented.

5.4.1 Compilation and Initialization

The instrumented source code is a C++ program, where all selected kernels have annotations. In Section 5.4.1.1, the necessary steps in compiling the instrumented code are discussed. The initialization of the code during run-time is discussed in Section 5.4.1.2.

5.4.1.1 Compilation of Instrumented Code

The C++ program generated by cprof can be compiled with all modern C++ compilers, on both Microsoft Windows and Linux. For each C++ program, a `CMake` file [38] is generated to manage the build of the project. More details on compilation are described in Appendix B.

5.4.1.2 Deserialization of Objects

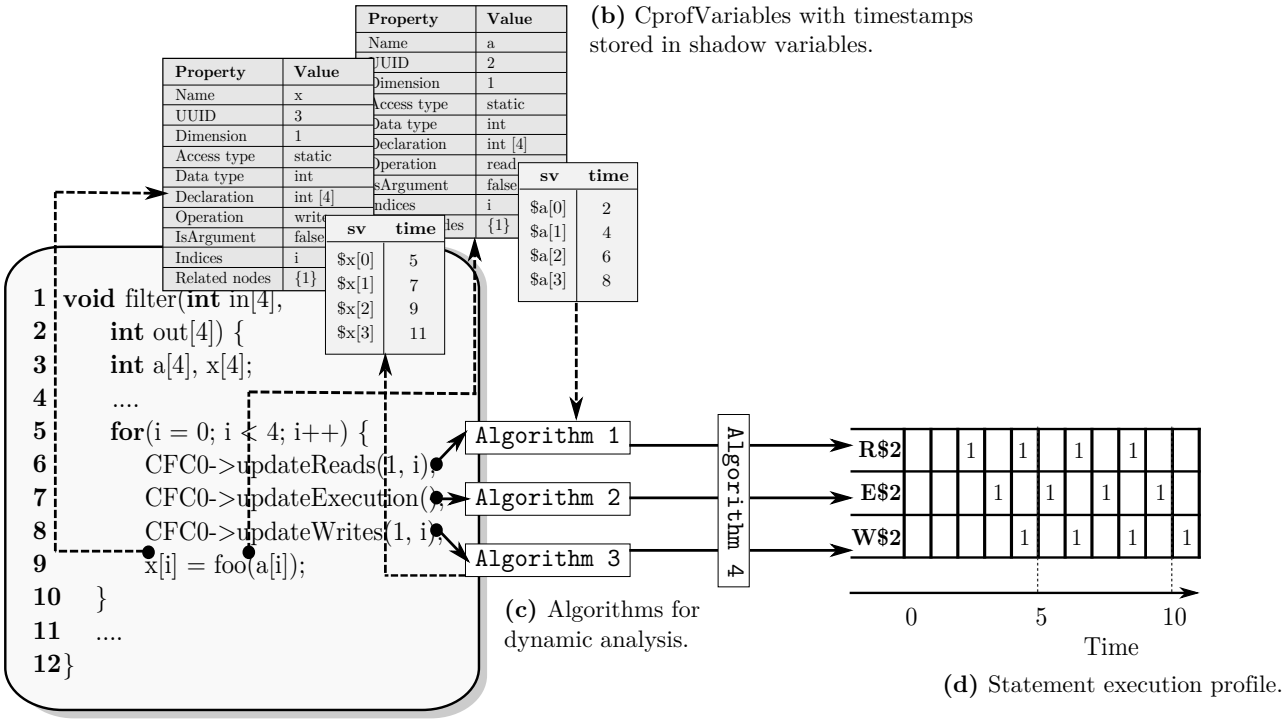
The cprof statements collected during static analysis are loaded during run-time. To make this possible, the data on disk is read, and the objects are deserialized. The `CprofSerializer` is asked to load the data from disk. After deserializing the archive, the statements can be tracked by cprof.

5.4.2 Algorithms for Dynamic Analysis

In this section, we present the algorithms used for dynamic analysis and the execution profiles are presented. In Section 5.4.2.1, the algorithms used during dynamic analysis are introduced. The algorithms for read, execute, and write operations are discussed in Sections 5.4.2.2, 5.4.2.3 and 5.4.2.4, respectively. In Section 5.4.2.5, the algorithm for constructing statement execution profiles is discussed.

5.4.2.1 Overview of Algorithms

In this section, we give an overview of the algorithms used during dynamic analysis. In Figure 5.6(a), the cprof statements are mapped to algorithms. The cprof variables, shown in Figure 5.6(b), provide access to shadow variables, which allows to find out when a variable is written. The algorithms for keeping track of execute and write operations are mapped as well. In Figure 5.6(c), the algorithms are listed and the output of all the algorithms combined is shown in Figure 5.6(d).



(a) Instrumented source code.

(b) CprofVariables with timestamps stored in shadow variables.

(c) Algorithms for dynamic analysis.

(d) Statement execution profile.

Figure 5.6: Overview of algorithms used in dynamic analysis.

To see the example output generated by the algorithms discussed in the following sections, we refer to the case studies in Chapter 4 (see Section 4.3). The case studies show the capabilities of the algorithms, as cprof was used to generate the tables and numbers for the absolute and unbounded throughput estimates in Section 4.3.

5.4.2.2 Read Operations

In the following, we discuss the algorithm used for keeping track of read operations is discussed. Each statement that reads at least one variable is preceded by the `updateReads` function. In Algorithm 1, the procedure is summarized.

In line 2 and 3, the start and stop value are determined. The start time is the control variable ($C\$s$), and the stop time of the read operation is determined by the latency of the read operation (Λ_R). The `maxStop` variable in line 4 is used to determine the latest write time associated with one of the read arguments.

In line 5, each read argument associated with the function call is visited. Then, each of the variables returns its number of dimensions in line 6. These indices, stored in an array and taken from the `indices` list, are passed as arguments to the `updateReads` function.

In line 9, the variable is asked to check at which time it is written. If it is later than `maxStop`, the value in `maxStop` is substituted with the new timestamp. If the program is profiled for the absolute throughput estimate, the control variable ($C\$s$) must be taken into account to determine the stop time of the read operation. In the case of the unbounded throughput estimate, the control variable ($C\$s$) is set to `maxStop`, because it can start its execution as soon as the data is available.

In Line 14, the read profile is updated, by calling the `updateStatementProfile(start, stop)` function, which is the subject of a later discussion. The space complexity of Algorithm 1 is $\mathcal{O}(c + r)$, and the time complexity is $\mathcal{O}(c \cdot r)$, where r is the number of read arguments identified by cprof, and

Algorithm 1 Update shadow variables for read access.

Precondition: *indices* is a list of indices for accessing the shadow variables of length n

```
1: function UpdateReads(indices)
2:     start ← C$s
3:     stop ← C$s + Λ_R
4:     maxStop ← 0
5:     for each argument a in readArguments do
6:         dims ← a.getDimensions()
7:         for y = 0 to dims do
8:             l.push(indices[y])
9:         a.getWrites(l, &maxStop)
10:    if UnboundedThroughputEstimate then
11:        C$s ← maxStop
12:    else
13:        C$s ← max(stop, maxStop)
14:    UpdateStatementProfile(start, stop, Λ_R)
```

c the number of dimensions associated with the read argument.

5.4.2.3 Execute Operations

In the following, the algorithm used for keeping track of execute operations is discussed. Each statement is preceded by `updateExecution` function. The procedure is summarized in Algorithm 2.

Algorithm 2 Update the execute statement profiles.

```
1: function UpdateExececution
2:     start ← C$s
3:     stop ← C$s + Λ_F
4:     UpdateStatementProfile(start, stop, Λ_F)
5:     if AbsoluteThroughputEstimate then
6:         C$s ← C$s + II_F
```

In line 2, the start time of the the execute operation is determined. In this case, start simply equals the control variable ($C\$s$). In line 3, the stop time is determined. The stop time equals the control variable ($C\$s$), including the function latency (Λ_F). In line 4 the statement profile is updated. If the program is profiled for the absolute throughput estimate, the initiation interval (II_F) must be taken into account. The space complexity of Algorithm 2 is $\mathcal{O}(1)$, and the time complexity is $\mathcal{O}(1)$.

5.4.2.4 Write Operations

In this section, we discuss the algorithm used for keeping track of write operations. Each statement that writes at least one variable is preceded by `updateWrites` function. In Algorithm 3, the procedure is summarized.

In line 2 the start time and stop time of the write operation are initialized. For both the start and stop times the function latency (Λ_F) is included. The stop time of the write operation is modeled using the write latency (Λ_W).

Algorithm 3 Update shadow variables for write access.

Precondition: *indices* is a list of indices for accessing the shadow variables of length n

```
1: function UpdateWrites(indices)
2:     start ← C$s + Λ_F
3:     stop ← C$s + Λ_F + Λ_W
4:     if AbsoluteThroughputEstimate then
5:         start ← start − II_F
6:         stop ← stop − II_F
7:     maxStopValue ← 0
8:     for each argument a in writeArguments do
9:         dims ← a.getDimensions()
10:        for y = 0 to dims do
11:            l.push(indices[y])
12:        a.updateWrites(l, stop)
13:    UpdateStatementProfile(start, stop, Λ_W)
```

In the case of the absolute throughput estimate, the start and stop times are corrected for the initiation interval (II_F) in line 5 and 6, respectively. If this is not taken into account, the write operation could possibly start too late.

In line 8, each write argument associated with the function call is visited. Then, each variable returns its number of dimensions in line 9. These indices are subsequently stored in an array and taken from the `indices` list. These indices are passed as arguments to the `updateWrites` function. In line 12 each variable `a` is updated with the stop time associated with the write operation for the given indices. In Line 13, the write profile is updated, by calling the `updateStatementProfile(start, stop)` function. The space complexity of Algorithm 3 is $\mathcal{O}(d + w)$, and the time complexity is $\mathcal{O}(d \cdot w)$, where w is the number of write arguments identified by cprof, and d the number of dimensions associated with the write argument.

5.4.2.5 Statement Execution Profiles

The execution profiles storing information about the read, execute, and write operations, and grow linearly in time. For example, consider a program that takes 2^{32} time units to finish. If the profiler is told that at time 2^{32} one write operation is in progress, it needs to store that information in the `W$s` variable. Now, storing this information at index 2^{32} of the `W$s` variable is expensive. The `W$s` is an array of unsigned long integers, costing 8 bytes per element. As a result, almost 35 gigabytes is necessary to store this information.

Therefore, an algorithm is used to store this data in an efficient way. We use two ways to store the statement execution profile in an efficient way. First, we only keep track of zeros, i.e., only store when there is no operation in progress. Second, we keep track of sequence of zeros, and update the interval map with a new stop time if successive zeros are found. The two optimizations are summarized in Algorithm 4.

In line 2, the latency of the operation is used to check if there is a gap between successive operations. In Line 3, the stop time of the previous operation is subtracted from the current operation start time. The algorithm detects a repetitive pattern, if this result is equal to the step-size calculated in line 10. The step-size is the interval length between successive zeros.

Then, if there is no pattern found, in line 7, a new object is created with the previous stop time, the current start time, and the initialized step-size. In line 10 and 11, the step-size and the previous stop time are stored for later use.

Algorithm 4 Update algorithm for the statement execution profiles.

Precondition: *start* is the start time of the operation and *stop* is the stop time and *latency*.

```
1:  function updateStatementProfile(start, stop, latency)
2:      if (start − prevStop) > latency then
3:          if (start − prevStop) = stepSize then
4:              intervalMap.back().stop ← start;
5:              intervalMap.back().stepSize ← stepSize;
6:          else
7:              opStepSize ← 0;
8:              timeInterval ← (prevStop, start, opStepSize);
9:              intervalMap.push(timeInterval);
10:     stepSize ← start - prevStop;
11:     prevStop ← stop;
```

The space complexity of Algorithm 4 is $\mathcal{O}(m)$, whereas the time complexity is $\mathcal{O}(1)$, where m is the number of intervals. The interval map is used to reconstruct the statement execution profile in the performance analysis. This is discussed in Section 5.5.

5.5 Performance Analysis

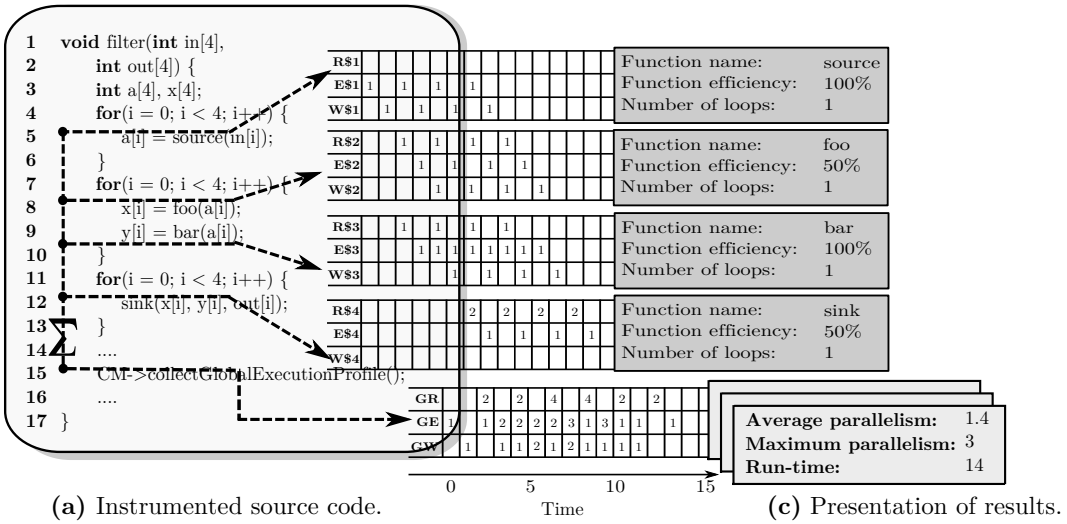
In Section 5.5.1, data processing and presentation of measurements are explained. Then, in Section 5.5.2, waveform generation is discussed. In Section 5.5.3, the system responsible for storing program information is presented.

5.5.1 Data Processing and Presentation

The algorithms discussed in Section 5.4.2 produce statement execution profiles. To derive performance metrics from the produced output, the data is processed by the `CprofManager`. This is shown in line 15 of Figure 5.7(a). The `CprofManager` asks each `CprofFunctionCall` object to reconstruct the statement execution profiles based on their measurements. Given the possibility that the size of these statement profiles is more than the memory system can handle, the execution profiles are constructed for a specified interval. This interval is specified using an offset, that is shifted each time the execution profile for the interval is reconstructed. After data in this interval is collected for each function, the global execution profile is scanned for peak parallel performance. The statement execution profiles are shown in Figure 5.7(b).

In 5.7(c), the metrics are shown. The run-time is determined by looking in the global execution profile for the latest execute or write operation. The efficiency of a function is given as a first-order hint of performance. The function `source` is 100%, as the initiation interval of the function is 2. The function `foo` starts at time 0 and the last iteration stops at time 7. In the execution interval, $[0, 7]$, four slots are used. The efficiency of the function is, therefore, $\frac{4}{8} \times 100 = 50.0\%$

A more interesting case is the function `bar`. Looking at the flat statement execution profile would suggest that this pipeline is 100% efficient. However, the function latency is two cycles instead of one. To determine the efficiency in this case, we zoom in on the execution profile, as shown in Figure 5.8. The resulting performance is 50%, instead of 100% shown in Figure 5.7(c).



(a) Instrumented source code.

(b) Statement execution profiles.

(c) Presentation of results.

Figure 5.7: Overview of processing and presenting the output of dynamic analysis.

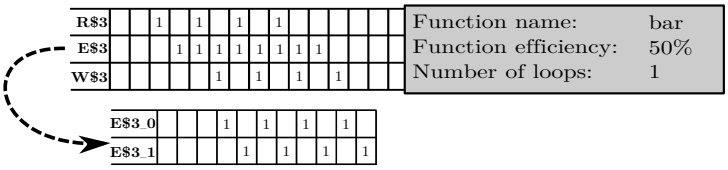


Figure 5.8: Performance of the function bar after a closer inspection.

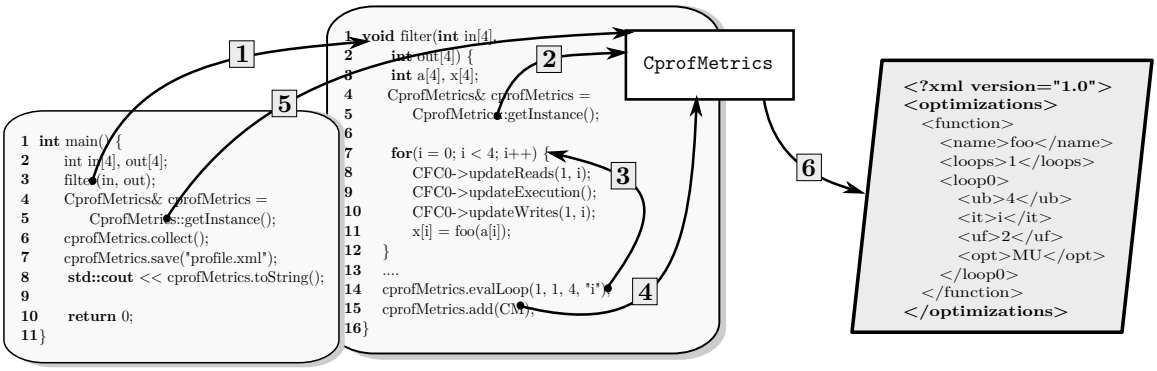
5.5.2 Waveform Generation

The processed data is used to generate waveforms to provide an insight to the behavior of the C programs, when implemented as a PPN in hardware. The *Value Change Dump* (VCD) format [39] is used to present information about value changes on selected signals. Cprof automatically generates VCD files when a kernel is finished executing.

5.5.3 Program Profile Generation

The data produced provides information about the efficiency of function calls. To improve the efficiency of a function we can apply optimizations. To perform these optimizations automatically, we need a program profile. In Figure 5.9, the approach used to generate the program profile is shown. The arrows are numbered, indicating the order the of operations in generating the program profile.

In line 3 in Figure 5.9(a), the `filter` function is called. Then, in line 2 of Figure 5.9(b), the `CprofMetrics` object is initialized. This object is responsible for keeping track of program information. In line 14 in 5.9(b), the loop associated with the function `foo` is evaluated. The unique identification number of the `CprofManager` and of the function `foo` are the first two arguments to `evalLoop`. The last two arguments represent the upper bound of the loop and the iterator. In line 4 of Figure 5.9(b), this information is saved in the `CprofMetrics` object. In each program, the required code for storing the program profile on disk is inserted in the `main` function of the program. This is shown in line 6 of Figure 5.9(a). Statement information, such as loop bounds and iteratores, are stored in the program profile that is used for optimizing the program.



(a) Program entry. **(b)** Profiled function. **(c)** Program profile in XML.

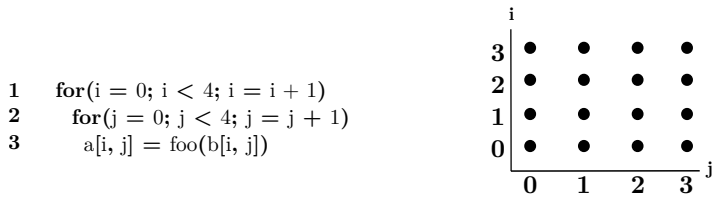
Figure 5.9: Generation of program profile.

5.6 Optimization

The absolute and unbounded throughput estimate represent the lower and upper bound of the design space. Between the lower and upper bound, many alternative design points may exist. In Section 5.6.1, the optimization methods used for visiting design points in the design space are discussed.

5.6.1 Methods

Cprof implements two optimization techniques: modulo unfolding (Section 5.6.1.1), and plane cutting (Section 5.6.1.2). Modulo unfolding and plane cutting split-up the process by assigning process iterations to different partitions. These optimizations are easily modeled in the C programming language. Loop skewing is another possible optimization possible to model in the C programming language. Anyhow, it is not part of this work. Other methods for optimizing PPNs are found in [2, 9] However, the algorithms for other optimization techniques require the derivation of the PPN. A one-to-one mapping of such algorithms to C code is impossible, as the level of abstraction in the C programming code is too high. Modulo unfolding and plane cutting are explained in the context of Figure 5.10. The program in Figure 5.10(a) consists of two for loops, and in line 3 the value `a[i, j]` is the result of some value produced by the function `foo`. There are no dependencies between variables, as shown in the dependency graph in Figure 5.10(b).

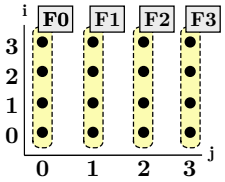


(a) Source code. **(b)** Dependency graph.

Figure 5.10: Sample program to show optimizations.

5.6.1.1 Modulo Unfolding

Modulo unfolding splits-up a process p into N partitions [2]. If modulo unfolding is applied to the code shown in Figure 5.10(a), it is possible to map each iteration of the function `foo` to its own process. As an example, we use modulo unfolding to map each iteration onto the same process for which the iterator `j` = 0. This selection of iteration points of the iterator `j` are shown in Figure 5.11(a).



(a) Modulo unfolding.

```
1  for(i = 0; i < 4; i = i + 1)
2    for(j = 0; j < 4; j = j + 1) {
3      if (j % 4 == 0)
4        a[i, j] = foo(b[i, j]);   // F0
5      if (j % 4 == 1)
6        a[i, j] = foo(b[i, j]);   // F1
7      if (j % 4 == 2)
8        a[i, j] = foo(b[i, j]);   // F2
9      if (j % 4 == 3)
10       a[i, j] = foo(b[i, j]);   // F3
11 }
```

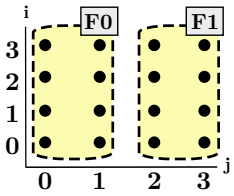
(b) Modulo unfolding in C code.

Figure 5.11: Modulo unfolding applied.

The C program optimized with modulo unfolding is shown in Figure 5.11(b). In hardware, each iteration is mapped onto its own process. The inner loop is represented using four different processes, which start executing as soon as data is available. The result is that the inner loop is fully unrolled in the process network.

5.6.1.2 Plane Cutting

Plane cutting splits-up a process p into N subdomains [2]. If plane cutting is applied for two subdomains to the code shown in Figure 5.10(a), the iterations are mapped as shown in Figure 5.12(a).



(a) Plane cutting.

```
1  for(i = 0; i < 4; i = i + 1)
2    for(j = 0; j < 4; j = j + 1) {
3      if (j >= 2)
4        a[i, j] = foo(b[i, j]);   // F1
5      else if (j < 2)
6        a[i, j] = foo(b[i, j]);   // F0
7  }
```

(b) Plane cutting in C code.

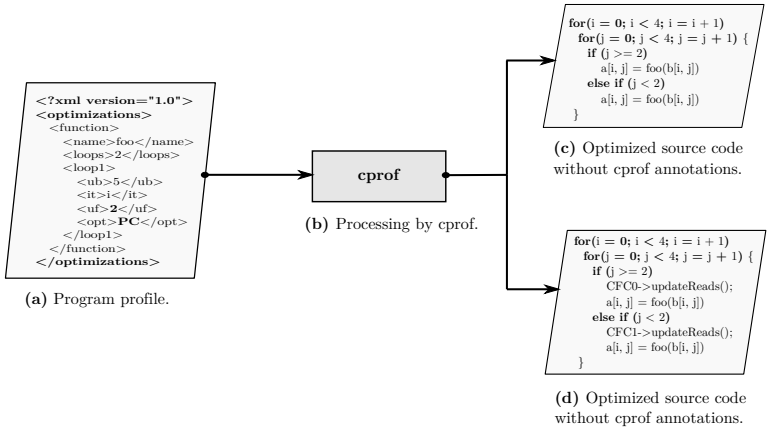
Figure 5.12: Plane cutting aplied.

The C program optimized with plane cutting is shown in Figure 5.12(b). As a result, the iterations points in the left rectangle of Figure 5.12(a) are mapped onto the function call in line 6 of Figure 5.12(b). The right rectangle is mapping the points that are mapped onto the function call in line 4 of Figure 5.12(b).

5.6.2 Implementation of Optimizations

In Section 5.5.3, program profiles were introduced. The program profile provides information about the functions in the program. The profiles stores the number of loops enclosing the function call, the upper bound, and used iterators. At this moment, cprof only supports the optimization the innermost loop. For inner loops, it is possible to specify the unroll factor used by the modulo unfolding and plane cutting.

During program profile generation, the modulo unfolding option (MU) is selected as default option for optimization. If, however, a variable is read and written in the same statement, this is classified as a self-dependency. In this case, program profile selects plane cutting as default method for optimization. This choice is based upon the selection criteria for optimization methods presented in Meijer's work [9]. In his work the direction of the dependencies is used for determining which optimization to apply. However, cprof does not know the direction of the self-dependencies as no fully PPN is derived.



(a) Program profile.

(b) Processing by cprof.

(c) Optimized source code without cprof annotations.

(d) Optimized source code without cprof annotations.

Figure 5.13: Overview of the optimization flow in cprof.

Cprof parses the optimization file depicted in Figure 5.13(a) generated by the profiler, and substitutes the statements with their optimized equivalents, as shown in Figure 5.13(b). The results of the source-to-source translation results in two files. The first file is not instrumented for profiling, as shown in Figure 5.13(c). This file can then be used in tools, such as Compaan and Daedalus, to derive the polyhedral process network. The second file is instrumented, as shown in Figure 5.13(d). This file can be used to profile the optimized program. The process for annotating optimized source code is no different from other programs, and cprof applies the regular systems and techniques to profile the program.

5.7 Hierarchical Program Analysis

HPA is about profiling programs with inter-procedural relations. In HPA, manual kernel selection is not necessary, as cprof automatically detects SANLPs. This is discussed in Section 5.7.1. In Section 5.7.2, the process of code instrumentation for HPA is explained. The changes required for dynamic analysis are discussed in Section 5.7.3. The collection of performance metrics in HPA is the subject of Section 5.7.4.

5.7.1 Static Analysis

Figure 5.14 is used throughout this section. To illustrate the process of automatic kernel selection in HPA, Figure 5.14(a) shows the entry point of the program. In line 7 of Figure 5.14(a), the function `producer` is called. In this case, the call to `producer` is marked for instrumentation. The reason is that the body of `producer` does not contain any function calls. In line 9, the function `hierarchy` is called. In this case, the call to `hierarchy` is not marked for instrumentation, as the body of `hierarchy` contains code that is annotated by cprof. If this is the case, the function `hierarchy` is responsible for keeping track of the variables `data_a` and `data_out`. To summarize, cprof detects SANLPs and annotates them. Otherwise, the statement is ignored.

In Figure 5.14(b), the function `hierarchy` is defined. In line 9 of Figure 5.14(b), the `assign` function is called. The `assign` call itself does not contain any code that is recognized by cprof. Therefore, this call is marked for instrumentation. In line 11, the function `transformer` is called. This function is responsible for tracking the variables `a` and `b`.

The most complicated function is defined in Figure 5.14(c). In this case, each call to `assign` is marked for instrumentation. However, the calls in line 19 and 20 are not, as the functions `foo` and `bar` have code statements that can be profiled. The definitions of both `foo` and `bar` are given in Figure 5.14(d) and 5.14(e), respectively. The calls made in line 9, 12 and 15 of both functions do not contain

any code that is marked for instrumentation.

The hierarchy in the program is flattened by cprof. Each function call in 5.14 that is not marked for instrumentation could be replaced by their implementation. For example, the body of `hierarchy` in 5.14(b) could replace the call in line 9 in Figure 5.14(a). This is the approach cprof takes on marking the statements for HPA.

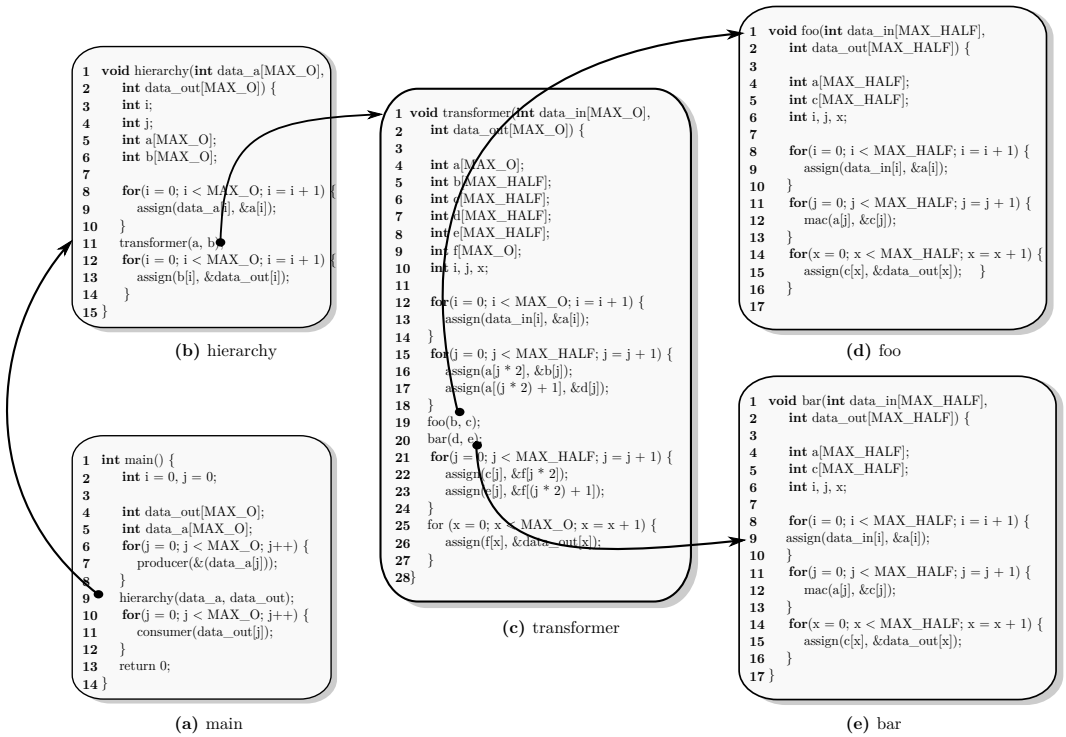


Figure 5.14: Example program with hierarchy used for HPA.

5.7.2 Instrumentation

The problem in HPA is that the data structures used for shadow variables must be shared inter-procedural. In Figure 5.15, the function `hierarchy` is annotated.

In line 15, the statement responsible for relating memory is shown. Since cprof knows about the dimensions of the variables, it is possible to insert a statement to find out the base memory address of a variable. The address of the first element is the base memory of arrays, and for non-array types just the address of the variable itself. Variables are related during run-time using this information. In line 16, the function `relateVariables` is inserted for this action. This function is responsible for finding out which functions are using the same variables based on their memory address. After registering the relations between variables, the memory used for shadow variables is shared between variables by the call to `mapMemory` in line 17. The bookkeeping from line 20 to 23 is the same as it is for regular programs and requires no different instrumentation.

From line 27 to 30, statements are inserted for performance analysis. The performance metrics are presented to the user after each invocation of the kernel. In line 30, the performance metrics are stored in the `CprofHPAManager` object. This object is responsible for collecting performance metrics during HPA.

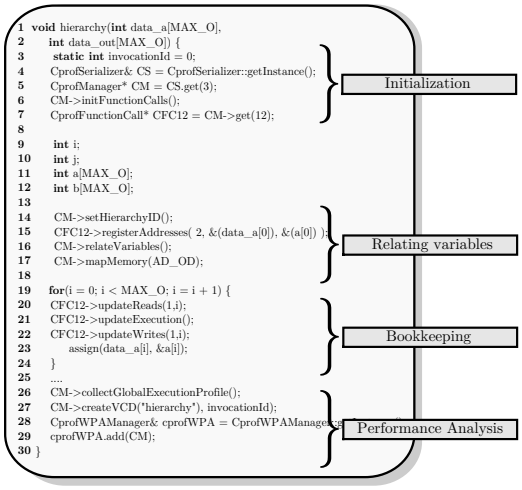
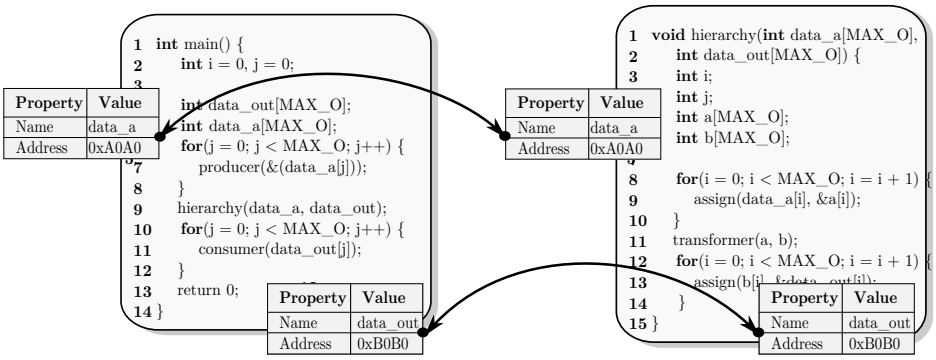


Figure 5.15: The hierarchy function instrumented with support for HPA.

5.7.3 Dynamic Analysis

In the previous section, it was shown how the code was annotated to determine inter-procedural relationships during run-time.



(a) Memory addresses in main. **(b)** Memory addresses in hierarchy.

Figure 5.16: Inter-procedural relations between variables.

In Figure 5.16(a), the memory addresses of the variables `data_a` and `data_out` are shown. By using this information, the relationship with the variables in the function `hierarchy`, shown in 5.16(b), is established. After establishing the relationship, it is possible to keep track of the behavior of hierarchical programs.

5.7.4 Performance Analysis

At the end of each function invocation, the performance metrics of the kernel are collected and presented. The `CprofHPAManager` is responsible for storing for information about each invocation of a kernel, such as the average and maximum parallelism as well as the run-time. Each program has an entry point, and in regular C code this is the `main` function. During static analysis, the `main` function is located and code is inserted to retrieve the data collected by the `CprofHPAManager`. The collection of program profiles, as discussed in Section 5.5.3, is the same for HPA programs.

5.8 The Cost of Profiling

Profiling comes with a price in terms of both space and time complexity. For each canonical declaration in the source code, a shadow and control variable are created. In Table 5.1, the cost of profiling in terms of space and time complexity of cprof are listed. The memory footprint of the generated program by cprof is at least twice the size of the original program. The reason is that for each variable, control and shadow variables are used to keep track of the performance of the program. The space complexity of profiling scales linearly with the number of statements used in the program.

The space complexity introduced by cprof in the instrumented program is $\mathcal{O}(n + v)$, where n is the number of variable declarations in the original program that are marked by cprof, and v is the dimension of each marked variable declaration. The time complexity introduced by cprof in the instrumented program is dominated by $\mathcal{O}(u \cdot v)$, where u is the number of times each variable is referenced in the program. Non-array types are modeled as arrays with one dimension and, hence, v never equals zero.

Algorithm	**Space**	**Time**
Dynamic Analysis of Read Operations	$\mathcal{O}(c + r)$	$\mathcal{O}(c \cdot r)$
Dynamic Analysis of Execute Operations	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Dynamic Analysis of Write Operations	$\mathcal{O}(d + w)$	$\mathcal{O}(d \cdot w)$
Updating the Statement Execution Profile	$\mathcal{O}(m)$	$\mathcal{O}(1)$

Table 5.1: Profiling cost in space and time. Used notations: c is the number of dimensions of the read argument, r is the number times the read arguments are referenced throughout the program, d is the number of dimensions of the write argument, w is the number times the write arguments are referenced throughout the program, and m is the number of statement intervals.

5.9 Summary and Conclusions

In this chapter, we have presented the design and implementation of cprof. Cprof has the ability to profile C and C++ code. The Clang framework used for implementing cprof provides libraries for source-to-source transformations. These transformations allows us to annotate the source code with statements for dynamic analysis. The performance analysis provides suggestions for optimization, and developers have the possibility to optimize the program. The optimized program can subsequently be used to explore the design space. Cprof supports hierarchical program analysis, making it possible to analyze the behavior of programs with inter-procedural relations. At the end of the chapter, we presented the cost of profiling. In the next chapter, we verify the implementation of the solution approach in cprof.

6 Verification

In this chapter, the implementation of the solution approach in cprof is verified. In Secton 6.1, we present the verification approach. Processes in PPNs communicate with each other in four different ways. In Section 6.2, we will show that each communication model is supported by cprof. In Section 6.3, we will show that it is possible to increase throughput by applying program optimizations. Following in Section 6.4, the verification of hierarchical program analysis is presented. In Section 6.5, a summary is given and conclusions are drawn.

6.1 Verification Approach

Cprof estimates the performance of C programs, based on the assumption that the C code is implemented as a PPN in hardware. The measurements performed by cprof are, therefore, validated against RTL implementations generated by the Compaan DDE. The Compaan DDE generates ISE projects that Xilinx ISE Simulator (ISim) can simulate. The FPGA board used is a Virtex-6 FPGA (xc6vlx240t).

The clock period used in Compaan DDE is 10 ns, and the clock frequency is 100 MHz. The estimates by cprof use the same clock period. The function latency (Λ_F) is 3 cycles, and the initiation interval (II_F) is 1 cycle. This configuration of the function latency and initiation interval applies to each program throughout this chapter, unless stated otherwise.

The validation approach is to compare the execution finish time from Xilinx ISim with the execution finish time found by cprof. Xilinx ISim is used to establish the execution finish time of a PPN in hardware. The execution finish time estimated by cprof is compared to this number. For the validation of the communication models, waveforms generated by Xilinx ISim are used to verify the estimates of cprof. For the purpose of clarity, the waveforms produced by cprof and ISim are recreated in Wavedrom [40].

6.2 Verification of the Communication Models

In this section, the following four communication models are verified: in-order without multiplicity (Section 6.2.1), in-order with multiplicity (Section 6.2.2), out-of-order without multiplicity (Section 6.2.3), and out-of-order with multiplicity (Section 6.2.2). In PPNs, each communication channel is implemented as a FIFO, and has one producer and one or more consumer nodes. The producer and consumer form a P/C pair.

The production order is determined by the `producer` node. If the `consumer` node consumes the tokens in the same order, we say that the communication between the producer and consumer is in-order. Otherwise, the communication between the producer and consumer is out-of-order. If the communication model has no multiplicity, it means that the tokens produced are not used in future executions of the consumer process. On the other hand, if the model has multiplicity, it means that the tokens produced are used in future executions of the consumer node.

In the following subsections, each program used for the verification of communication models consists of the following three nodes: `producer`, `consumer` and `sink`. The relationship between the `producer` and `consumer` node is shown in Figure 6.1. The `sink` node is used to stream data out of the process network. The number of iterations is bound by the variable `X`, where `X` is 5. For the purpose of clarity, the generated waveforms used for the verifcation of the communication models are shown in Appendix F.

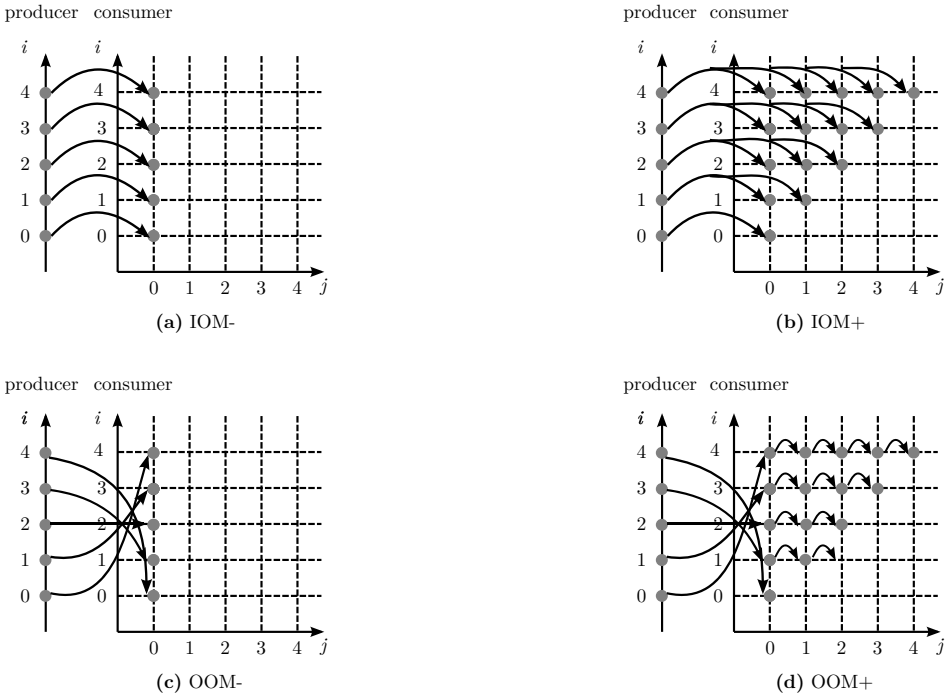


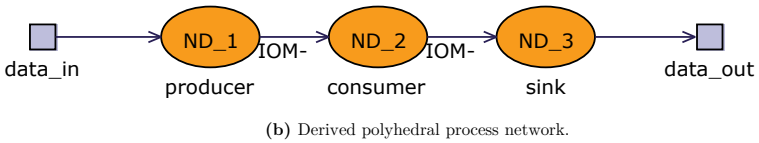
Figure 6.1: Communication models in PPNs.

6.2.1 In-Order without Multiplicity (IOM-)

In Figure 6.2(a), an example source code is shown for implementing the IOM- communication model. The `producer` and `consumer` are shown in line 2 and 5 of Figure 6.2(a). In Figure 6.2(b), the derived PPN of the source code is shown. The communication pattern between `producer` and `consumer` is correctly identified as IOM-.

```
1  for(i=0; i<X; i++) {
2    producer(&a[i], data_in[i]);
3  }
4  for(i=0; i<X; i++) {
5    consumer(a[i], &b[i]);
6  }
7  for(i=0; i<X; i++) {
8    sink(b[i], &data_out[i]);
9  }
```

(a) Source code implementing IOM-.



(b) Derived polyhedral process network.

Figure 6.2: Example implementation of the IOM- communication model.

The source code is profiled by cprof, and the waveform shown in Figure F.1(a) of Appendix F is generated. After RTL simulation, Xilinx ISim generates the waveform shown in Figure F.1(b) of Appendix F. Cprof estimates that it takes 190 ns to finish execution, and the RTL simulation is finished after 210 ns. The two waveforms are almost identical. The difference between both waveforms is that in Figure F.1(b) of Appendix F, the `producer` stalls its read operation for two cycles. This stall in the read stage in hardware is undefined behavior in Compaan DDE, and seems to represent a bug in the hardware.

6.2.2 In-Order with Multiplicity (IOM+)

In Figure 6.3(a), example source code is shown for the IOM+ communication model. In line 6 of Figure 6.3(a), the `consumer` node is enclosed by two loops. The reason is that in this case, the consumer consumes multiple tokens in following iterations. For example, in the second iteration, the token stored in `a[1]` is consumed twice by the consumer. The derived PPN is shown in Figure 6.3(b). The communication pattern between `producer` and `consumer` is correctly identified as IOM+.

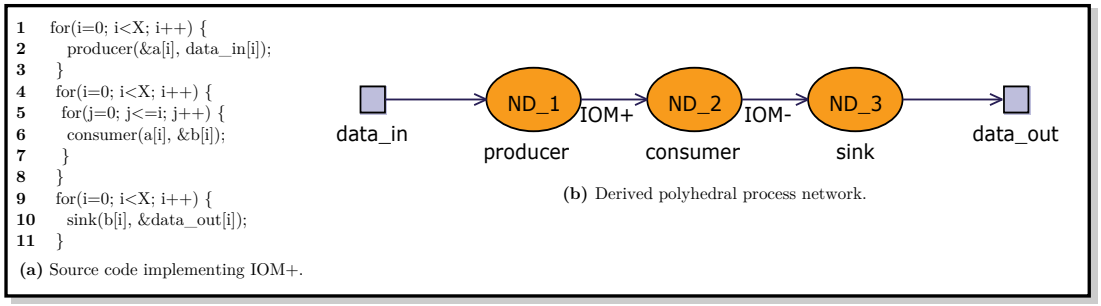


Figure 6.3: Example implementation of the IOM+ communication model.

The source code is profiled by cprof, and the waveform shown in Figure F.2(a) of Appendix F is generated. After RTL simulation, Xilinx ISim generates the waveform shown in Figure F.2(b) of Appendix F. Cprof estimates that it takes 290 ns to finish execution, and the RTL simulation is finished after 310 ns. In this case, the consumer takes more time to execute, because of the two loops enclosing the statement. The two waveforms are almost identical. The difference between the waveforms is that in Figure F.1(b) of Appendix F, the `producer` stalls its read operation for two cycles. The stall is, in this case, not caused by the communication model.

6.2.3 Out-of-Order without Multiplicity (OOM-)

In Figure 6.4(a), example source code is shown for implementing the OOM- communication model. In line 5 of Figure 6.4(a), the consumer access the data produced by the `producer` in reverse order. The result is that the `consumer` process has to postpone execution, until the last element is produced by the `producer`. In this case, the `consumer` has to wait 5 iterations before it starts executing. The derived PPN is shown in Figure 6.4(b). The communication pattern between `producer` and `consumer` is correctly identified as OOM-.

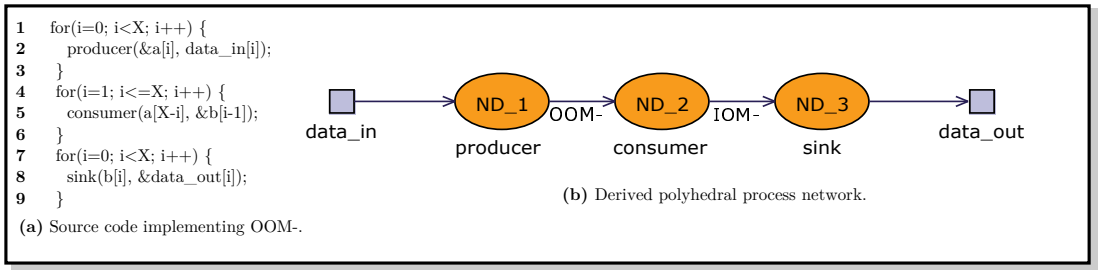


Figure 6.4: Example implementation of the OOM- communication model.

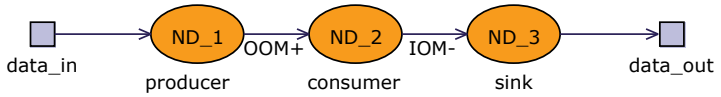
The source code is profiled by cprof, and the waveform shown in Figure F.3(a) of Appendix F is generated. After RTL simulation, Xilinx ISim generates the waveform shown in Figure F.3(b) of Appendix F. Cprof estimates that it takes 230 ns to finish execution, and the RTL simulation is finished after 230 ns. The two waveforms are identical.

6.2.4 Out-of-Order with Multiplicity (OOM+)

In Figure 6.5(a), example source code is shown for implementing the OOM+ communication model shown in Figure 6.1(d). In line 6 of Figure 6.5(a), the consumer is enclosed by two loops, and accesses the data produced by the `producer` in reverse order. The result is that the `consumer` process has to postpone execution, until the last element is produced by the `producer`. In this case, the `consumer` has to wait 5 iterations before it can actually start executing. Multiplicity is present, because a token produced is consumed in multiple iterations. For example, `a[0]` is consumed 5 times in last iteration of the consumer node. The derived PPN is shown in Figure 6.5(b). The communication pattern between `producer` and `consumer` is correctly identified as OOM+.

```
1   for(i=0; i<X; i++) {
2     producer(&a[i], data_in[i]);
3   }
4   for(i=1; i<X; i++) {
5    for(j=1; j<=i; j++) {
6     consumer(a[X-i], &b[i-1]);
7    }
8   }
9   for(i=0; i<X; i++) {
10    sink(b[i], &data_out[i]);
11  }
```

(a) Source code implementing IOM+.



(b) Derived polyhedral process network.

Figure 6.5: Example implementation of the OOM+ communication model.

The source code is profiled by cprof, and the waveform shown in Figure F.4(a) of Appendix F is generated. After RTL simulation, Xilinx ISim generates the waveform shown in Figure F.4(b) of Appendix F. Cprof estimates that it takes 330 ns to finish execution, and the RTL simulation is finished after 330 ns. The two waveforms are identical.

6.2.5 Results

Cprof showed that it is capable of correctly profiling the four different communication models used in PPNs. In Figure 6.6, the estimated execution finish times by cprof and Xilinx ISim are shown. The only difference in the execution finish times is found in the IOM- and IOM+ communication models. In Figure F.1(b) of Appendix F, we see that in the RTL simulation of the IOM- program the read operation is delayed at clock cycle 4 and 5. This delay is introduced in the simulation of the IOM+ program as well, as shown in Figure F.2(b) of Appendix F. Given the nature of the communication models, we conclude that this is undefined behavior in the RTL generated by Compaan DDE. The estimates of the execution finish times of the OOM- and OOM+ models are exactly the same as the execution finish time of the RTL simulations.

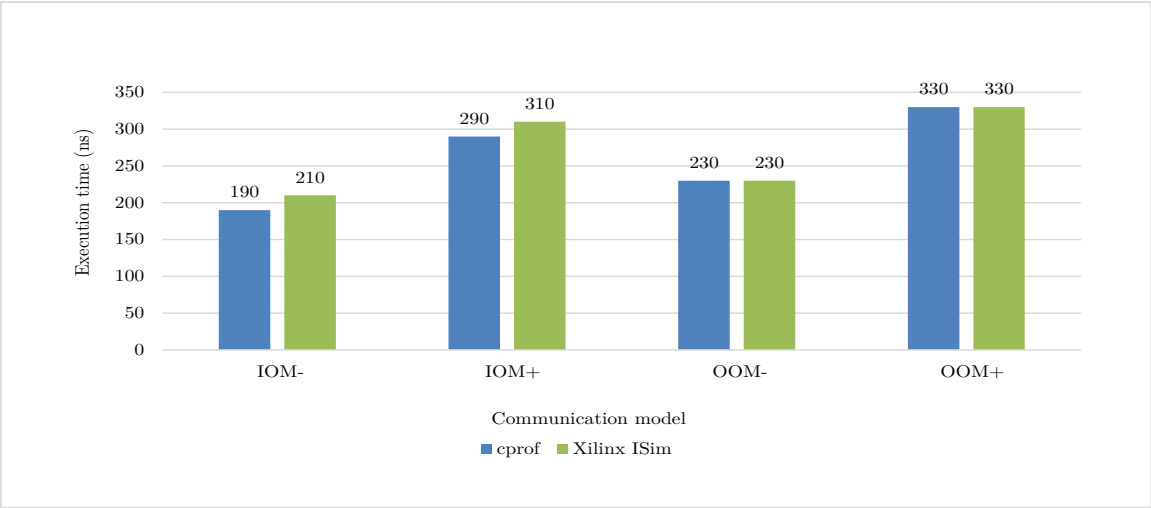


Figure 6.6: Execution times of the communication models measured by cprof and ISim.

6.3 Verification of the Absolute and Unbounded Throughput Estimates

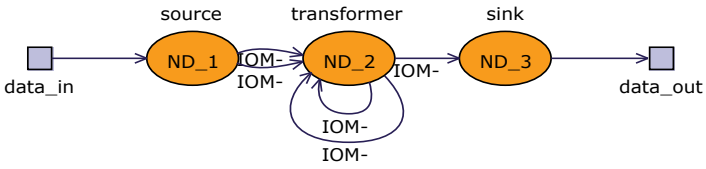
In this section, we will show that it is possible to transform a program to achieve the performance measured by the unbounded throughput estimate. Program transformations are used to optimize the program, and the results are discussed in this section. We use the `predictor` program throughout this section, as it is resentative for a large class of scientific applications [41]. In Section 6.3.1, the predictor program is explained in more detail. The optimizations used to increase throughput in predictor are explained in Section 6.3.2. Finally, in Section 6.3.3, the results are discussed.

6.3.1 The Predictor Program

The predictor program is a program with complex data dependencies. The source code of the program is shown in Figure 6.7(a), and the derived PPN in 6.7(b).

```
1   for (i = 0; i <= 4; i = i + 1) {
2     for(j = 0; j <= 4; j = j + 1) {
3     source(&a[i][j], data_in[i][j]);
4     }
5   }
6   for (i = 1; i <= 4; i = i + 1) {
7     for(j = 1; j <= 4; j = j + 1) {
8     transformer(&a[i][j], a[i-1][j], a[i][j-1]);
9     }
10  }
11  for (i = 1; i <= 4; i = i + 1) {
12    for(j = 1; j <= 4; j = j + 1) {
13     sink(a[i][j], &data_out[i][j]);
14    }
15  }
```

(a) Source code of predictor.



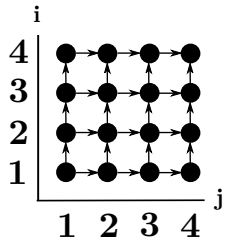
(b) Derived polyhedral process network.

Figure 6.7: The source code and derived PPN of predictor.

There are two self-links in `transformer`, as the value of `a[i][j]` is dependent on `a[i-1][j]` and on `a[i][j-1]`. In Figure 6.8(b), the dependency graph of the function `transformer` is shown. The dependency graph shows the data flow dependencies between operations.

```
1   for(i = 1; i <= 4; i = i + 1)
2     for(j = 1; j <= 4; j = j + 1)
3       transformer(&a[i][j], a[i - 1][j], a[i][j - 1])
```

(a) Transformer function.



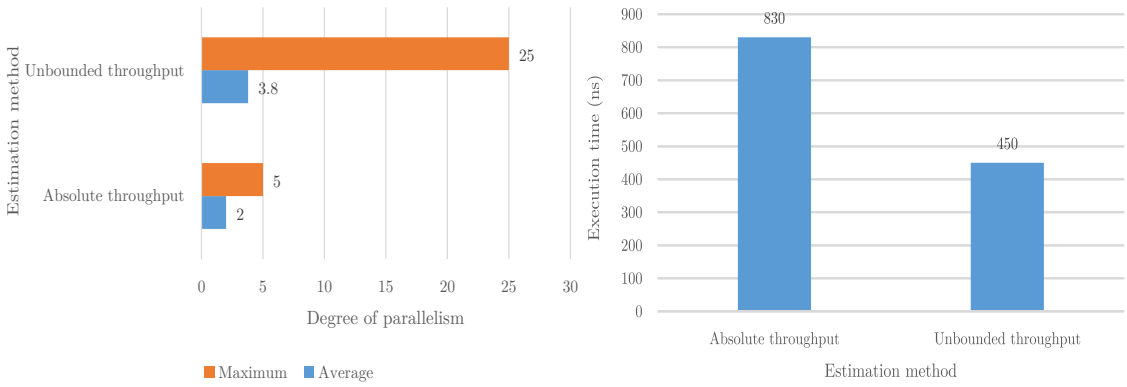
(b) Dependency graph of transformer.

Figure 6.8: The dependency graph of the `transformer` function.

Function	Used slots	Available slots	Pipeline utilization (%)
source	75	75	100.0
transformer	48	192	25.0
sink	48	192	25.0

Table 6.1: Pipeline efficiency of the predictor.

Before we start optimizing the predictor, we need to know the absolute and unbounded throughput estimates. The measurements are shown in Figure 6.9.



(a) The average and maximum degree of parallelism of predictor.

(b) The estimated execution time of predictor.

Figure 6.9: The absolute and unbounded throughput estimates for the predictor.

The average and maximum degree of parallelism, found with the absolute throughput estimate, are 2 and 5. The numbers found with the unbounded throughput estimate are 3.8 and 25. This means that the average degree of parallelism can be increased by 90.0%, and the maximum degree of parallelism by 500%. The execution finish time of the original predictor can be reduced with a maximum of 45.7%.

Cprof measures the pipeline utilization based on the function latency and the initiation interval. The pipeline utilization provides a first-order hint for optimizations. In this case, the efficiency of `source` is 100%. The other two functions, `transformer` and `sink`, have an utilization of 25%. The

pipeline utilization of `transformer` suggests that there is room for improvement.

6.3.2 Optimization of Predictor

In the previous subsection, we determined the absolute and unbounded throughput of the predictor program. There is room for improvement, as the numbers for the absolute and unbounded throughput are not the same, and the pipeline utilization shows that the functions are not 100% efficient.

First iteration		
W[i, j]	**R[i - 1, j]**	**R[i, j - 1]**
1, 1	0, 1	1, 0
1, 2	0, 2	1, 1
1, 3	0, 3	1, 2
1, 4	0, 4	1, 3

Second iteration		
W[i, j]	**R[i - 1, j]**	**R[i, j - 1]**
2, 1	1, 1	2, 0
2, 2	1, 2	2, 1
2, 3	1, 3	2, 2
2, 4	1, 4	2, 3

Third iteration		
W[i, j]	**R[i - 1, j]**	**R[i, j - 1]**
3, 1	2, 1	3, 0
3, 2	2, 2	3, 1
3, 3	2, 3	3, 2
3, 4	2, 4	3, 3

Fourth iteration		
W[i, j]	**R[i - 1, j]**	**R[i, j - 1]**
4, 1	3, 1	4, 0
4, 2	3, 2	4, 1
4, 3	3, 3	4, 2
4, 4	3, 4	4, 3

Figure 6.10: Iteration dependencies in the predictor.

This means that there is data parallelism available in the predictor. We know that the functions `source` and `sink` are not suited for optimization, as we assume that there is no data parallelism at the input and output of the predictor. However, in this section the complete predictor will be optimized to show that it is possible to transform the program to achieve maximum performance.

The function `transformer` has room for optimization, as it is possible to exploit data parallelism. The data dependencies in the iterations of `transformer` are shown in Figure 6.10. At time $t = 0$, only operation [1, 1] can be executed. When $t = 1$, it is possible to simultaneously execute operation [1, 2] and [2, 1]. At $t = 3$, it is possible to execute [1, 3], [2, 2] and [3, 1] in parallel. This means that it should be possible to optimize `transformer` to increase the throughput.

To increase throughput, it is necessary to transform the source code to exploit the available data-parallelism. In this case, it is possible to unroll the outer and inner loops, as both have the same execution profile. That is, the data dependencies in predictor are horizontal and vertical. Unrolling both will not give any performance advantages. In Figure 6.11, two possible optimizations of the predictor are shown. The inner loop is fully unrolled in Figure 6.11(a). The outer loop is fully unrolled in Figure 6.11(b).

```
1  for(i = 1; i <= 4; i = i + 1) {
2      transformer(&a[i, 1],a [i - 1, 1], a[i, 1 - 1]) // st0
3      transformer(&a[i, 2], a[i - 1, 2], a[i, 2 - 1]) // st1
4      transformer(&a[i, 3], a[i - 1, 3], a[i, 3 - 1]) // st2
5      transformer(&a[i, 4], a[i - 1, 4], a[i, 4 - 1]) // st3
6  }
```

(a) Inner loop unrolled.

```
1  for(j = 1; j <= 4; j = j + 1)
2      transformer(&a[1, j], a[1 - 1, j], a[1, j - 1]) //st0
3  for(j = 1; j <= 4; j = j + 1)
4      transformer(&a[2, j], a[2 - 1, j], a[2, j - 1]) // st1
5  for(j = 1; j <= 4; j = j + 1)
6      transformer(&a[3, j],a[3 - 1, j], a[3, j - 1]) // st2
7  for(j = 1; j <= 4; j = j + 1)
8      transformer(&a[4, j],a [4 - 1, j], a[4, j - 1]) // st3
```

(b) Outer loop unrolled.

Figure 6.11: Two possible optimizations of predictor.

After unrolling the inner loop, the data dependencies still exist. Each statement in the unrolled loops are mapped onto their own dedicated process. Now, at time $t = 3$, it is possible to execute operation [1, 3], [2, 2] and [3, 1] in parallel, as the processes start executing as soon as input data is available. The same applies to unrolling the outer loop. In line 2 of Figure 6.11(b), the operations are

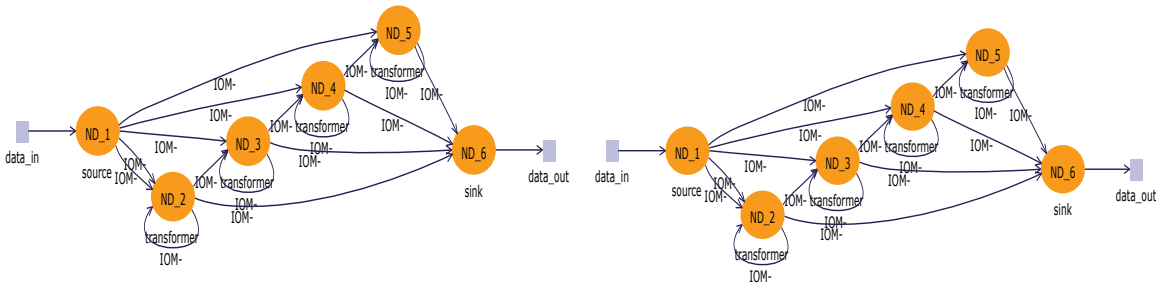
Function	Used slots	Available slots	Pipeline utilization (%)
source	75	75	100.0
transformer	12	48	25.0
transformer	12	48	25.0
transformer	12	48	25.0
transformer	12	48	25.0
sink	48	93	51.6

Table 6.2: Pipeline efficiency of the predictor after unrolling the inner or outer loop.

executed sequentially within that process. That is, [1, 2] is dependent on [1, 1]. Now, let us consider the statement in line 5. Operation [2, 1] execute if and only if operation [1, 1] has finished. As soon as this happens it can start executing, while at that same time the statement in line 1 is executing operation [1, 2].

Up to this point, we have discussed only the optimization of the `transformer` function. However, the functions `source` and `sink` can be unrolled as well. The unrolling of both `source` and `sink` cannot be verified in hardware, as we assume that the input and output streams are sequential. The predictor program will be optimized in six different ways. The source code of the optimized versions of the predictor are listed in Appendix C. Three out of six of the optimized designs can be implemented in hardware. The other three optimize either the input/output streams or both and, as we assume that the input and output streams are sequential by nature, no RTL can be generated for such designs.

Figure 6.12(a) shows the derived PPN if the inner loop is fully unrolled. In this case, the `transformer` function is replicated four times. Figure 6.12(b) shows the PPN of the predictor when the outer loop is fully unrolled. The figures are exactly the same, as expected, as there is no difference in unrolling the inner or outer loop. The reason is that the orientation of data dependencies in predictor is the same in both versions.



(a) Derived process network after unrolling the inner loop.

(b) Derived process network after unrolling the outer loop.

Figure 6.12: Polyhedral process network of the predictor with the inner and outer loop unrolled.

This effect is also noticeable in cprof, since the pipeline utilization for both versions is exactly the same, as shown in Table 6.2:

Each `transformer` function still operates at 25%. However, because each transformer starts executing as soon as input data is available, the pipeline utilization of `sink` is increased. The result is that the throughput of `sink` is increased, and the execution time of the predictor is reduced.

If the inner and outer loop of the predictor are both fully unrolled, the result is that `transformer` is replicated 16 times. Now, each `transformer` is responsible for one iteration, and the pipeline

utilization is 100%. However, the `sink` function has a pipeline utilization of 51.6%. Therefore, for maximum performance, it will suffice to either unroll the inner or outer loop, with a reduced number of resources. Despite the complexity of the network, the execution time is not reduced. The reason is that the data dependencies in the predictor do not allow for further optimizations. The complexity of this process network is considerable, and generating the RTL of the design takes almost 5 minutes. However, estimating the performance in cprof takes less than 5 seconds.

6.3.3 Results

The absolute and unbounded throughput estimates deliver a lower and upper bound of the design space in terms of parallelism. After optimizing the source code, it should be possible to achieve performance found by the unbounded throughput estimate. In the previous section, six optimized versions of the predictor were presented. The performance of the predictor program is measured using both the absolute and unbounded throughput estimate. In Figure 6.13, cprof estimates that the execution time for the unmodified program is 830 ns, whereas in hardware it is 850 ns. The fully unrolled predictor has an execution time of 450 ns.

In theory, it is possible to reduce the execution time of the predictor with a factor of 1.8. However, we know that it is impossible to fully unroll the program, given the restrictions of the `source` and `sink` node, as discussed in Section 6.3.2. Therefore, the optimized predictors implementable in hardware optimize either the inner or outer loop or both. The estimated execution finish times by cprof for the three versions implementable in hardware are the same, for previously discussed reasons. The estimated execution time is 500 ns, and the implementations in hardware finish after 520 cycles. The reduction in the execution time is almost 1.7x. Cprof has overestimated the performance in this case by 3.8%. The theoretical minimum execution finish time is shown in Figure 6.9(b), and is found to be 450 ns. In the last two optimization methods, shown in Figure 6.13, this number is achieved. The maximum performance achievable in hardware takes about 1.2 times more than the maximum speed.

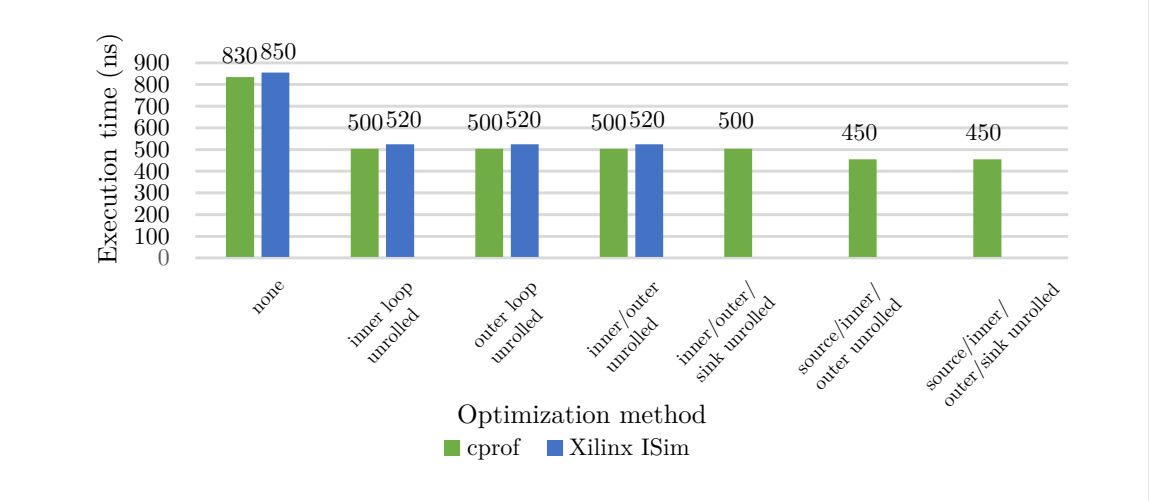


Figure 6.13: Execution times of the predictor measured by cprof and by Xilinx ISim.

In Table 6.3, the resource costs associated with the RTL implementations of predictor are listed. Optimization means that more resources are needed to implement the design in hardware. The predictors with either the inner or the outer loop optimized have the same resource usage. The reason is that the optimized networks implement the same behavior. The optimized versions of the predictor have a substantial increase in the usage of lookup tables (LUT) and flip-flops (FF). There is a minor growth of block random access memory (BRAM). The fully unrolled version is the most expensive in terms of resources, as each iteration of a statement is mapped onto its own dedicated processing

Optimization method	LUT	FF	BRAM
none	327	135	1
inner loop unrolled	987	362	3
outer loop unrolled	987	362	3
inner/outer loop unrolled	2520	736	4

Table 6.3: Resource cost of the predictor after unrolling the inner or outer loop.

resource.

In Figure 6.14(a), the average degree of parallelism associated with each version of the predictor is shown. The lowest average degree of parallelism is found in the original predictor. No optimizations are applied, and the average degree of parallelism is 2. In the graph the highest average degree of parallelism is achieved if the source, inner and outer loop are fully unrolled or if the program is fully unrolled. This number, 3.8, is the same as the average degree of parallelism found for the unbounded throughput estimate in Figure 6.9(a). In Figure 6.14(b), the maximum degree of parallelism associated with each version of the predictor is shown. The lower bound on the maximum degree of parallelism is 5, and is found in the unmodified version of the predictor. The upper bound of the maximum degree of parallelism is 25.

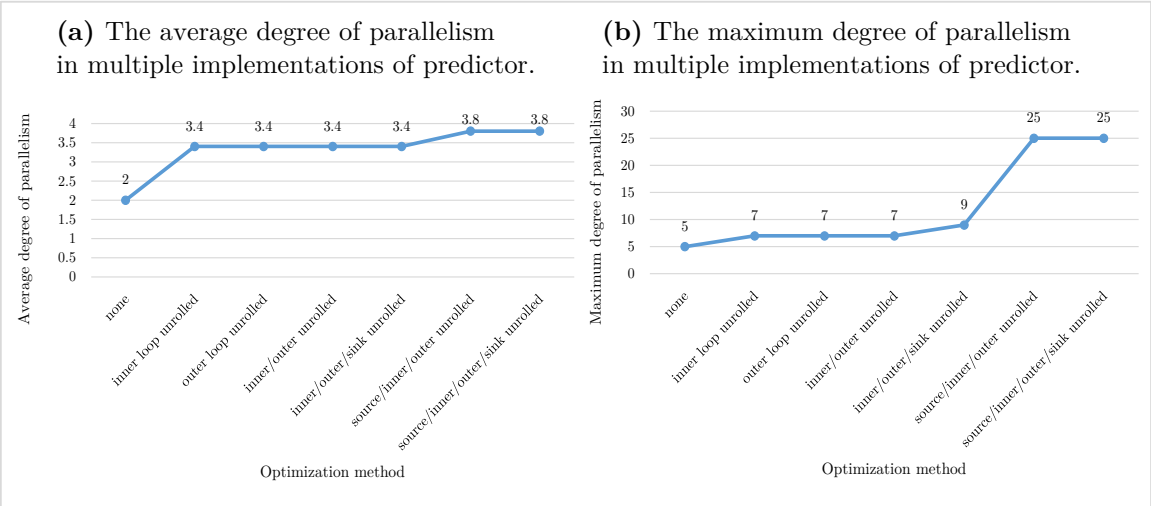


Figure 6.14: The average and maximum degree of parallelism available in the predictor.

In this section, we have shown that it is possible to move from the absolute throughput estimate to the performance found by the unbounded throughput estimate. With this information, we can limit design space exploration, as shown in Figure 1.1.

6.4 Verification of Hierarchy Program Analysis

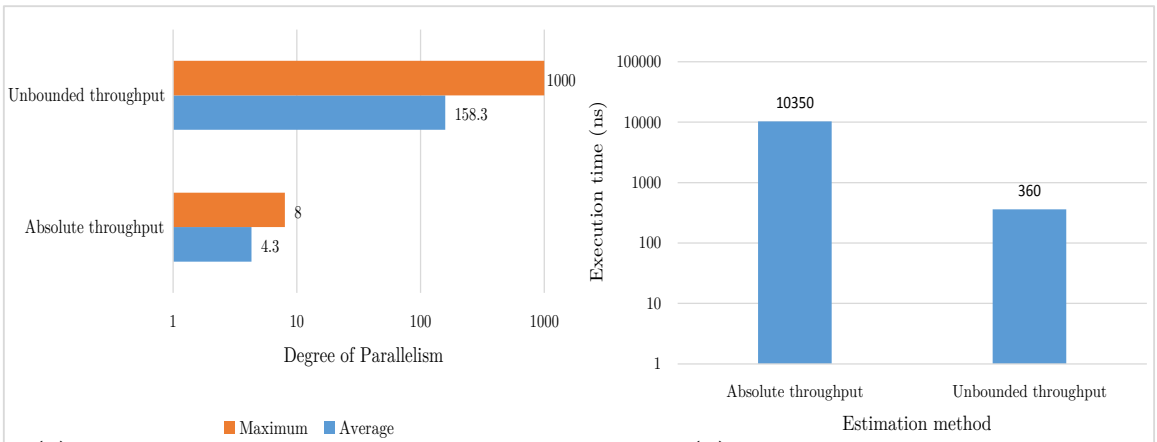
In this section, we show that cprof is capable of profiling programs with hierarchy. In Section 6.4.1, we introduce the hierarchy program used for verification. The verification results are discussed in Section 6.4.2.

6.4.1 The Hierarchy Program

In Figure 5.14 (see Section 5.7.1), we show the program used for the verification of HPA. The program is complicated enough to demonstrate HPA, as the program uses inter-procedural function calls. Cprof automatically identifies the following five functions for profiling: `foo`, `bar`, `transformer`, `hierarchy`, and `main`. The five functions identified describe the inter-procedural behavior in the program. We assume that the function latency (Λ_F) of each function is 2 cycles. The `MAX_O` and `MAX_HALF` definitions used for sizing the arrays, are set to 1000 and 500, respectively.

6.4.2 Results

We first estimate the absolute and unbounded throughput of the hierarchy program. The unbounded throughput estimates that the execution finish time is 360 ns, whereas the absolute throughput estimate is 10350 ns. This means that the execution finish time is reduced by 95.5%.



(a) The average and maximum degree of parallelism of hierarchy.

(b) The estimated execution time of hierarchy.

Figure 6.15: The absolute and unbounded throughput estimates for hierarchy.

In Figure 6.16, the execution times estimated by cprof and measured with ISim are shown. Cprof estimates the performance of the hierarchy with high accuracy, because the execution finish time is only overestimated by 0.34%.

Figure 6.16: The execution finish times of hierarchy.

6.5 Summary and Conclusions

In this chapter, we have shown that cprof supports the necessary communication models, and that the estimated performance is verifiable with RTL simulations. We showed that it is possible to optimize a program to achieve the maximum performance, given by the unbounded throughput estimate. Hierarchical program analysis is used for programs with complex inter-procedural relations, and cprof has the ability to profile such programs as well. In this chapter, we have verified that it is possible to use cprof to estimate the performance of C programs implemented in hardware as polyhedral process networks. In the next chapter, we show that cprof is capable of profiling complicated mathematical benchmarks.

7 Results

In this chapter, we use the PolyBench/C 3.1 benchmarks to show that cprof is capable of profiling benchmarks, which consist of complicated mathematical functions. The control parts of PolyBench/C 3.1 benchmarks are static and are, therefore, they are well suited for implementation as PPNs in hardware. In Section 7.1, the experimental setup is explained. The absolute throughput estimates and the unbounded throughput estimates are presented in Section 7.2 and Section 7.3, respectively. In Section 7.4, the measurements of the RTL implementations of PolyBench/C benchmarks are presented. The PolyBench/C benchmarks design boundaries are presented in Section 7.5. In Section 7.6, we show how cprof is capable of optimizing one of the benchmarks. The scalability of cprof is discussed in Section 7.7. A summary is given and conclusions are drawn in Section 7.8.

7.1 Experimental Setup

One of the goals of this work is to validate cprof against hardware implementations of PolyBench/C 3.1 benchmarks [6]. For this purpose, we use RTL implementations of PolyBench/C 3.1, generated by Compaan DDE. In Appendix D, a short description of each benchmark in PolyBench/C is given. The "**C**" in PolyBench/C 3.1 refers to the fact that the benchmarks are written in the C programming language. Each PolyBench/C 3.1 benchmark is configured to use the `MINI_DATASET` size.

Cprof accepts programs that are modeled with function calls. The statements in PolyBench/C benchmarks are not modeled in this way. Compaan DDE automatically creates functions for such statements. However, Compaan DDE does not substitute the statements in the original kernel. To solve this problem, we developed a compiler plugin. The design and implementation of this plugin are described in Appendix A.

The platform used for profiling the benchmarks with cprof is an Intel i7-3520M operating at 2.9 GHz, with 8 GB internal memory. Xilinx ISE Simulator (ISim) is used for the purpose of simulating the RTL designs of the benchmarks, and the FPGA board used is a Virtex-6 FPGA (xc6vlx240t). The clock period is 10 ns and the clock frequency is 100 MHz. The function latency (Λ_F) is 3 cycles, and the initiation interval (II_F) is 1 cycle. The measurements are rounded down, and have one degree of decimal accuracy. The main reason for this degree of accuracy is that it shows if the average degree of parallelism is moving away or towards the nearest integer.

7.2 Absolute Throughput Estimates of PolyBench/C

Each Polybench/C benchmark is profiled with cprof for the absolute throughput estimate. In Section 7.2.1, the estimated execution times of the benchmarks are presented. The average and maximum degree of parallelism in the benchmarks are given in Section 7.3.2.

7.2.1 Execution Times

Cprof estimates the execution finish times of the C programs when implemented as a PPN in hardware. The results are shown in Figure 7.1. The Y-axis is in $\log_{10}$ scale. Instead of multiple invocations of the same kernel, the `mm3`, `mvt`, `syr2k`, and `syrk` are modified to execute the kernel only once.

The computational complexity of the benchmarks varies greatly, and this directly affects the estimated execution times. For example, the `jacobi_1d_imper` benchmark consists of 5 for loops, and

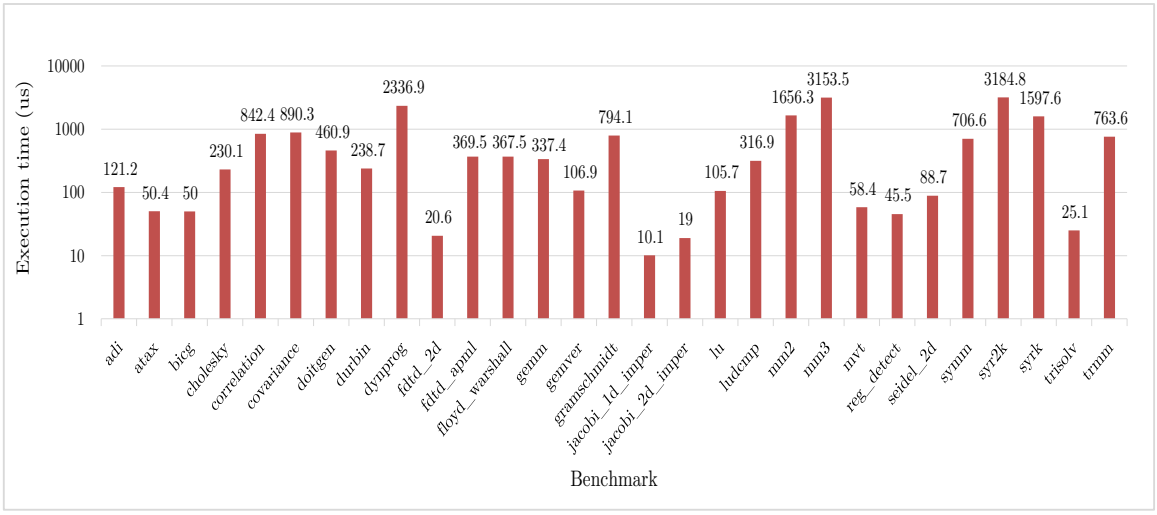


Figure 7.1: Absolute throughput estimates of the execution finish times of PolyBench/C kernels.

only one multiplication, and a few additions, whereas `syr2k` has 13 loops and has 5 multiplications. On average, the execution time of a benchmark is 653.4 µs.

7.2.2 The Average and Maximum Degree of Parallelism

The average and maximum degree of parallelism found by the absolute throughput estimate gives insight about the parallel behavior of process networks. The results are shown in Figure 7.2.

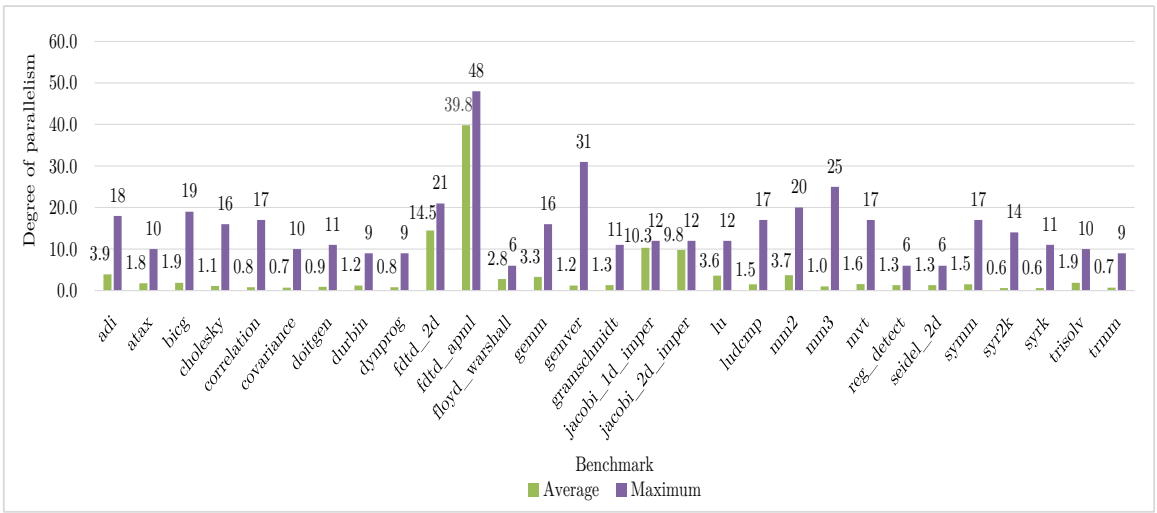


Figure 7.2: Absolute throughput estimates of the average and maximum degree of parallelism in PolyBench/C kernels.

The average degree of parallelism is quite low in all benchmarks. The average degree of parallelism over all benchmarks given by the absolute throughput estimate is 4. Two noticeable exceptions are the `fdtd_2d` and `fdtd_apml` benchmarks. In the source code of the two benchmarks, multiple statements are working on different data sets, i.e., the data dependencies between statements is quite low, and

the result is a high average degree of parallelism. The maximum degree of parallelism is similar for all the benchmarks. The maximum degree of parallelism represents the peak parallel performance. The maximum degree of parallelism on average given by the absolute throughput estimate is 15.2. For example, in the `gemver` benchmark the maximum degree of parallelism is 31. This means that at one point in time, 31 operations are executing in parallel.

7.3 Unbounded Throughput Estimates of Polybench/C

Each Polybench/C benchmark is profiled with cprof for the unbounded throughput estimate. In Section 7.3.1, the estimated execution times of the benchmarks are presented. The average and maximum degree of parallelism in the benchmarks are given in Section 7.3.2.

7.3.1 Execution Times

Cprof estimates the theoretical execution finish times of the C programs using the unbounded throughput estimate. The results are shown in Figure 7.3. The Y-axis is in $\log_{10}$ scale.

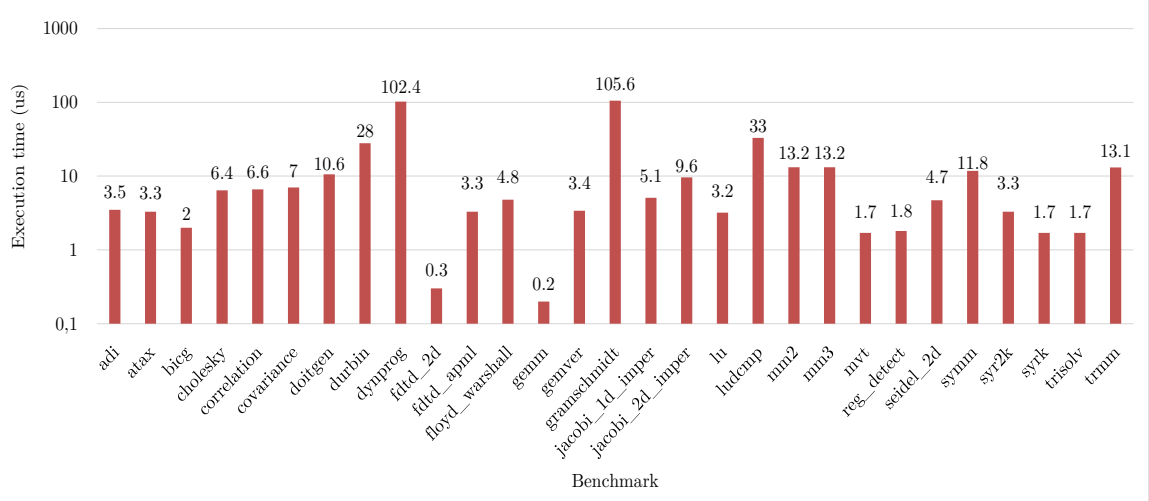


Figure 7.3: Unbounded throughput estimates of the execution finish times of PolyBench/C kernels.

The average execution time of a benchmark found by the unbounded throughput is 13.9 µs. Because each iteration of a statement is mapped onto its own processing resource, the execution times of the benchmarks are relatively low compared to the absolute throughput estimates.

7.3.2 The Average and Maximum Degree of Parallelism

The average and maximum degree of parallelism found by the unbounded throughput estimate gives insight to the theoretical maximum parallel performance of PPNs. The results are shown in Figure 7.4. The Y-axis is in $\log_{10}$ scale.

The average degree of parallelism given by the unbounded throughput estimate is 504.9. On average, the maximum degree of parallelism given by the unbounded throughput estimate is 10989. These percentages are theoretical estimates, i.e., cprof assumes that all statements can be parallelized, as long as data dependencies are not violated.

Figure 7.4: Unbounded throughput estimates of the average and maximum degree of parallelism in PolyBench/C kernels.

7.4 RTL Simulations

In the previous sections, the absolute and unbounded throughput estimates of the PolyBench/C benchmarks were presented. In this section, 9 of the PolyBench/C benchmarks are represented in the RTL specification. The results are shown in Figure 7.5. The Y-axis is in $\log_{10}$ scale.

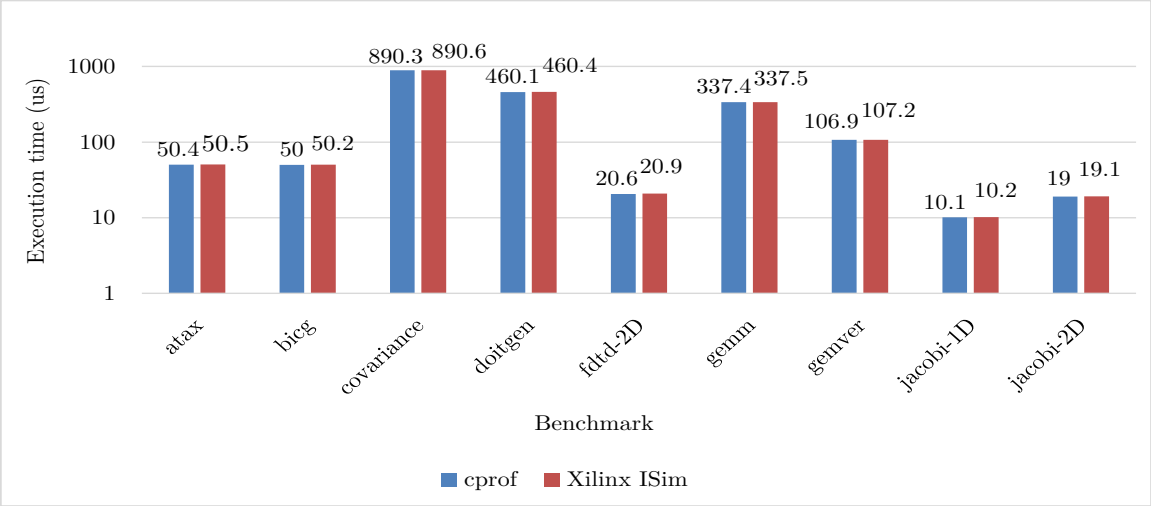


Figure 7.5: Absolute throughput estimates of the execution finish time of PolyBench/C kernel versus the execution finish time of RTL implementations.

The absolute throughput estimates by cprof are close to the numbers achieved in RTL simulation. Cprof overestimates the performance, on average by 0.44%. For large systems, the overestimates increase, as in the simulated designs the FIFOs were large enough to facilitate communication without synchronization problems. That is, cprof assumes FIFOs of unbounded size, and in large designs the FIFO can become full. In this case, the process is blocked until there is enough free space available in the FIFO.

7.5 Design Space Boundaries

The average degree of parallelism estimates by the absolute and unbounded throughput are shown in Figure 7.6. The Y-axis is in $\log_{10}$ scale. These measurements give a lower and upper bound of the design space, in terms of the average degree of parallelism, as presented in Figure 1.1. The average degree of parallelism provides a more realistic design point that is close to the maximum achievable performance [35].

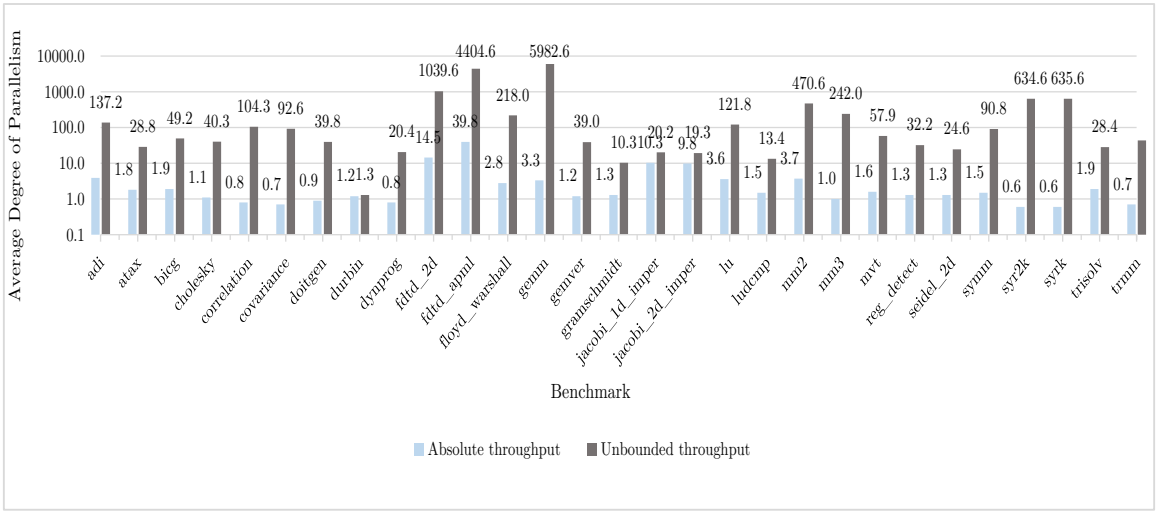


Figure 7.6: Design space boundaries: the average degree of parallelism found by the absolute and unbounded throughput estimates.

The measurements show that the performance of PolyBench/C benchmarks can be significantly improved. In the next section, we show that cprof is capable of evaluating various design points, by optimizing the `atax` benchmark.

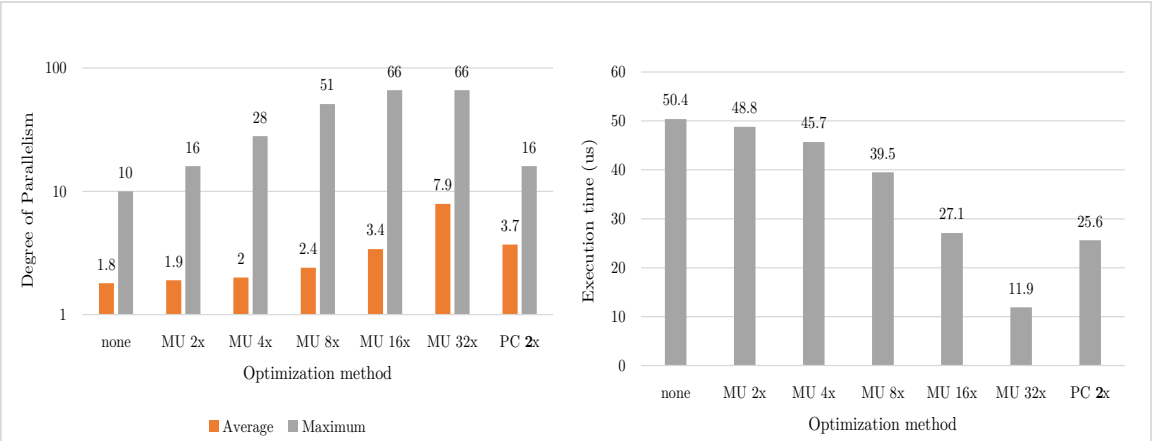
7.6 Optimization

In this section, cprof is used to apply program optimizations to increase the performance of the `atax` benchmark. We assume that the input and output streams of the `atax` benchmark are not parallelizable. All other statements are eligible for optimization. The optimization methods implemented in cprof can only be applied to optimize inner loops, and plane cutting divides iterations over two planes in the iteration domain. Cprof optimizes the inner loops in the source code, and the absolute throughput is subsequently used to determine the performance metrics.

In Figure 7.7(a), the average and maximum degree of parallelism are shown. The unbounded throughput estimate gives an average degree and maximum degree of parallelism of 28.8 and 1088, respectively. If modulo unfolding with a factor 32 is applied to the inner loops of `atax`, the average and maximum degree of parallelism become 7.9 and 66, respectively. The average degree of parallelism is increased by 338.8% and the maximum degree of parallelism with 560%, in comparison to the none-optimized `atax`.

However, we now have design points, which are eligible for implementation in hardware. If the statements are optimized with plane cutting, the average and maximum degree of parallelism are increased by 105.5% and by 60%, respectively. These numbers are considerably lower than the numbers found if `atax` is optimized with modulo unfolding (32x). However, the program optimized with plane cutting consists of 10 processes, whereas the one optimized with modulo unfolding has 130.

The execution finish times of `atax` are shown in Figure 7.7(b). The unbounded throughput estimate



(a) The degree of parallelism of the optimized atax benchmark. **(b)** The execution finish times of the optimized atax benchmarks.

Figure 7.7: Optimization of the `atax` benchmark.

gives, as execution finish time, 3.3 µs, as shown in Figure 7.3. If the inner loops are optimized with modulo unfolding (32x), the execution time is reduced by 76.6%. We have already seen that the number of processes we need in order to achieve this performance is substantial. Plane cutting reduces the execution finish time by 49.2%. The version of `atax` optimized with plane cutting seems a good choice. There is a considerable increase in performance, and the number of processes needed to achieve this performance is only doubled in comparison to `atax` without optimizations. Furthermore, we were able to estimate the performance without actually implementing the design in hardware.

7.7 Scalability

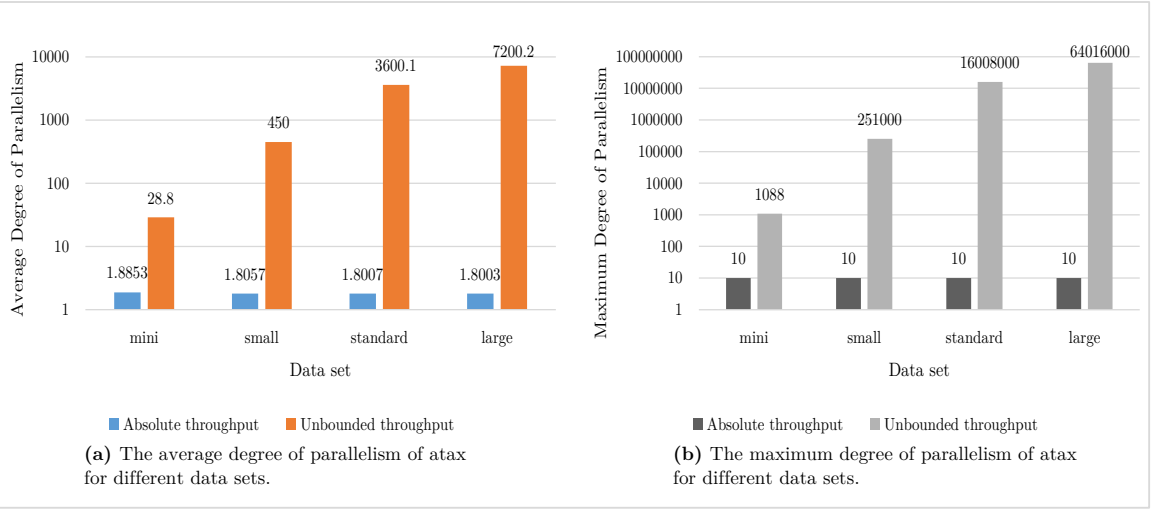
In the previous section, we used the `MINI_DATASET` for all measurements. To show the scalability of cprof, the `atax` benchmark is profiled with larger data sets: `SMALL_DATASET`, `STANDARD_DATASET`, and the `LARGE_DATASET`. In Table 7.1, the data sets are specified. The accuracy of the average degree of

Data set	**NX**	**NY**
`MINI_DATASET`	32	32
`SMALL_DATASET`	500	500
`STANDARD_DATASET`	4000	4000
`LARGE_DATASET`	8000	8000

Table 7.1: PolyBench/C data set specifications.

parallelism is 4 decimals, in order to show the difference between the measurements. In Figure 7.8, the results are shown. The Y-axis is in $\log_{10}$ scale.

Figure 7.8(a) shows that the average degree of parallelism found by the absolute throughput estimate is in the range of 1.8, whereas the maximum degree of parallelism varies greatly for both the absolute and the unbounded estimate. In Figure 7.8(b), the maximum degree of parallelism of `atax` for each data set is shown. The execution finish times for the different data sets are shown in Figure 7.9. The Y-axis is in $\log_{10}$ scale. The execution time of `atax` is influenced by the data size. As a result, the execution times vary greatly.



(a) The average degree of parallelism of atax for different data sets.

(b) The maximum degree of parallelism of atax for different data sets.

Figure 7.8: Estimates of the average and maximum degree of parallelism of the atax benchmark for various data sets.

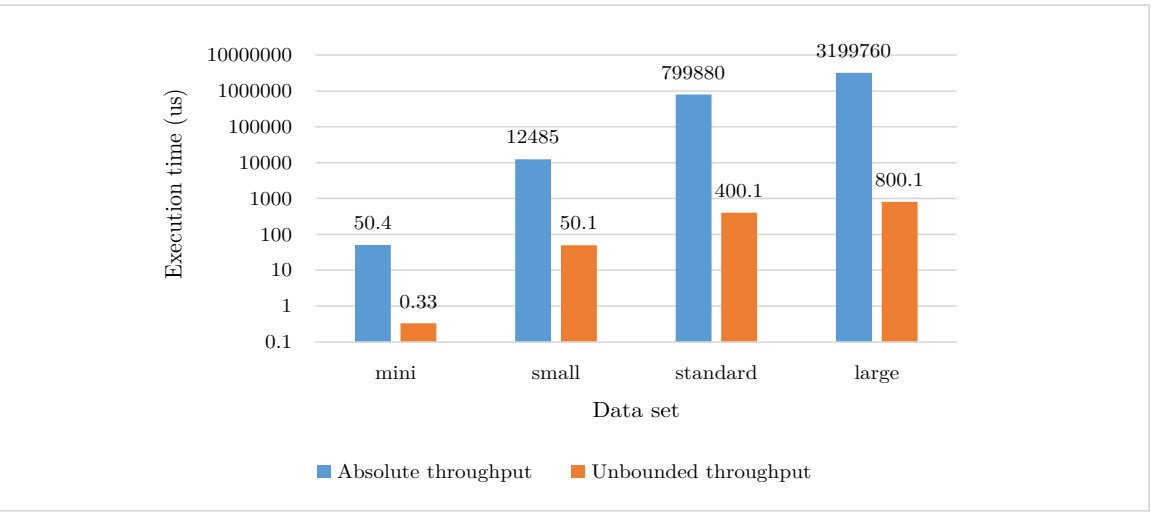


Figure 7.9: Estimates of the execution time of the atax benchmark for various data sets.

Profiling of `atax` varies from a few seconds for the `MINI_DATASET` to about 10 minutes for the `LARGE_DATASET`. If we use Compaan DDE to generate a design of `atax` for both data sets, the time necessary for the `MINI_DATASET` is a few minutes, whereas the same procedure takes more than an hour for the `LARGE_DATASET`. The differences are shown in Table 7.2, and estimating the performance with cprof is significantly faster than going through the complete design flow of Compaan DDE.

Data set	Cprof (hh:mm:ss)	Compaan DDE (hh:mm:ss)
MINI_DATASET	00:00:05	00:00:45
SMALL_DATASET	00:00:09	00:11:15
STANDARD_DATASET	00:04:30	00:31:27
LARGE_DATASET	00:13:02	01:55:33

Table 7.2: Time spent by cprof on estimating the performance of atax, and by Compaan DDE to generate a hardware implementation of atax.

7.8 Summary and Conclusions

In this chapter, we have shown that we can estimate the execution finish times of PolyBench/C benchmarks correctly. The performance of RTL simulations is close to the performance estimated by cprof, the average overestimation is 0.44%. We can obtain bounds on the design space, thereby giving insight into the performance potential of the PolyBench/C benchmarks. We have demonstrated the capabilities of cprof with respect to program optimization. We have also shown that cprof scales well with large data sets, by profiling, as an example, the `atax` benchmark for various data sets.

8 Conclusions and Future Work

In this thesis, we presented cprof, a tool for the profiling of polyhedral process networks. Cprof provides the possibility to estimate the performance of designs, specified in C or C++, early in the design flow. We continued the work of Van Haastregt [2], as none of the other profilers are capable of profiling PPNs. Cprof instruments source code to estimate the performance of sequential C or C++ code, when implemented as a PPN in hardware. We have used the LLVM/Clang compiler infrastructure to automatically generate the instrumented program. The instrumented source code is subsequently used for profiling.

Two modes of performance estimation are implemented: the absolute and unbounded throughput estimates. Absolute throughput assumes that all iterations of a statement are mapped onto the same processing resource. Unbounded throughput assumes that the execution of an iteration of a statement is mapped onto its own dedicated processing resource.

The performance metrics used in the absolute and unbounded throughput estimate are as follows: the average degree of parallelism, the maximum degree of parallelism, and the execution finish time. The average degree of parallelism represents a design point close to the maximum achievable performance [35]. The maximum degree of parallelism represents the peak parallel performance in a program. The average degree of parallelism found in the absolute and unbounded throughput estimate, give a possible lower and upper bound of the design space. Cprof evaluates design points between these two extremes, by applying optimization techniques. The execution finish time is the time required by a PPN to complete its execution. The performance we determine in cprof relates to the run-time of a given application in hardware, and resource usage is not modeled.

We verified that cprof supports the four different communication models used in PPNs. Hierarchical program analysis is used to profile programs with inter-procedural relations. The PolyBench/C benchmarks were profiled, and we showed that on average, cprof overestimates the execution finish time of the PolyBench/C benchmarks implemented in hardware by 0.44%. Cprof helps increasing engineering productivity by assisting in DSE, and risk is reduced by making design limitations explicit at an early stage in the design process. The result is that the hardware design flow looks like a regular software design flow, and no special hardware skills are required to analyze and optimize a design that is eventually implemented as a PPN in hardware.

8.1 Contributions

The main contributions of this thesis can be summarized as follows.

1. **A profiler capable of profiling polyhedral process networks.** The profiler, referred to as cprof, is capable of profiling sequential C and C++ programs, which are realized in hardware as PPNs. The performance of a program is characterized by the absolute and unbounded throughput estimates, which give a lower and upper bound of the design space, respectively;

2. **Hierarchical Program Analysis for estimating the performance of programs with inter-procedural behavior.** We have introduced HPA in cprof. As a result, we can profile programs with inter-procedural behavior, which are specified in the C or C++ programming language. HPA allows us to estimate the performance of complex and real-world designs;

3. **Assistance in Design Space Exploration.** Cprof applies source code optimization to generate design points between the minimum and maximum achievable performance. The result is that designers can evaluate design points before actually implementing the design in hardware;

4. **Verification and results.** We verified the solution approach implemented in cprof, and we validated cprof against RTL implementations of the PolyBench/C benchmarks. A compiler plugin for Compaan was developed. With this plugin, we translate statements unsupported by cprof to their supported equivalents. We have also shown that cprof scales well with large data sets.

8.2 Future Work

In this section, we present several recommendations for future work.

Extension of Programming Language Support

Cprof only supports statements modeled as a function call. Supporting statements not modeled as function calls allows to profile a larger class of programs. SystemC is an extension of the C++ programming language, and cprof could be extended to support the SystemC classes and take advantage of the hardware descriptions for better performance estimates.

Modeling of Hardware Resources

Cprof assumes a one-to-one mapping of a statement onto a process. Cprof has no knowledge about the hardware resources used by an IP block. One way to model hardware resources is by introducing the concept of *resource variables*, denoted as `R$s`. If the system is modeled with N resources, it is possible to claim a resource with the `R$s` variable. All the statements use this variable to determine whether a hardware resource is available. If the resource is unavailable, the execution of the statement is delayed until the resource is available. The use of a resource variable would allow us to model resource contention.

Resource Cost Estimation

The execution finish time is one aspect of the performance of PPNs. Another aspect is the hardware resource cost. One way to estimate resource usage of processes is to specify the resource cost in a configuration file. Another approach is to apply statistical estimation to predict the resource cost, similar to the Quipu approach [22]. Implementing resource cost estimation in cprof provides insight to the feasibility of design points generated by cprof, in terms of performance and resource cost.

Limiting the Unbounded Throughput Estimate

If we estimate the unbounded throughput of a program, we assume that the program is fully unrolled. Designers often know which functions cannot be parallelized. If it is possible to mark those kernels, the predicted performance can become more accurate.

Dynamic Performance Modeling

Cprof assumes a fixed latency for each IP block. Most IP blocks have different latencies, depending on the inputs. For example, a given multiplier normally takes 8 cycles to finish. If one of the inputs is a zero, the result is delivered after one cycle. Cprof already supports the dynamic modeling of functions. However, the designer has to manually insert statements to check the input to determine the latency. A better solution is to use a configuration file, and to define for which inputs the function latency varies.

Modeling of Communication Channels

Cprof assumes that FIFOs of unbounded size are used to facilitate communication between processes. For larger designs, this assumption is unrealistic and this affects the found result. Before a process can resume execution, the FIFO must be ready to process data. The derivation of PPNs requires the calculation of the buffer sizes, because communication channels are not explicitly modeled. However, estimating the performance degradation caused by FIFO congestion should be possible, if the communication between processes is explicitly modeled. For example, a `CprofFIFO` object could be used to represent the communication channel between processes.

Extending Optimization

Cprof applies modulo unfolding and plane cutting to explore the performance of programs. Cprof only optimizes the innermost loop, and with plane cutting it is only possible to divide the iterations of a statement over two processes. Another optimization that can be represented in C code is skewing. Further work is necessary to make it possible to optimize all the enclosing loops of a statement.

Support of Control Flow Architectures

PPNs are classified as a dataflow architecture. Estimating the performance of programs that use a control flow architecture requires the modeling of different data hazards. We already provide basic support for control flow architectures (see Appendix E). If it is possible to profile an application on both platforms, a designer has the opportunity to evaluate the performance of an application on multiple platforms, and make design choices accordingly.

Bibliography

[1] J. Hennessy, "The future of systems research," *Computer*, vol. 32, no. 8, pp. 27–33, 1999.

[2] S. J. J. Haastregt, *Estimation and optimization of the performance of polyhedral process networks*. PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), Faculty of Science, Leiden University, 2013.

[3] J. Castrillon, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. PhD thesis, PhD thesis, RWTH Aachen university, 2013.

[4] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: Deriving process networks from matlab for embedded signal processing architectures," in *Proceedings of the eighth international workshop on Hardware/software codesign*, pp. 13–17, ACM, 2000.

[5] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere, "Daedalus: toward composable multimedia mp-soc design," in *Proceedings of the 45th annual Design Automation Conference*, pp. 574–579, ACM, 2008.

[6] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *http://www.cs.ucla. edu/˜ pouchet/software/polybench/*, 2012.

[7] G. Kahn, D. MacQueen, *et al.*, "Coroutines and networks of parallel processes," 1976.

[8] A. Turjan, B. Kienhuis, and E. Deprettere, "Translating affine nested-loop programs to process networks," in *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 220–229, ACM, 2004.

[9] S. Meijer *et al.*, *Transformations for polyhedral process networks*. Leiden Institute of Advanced Computer Science (LIACS), Faculty of Science, Leiden University, 2010.

[10] C. Lattner, "Llvm and clang: Next generation compiler technology," 2008. Poster presented at the BSDCan 2008, Ottawa, Canada.

[11] A. Mycroft and A. Zeller, *Compiler Construction: 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*, vol. 3923. Springer, 2006.

[12] J. R. Larus, "Whole program paths," in *ACM SIGPLAN Notices*, vol. 34, pp. 259–269, ACM, 1999.

[13] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 7–16, IEEE Computer Society, 2004.

[14] S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 39, no. 4, pp. 49–57, 2004.

[15] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Notices*, vol. 42, pp. 89–100, ACM, 2007.

[16] L. Gao, J. Huang, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr, "Totalprof: a fast and accurate retargetable source code profiler," in *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pp. 305–314, ACM, 2009.

[17] S. Rul, H. Vandierendonck, and K. De Bosschere, "A profile-based tool for finding pipeline parallelism in sequential programs," *Parallel Computing*, vol. 36, no. 9, pp. 531–551, 2010.

[18] R. Bell, A. D. Malony, and S. Shende, "Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis," in *Euro-Par 2003 Parallel Processing*, pp. 17–26, Springer, 2003.

[19] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: collection and analysis of parallel block vectors," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 452–463, IEEE Computer Society, 2012.

[20] S. Blair-Chappell and A. Stokes, *Parallel Programming with Intel Parallel Studio XE*. John Wiley & Sons, 2012.

[21] S. A. Ostadzadeh, R. Meeuws, I. Ashraf, C. Galuzzi, and K. Bertels, "Profile-guided application partitioning for heterogeneous reconfigurable platforms," in *Computer Architecture and Digital Systems (CADS), 2012 16th CSI International Symposium on*, pp. 37–43, IEEE, 2012.

[22] R. Meeuws, *Quantitative hardware prediction modeling for hardware/software co-design*. Ph. D. thesis, 2012.

[23] S. A. Ostadzadeh, R. J. Meeuws, C. Galuzzi, and K. Bertels, "Quad–a memory access pattern analyser," in *Reconfigurable computing: architectures, tools and applications*, pp. 269–281, Springer, 2010.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[25] C. Feenstra and B. Eng, *A Memory Access and Operator Usage Profiler Framework for HLS Optimization*. PhD thesis, Masters thesis, Delft University of Technology, 2011.

[26] X. Wu, *Performance evaluation, prediction and visualization of parallel systems*, vol. 4. Springer, 1999.

[27] M. J. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *Computers, IEEE Transactions on*, vol. 37, no. 9, pp. 1088–1098, 1988.

[28] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: rethinking and rebooting gprof for the multicore age," in *ACM SIGPLAN Notices*, vol. 46, pp. 458–469, ACM, 2011.

[29] D. Jeon, S. Garcia, C. Louie, S. Kota Venkata, and M. B. Taylor, "Kremlin: Like gprof, but for parallelization," in *ACM SIGPLAN Notices*, vol. 46, pp. 293–294, ACM, 2011.

[30] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor, "Parkour: Parallel speedup estimates for serial programs," in *HotPar11: Proceedings of the USENIX workshop on Hot Topics in Parallelism*, 2011.

[31] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor, "Kismet: parallel speedup estimates for serial programs," in *ACM SIGPLAN Notices*, vol. 46, pp. 519–536, ACM, 2011.

[32] Z. Li, A. Jannesari, and F. Wolf, "Discovery of potential parallelism in sequential programs," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, pp. 1004–1013, IEEE, 2013.

[33] A. J. C. Van Gemund, *Performance modeling of parallel systems*. Delft University Press, 1996.

[34] G. D. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[35] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," *Computers, IEEE Transactions on*, vol. 38, no. 3, pp. 408–423, 1989.

[36] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[37] R. Ramey, "Boost serialization library," 2008.

[38] K. Martin and B. Hoffman, *Mastering CMake*. Kitware, 2010.

[39] L. VHDL, "Verilog system users manual," *VHDL Compiler Version*, vol. 4, no. 1, 1993.

[40] M. LLC, "Wavedrom," *http://wavedrom.com*, 2014-09-15.

[41] A. Balevic *et al.*, *Exploiting multi-level parallelism in streaming applications for heterogeneous platforms with GPUs*. PhD thesis, Leiden Institute of Advanced Computer Science (LIACS), and Leiden Embedded Research Center, Faculty of Science, Leiden University, 2013.

[42] D. Nadezhkin, H. Nikolov, and T. Stefanov, "Translating affine nested-loop programs with dynamic loop bounds into polyhedral process networks," in *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pp. 21–30, IEEE, 2010.

[43] S. Verdoolaege, "Polyhedral process networks," in *Handbook of Signal Processing Systems*, pp. 1335–1375, Springer, 2013.

[44] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.

[45] O. Krzikalla, "Performing source-to-source transformations with clang," 2013. Poster presented at the European LLVM Conference Paris 2013, Paris, France.

[46] D. Spinellis, "Global analysis and transformations in preprocessed languages," *Software Engineering, IEEE Transactions on*, vol. 29, no. 11, pp. 1019–1030, 2003.

[47] S. J. Geuns, M. J. G. Bekooij, T. Bijlsma, and H. Corporaal, "Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6, IEEE, 2011.

[48] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic c-to-cuda code generation for affine programs," in *Compiler Construction*, pp. 244–263, Springer, 2010.

[49] S. S. Kumar, A. Chahar, and R. van Leuken, "Cit: A gcc plugin for the analysis and characterization of data dependencies in parallel programs," 2013.

[50] C.-Q. Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *Distributed Computing Systems, 1988., 8th International Conference on*, pp. 366–373, IEEE, 1988.

[51] A. Chahar and B. Tech, *Compile Time Analysis for Hardware Transactional Memory Architectures*. PhD thesis, Masters thesis, Delft University of Technology, 2012.

[52] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[53] R. Duncan, "A survey of parallel computer architectures," *Computer*, vol. 23, no. 2, pp. 5–16, 1990.

[54] D. C. Atkinson and W. G. Griswold, "The design of whole-program analysis tools," in *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pp. 16–27, IEEE, 1996.

[55] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[56] M. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 948–960, 1972.

[57] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis*. Xilinx, 2013.

[58] B. Karlsson, *Beyond the C++ standard library: an introduction to boost*. Pearson Education, 2005.

[59] A. Alexandrescu, *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[60] K. K. Parhi, *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons, 2007.

Compiler Extension for Compaan DDE

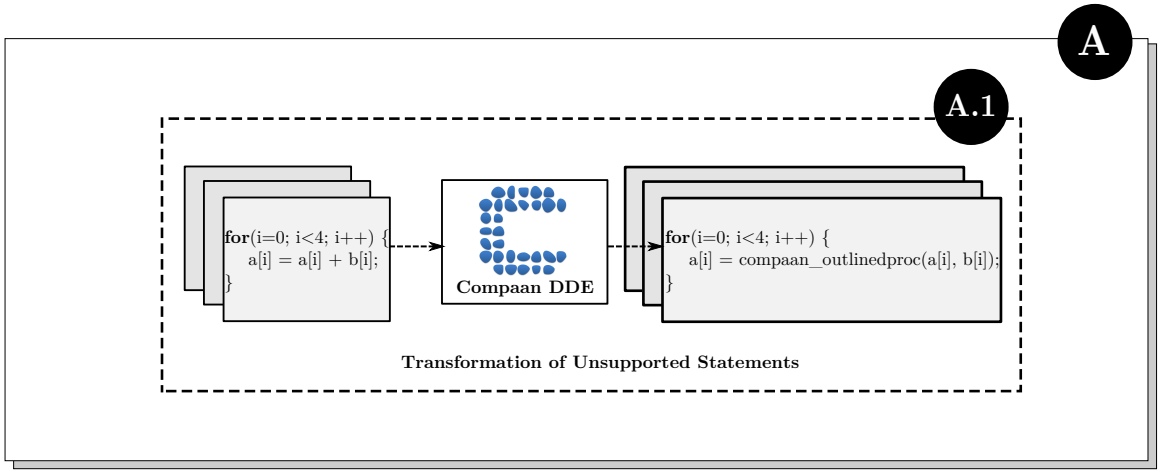


Figure A.1: Overview of the compiler extension.

A.1 Introduction

The Compaan *Design Development Environment* (DDE) is a high-level synthesis tool to transform SANLPs to synthesizable VHDL. Large subsystems of the DDE are implemented in the LLVM/Clang framework. Compaan DDE supports processing of SANLPs that are not modeled using function calls, as required by cprof. It collects all statements not modeled with function calls, and automatically creates functions with equivalent behavior to the original statement, as shown in Figure A.1.

Compaan DDE generates new functions, but does not substitute them in the original source code. The functions are automatically mapped to a database with their implementations, and the code is derived automatically in subsequent processes. Now, the problem is, that in order to automatically profile source code, the generated functions need to replace their counterparts in the original source code. To facilitate this transformation, a compiler extension has been developed. This compiler extension is responsible for substituting the unsupported statements with their supported equivalents. This extension is integrated in the design flow and, hence, it is possible to automatically generate supported source code for cprof.

A.1.1 Design and Implementation of the Compiler Extension

The extension is designed for the LLVM/Clang framework. The extension is activated by a command line argument, which has been added to the compiler front-end. Compaan DDE generates the declarations and implementations of function calls, and stores this information in a database. The data is stored in an *Extensible Markup Language* (XML) file, where the data is modeled as an AST. Furthermore, the XML representation of the AST is enhanced with information about the location of unsupported statements.

With the locations of the unsupported statements in place, it is possible to traverse the AST and apply source-to-source transformations to generate a new program, with the unsupported statements substituted with their supported equivalents. In order to successfully substitute the unsupported statements with their supported equivalents, Algorithm 5 is implemented in the compiler infrastructure.

Algorithm 5 Substituting algorithm for inserting valid statements into the AST.

Precondition: *stmts* is a tree of valid Compaan DDE statements of length n

```
1: function SUBSTITUTE(stmts)
2:     for each stmt ∈ stmts do
3:         if stmt is assignmentStmt then
4:             if substitute[stmt] then
5:                 stmt ← substitute[stmt]
6:         else if stmt is otherStmt then
7:             return SUBSTITUTE(stmt.GETCHILDS())
8:         else
9:             return
```

This is a recursive algorithm, which is necessary because of the nature of the abstract syntax three. In line 4, there is a lookup to check if the statement has a suitable replacement. If and only if this is the case, the statement is replaced in line 5. The space and time complexity of the algorithm is $\mathcal{O}(n)$. The place of Algorithm 5 within the system is shown in Figure A.2.

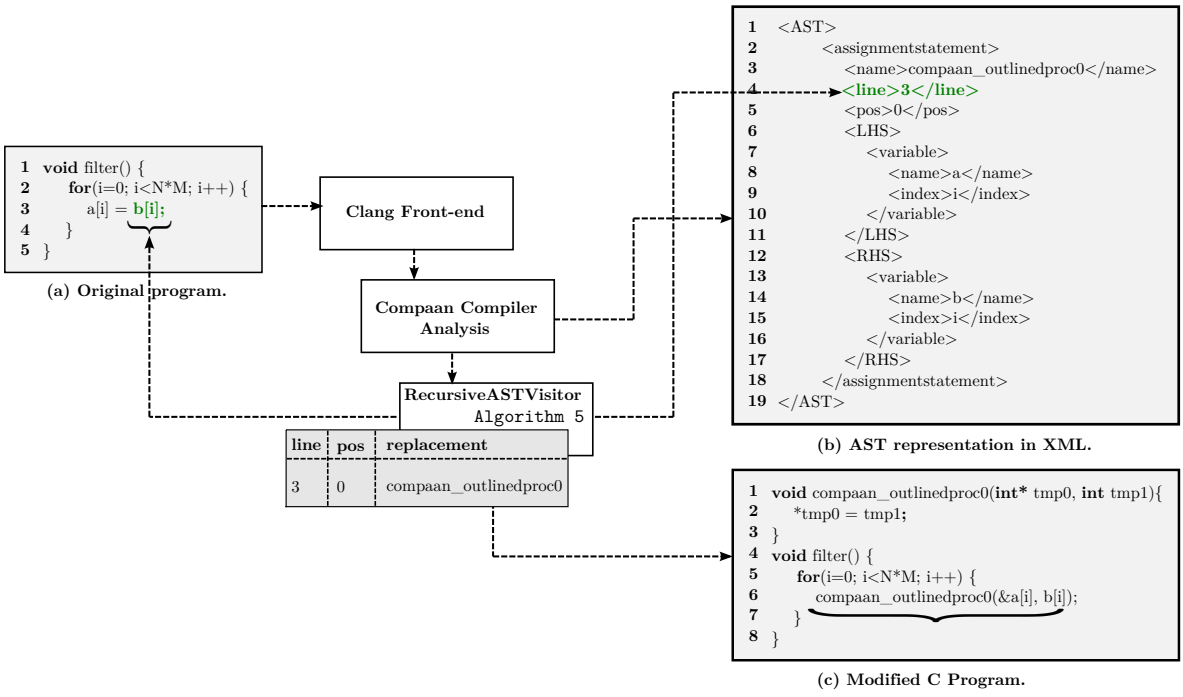


Figure A.2: Compiler Extension.

The original program, shown in Figure A.2(a), is processed by the Clang front-end and subsequently inspected by the Compaan analyzer. The resulting AST, including a substitution for the code in line 3 in Figure A.2(a), is shown in Figure A.2(b). The `RecursiveASTVisitor` is responsible for

executing Algorithm 5. The table with the line and position information is used to check if a statement has a valid replacement. After substituting the unsupported code, the new source code is generated using the `Rewriter` object, and the result is shown in Figure A.2(c).

Cprof Usage Instructions

B

In this appendix, we explain the usage instructions of cprof.

B.1 Introduction

Cprof can be used on both Microsoft Windows and Linux. To successfully compile cprof, LLVM 3.1 and Boost 1.55.0 are necessary. For Boost you need the header files and the (precompiled) libraries.

B.2 Installation

On both platforms, cmake can be used to compile cprof. The instructions for cmake are in the `README` file.

B.3 Usage

After compilation, the user should run cprof with the `--help` argument to see the available command line arguments. Example invocations are shown in the `README` file. If the user wish to profile with hierarchical program analysis disabled, he/she should select kernels of interest with the `#pragma cprof_procedure kernel_name` directive. The `AD_OD` definition in the instrumented source code can be used to indicate whether anti (WAR) and output (WAW) dependencies must be modeled. The user should set `CPROF_ABS_THROUGHPUT` to false to determine the unbounded throughput estimate. The absolute throughput estimate is the default mode of operation. It is possible to print the contents of the statement execution profiles to screen by setting the `CPROF_DEBUG` statement to true. To view the generated waveforms, a program such as Waveview (http://www.eda.ir/page_waview.htm) can be used.

C Predictor Optimized Versions

C.1 Inner Loop Unrolled

```
1 void predictor(int data_in[X][Y], int data_out[X][Y]) {
2   int a[X][Y];
3   int i, j, x;
4
5   // source
6   for (i = 0; i <= 4; i = i + 1) {
7     for(j = 0; j <= 4; j = j + 1) {
8     source(&a[i][j], data_in[i][j]);
9     }
10  }
11  // transformer
12  for (i = 1; i <= 4; i = i + 1) {
13      transformer(&a[i][1], a[i-1][1], a[i][1-1]);
14      transformer(&a[i][2], a[i-1][2], a[i][2-1]);
15      transformer(&a[i][3], a[i-1][3], a[i][3-1]);
16      transformer(&a[i][4], a[i-1][4], a[i][4-1]);
17  }
18  // sink
19  for (i = 1; i <= 4; i = i + 1) {
20    for(j = 1; j <= 4; j = j + 1) {
21    sink(a[i][j], &data_out[i][j]);
22    }
23   }
24 }
```

Listing C.1: Inner loop unrolled.

C.2 Outer loop Unrolled

```
1 void predictor(int data_in[X][Y], int data_out[X][Y]) {
2   int a[X][Y];
3   int i, j, x;
4
5   // source
6   for (i = 0; i <= 4; i = i + 1) {
7     for(j = 0; j <= 4; j = j + 1) {
8     source(&a[i][j], data_in[i][j]);
9     }
10  }
11
```

```
12   // transformer
13   for (j = 1; j <= 4; j = j + 1) {
14     transformer(&a[1][j], a[1-1][j], a[1][j-1]);
15   }
16   for (j = 1; j <= 4; j = j + 1) {
17     transformer(&a[2][j], a[2-1][j], a[2][j-1]);
18   }
19   for (j = 1; j <= 4; j = j + 1) {
20     transformer(&a[3][j], a[3-1][j], a[3][j-1]);
21   }
22   for (j = 1; j <= 4; j = j + 1) {
23     transformer(&a[4][j], a[4-1][j], a[4][j-1]);
24   }
25   // sink
26   for (i = 1; i <= 4; i = i + 1) {
27     for(j = 1; j <= 4; j = j + 1) {
28     sink(a[i][j], &data_out[i][j]);
29     }
30    }
31 }
```

Listing C.2: Outer-loop unrolled.

C.3 Inner/Outer Loops Unrolled

```
1  void predictor(int data_in[X][Y], int data_out[X][Y]) {
2    int a[X][Y];
3    int i, j, x;
4
5    // source
6    for (i = 0; i <= 4; i = i + 1) {
7      for(j = 0; j <= 4; j = j + 1) {
8      source(&a[i][j], data_in[i][j]);
9      }
10   }
11   // transformer
12   transformer(&a[1][1], a[1-1][1], a[1][1-1]);
13   transformer(&a[1][2], a[1-1][2], a[1][2-1]);
14   transformer(&a[1][3], a[1-1][3], a[1][3-1]);
15   transformer(&a[1][4], a[1-1][4], a[1][4-1]);
16
17   transformer(&a[2][1], a[2-1][1], a[2][1-1]);
18   transformer(&a[2][2], a[2-1][2], a[2][2-1]);
19   transformer(&a[2][3], a[2-1][3], a[2][3-1]);
20   transformer(&a[2][4], a[2-1][4], a[2][4-1]);
21
22   transformer(&a[3][1], a[3-1][1], a[3][1-1]);
23   transformer(&a[3][2], a[3-1][2], a[3][2-1]);
24   transformer(&a[3][3], a[3-1][3], a[3][3-1]);
25   transformer(&a[3][4], a[3-1][4], a[3][4-1]);
26
```

```
27   transformer(&a[4][1], a[4-1][1], a[4][1-1]);
28   transformer(&a[4][2], a[4-1][2], a[4][2-1]);
29   transformer(&a[4][3], a[4-1][3], a[4][3-1]);
30   transformer(&a[4][4], a[4-1][4], a[4][4-1]);
31   // sink
32   for (i = 1; i <= 4; i = i + 1) {
33     for(j = 1; j <= 4; j = j + 1) {
34     sink(a[i][j], &data_out[i][j]);
35     }
36    }
37 }
```

Listing C.3: Inner/Outer Unrolled.

C.4 Inner/Outer/Sink Loops Unrolled

```
1  void predictor(int data_in[X][Y], int data_out[X][Y]) {
2
3    int a[X][Y];
4    int i, j, x;
5
6    // source
7    for (i = 0; i <= 4; i = i + 1) {
8      for(j = 0; j <= 4; j = j + 1) {
9      source(&a[i][j], data_in[i][j]);
10     }
11   }
12
13   // transformer
14   transformer(&a[1][1], a[1-1][1], a[1][1-1]);
15   transformer(&a[1][2], a[1-1][2], a[1][2-1]);
16   transformer(&a[1][3], a[1-1][3], a[1][3-1]);
17   transformer(&a[1][4], a[1-1][4], a[1][4-1]);
18
19   transformer(&a[2][1], a[2-1][1], a[2][1-1]);
20   transformer(&a[2][2], a[2-1][2], a[2][2-1]);
21   transformer(&a[2][3], a[2-1][3], a[2][3-1]);
22   transformer(&a[2][4], a[2-1][4], a[2][4-1]);
23
24   transformer(&a[3][1], a[3-1][1], a[3][1-1]);
25   transformer(&a[3][2], a[3-1][2], a[3][2-1]);
26   transformer(&a[3][3], a[3-1][3], a[3][3-1]);
27   transformer(&a[3][4], a[3-1][4], a[3][4-1]);
28
29   transformer(&a[4][1], a[4-1][1], a[4][1-1]);
30   transformer(&a[4][2], a[4-1][2], a[4][2-1]);
31   transformer(&a[4][3], a[4-1][3], a[4][3-1]);
32   transformer(&a[4][4], a[4-1][4], a[4][4-1]);
33
34   // sink
35   sink(a[1][1], &data_out[1][1]);
```

```
36    sink(a[1][2], &data_out[1][2]);
37    sink(a[1][3], &data_out[1][3]);
38    sink(a[1][4], &data_out[1][4]);
39
40    sink(a[2][1], &data_out[2][1]);
41    sink(a[2][2], &data_out[2][2]);
42    sink(a[2][3], &data_out[2][3]);
43    sink(a[2][4], &data_out[2][4]);
44
45    sink(a[3][1], &data_out[3][1]);
46    sink(a[3][2], &data_out[3][2]);
47    sink(a[3][3], &data_out[3][3]);
48    sink(a[3][4], &data_out[3][4]);
49
50    sink(a[4][1], &data_out[4][1]);
51    sink(a[4][2], &data_out[4][2]);
52    sink(a[4][3], &data_out[4][3]);
53    sink(a[4][4], &data_out[4][4]);
54 }
```

Listing C.4: Inner/Outer/Sink unrolled.

C.5 Source/Inner/Outer Loops Unrolled

```
1  void predictor(int data_in[X][Y], int data_out[X][Y]) {
2
3    int a[X][Y];
4    int i, j, x;
5
6    // source
7    source(&a[0][0], data_in[0][0]);
8    source(&a[0][1], data_in[0][1]);
9    source(&a[0][2], data_in[0][2]);
10   source(&a[0][3], data_in[0][3]);
11   source(&a[0][4], data_in[0][4]);
12
13   source(&a[1][0], data_in[1][0]);
14   source(&a[1][1], data_in[1][1]);
15   source(&a[1][2], data_in[1][2]);
16   source(&a[1][3], data_in[1][3]);
17   source(&a[1][4], data_in[1][4]);
18
19   source(&a[2][0], data_in[2][0]);
20   source(&a[2][1], data_in[2][1]);
21   source(&a[2][2], data_in[2][2]);
22   source(&a[2][3], data_in[2][3]);
23   source(&a[2][4], data_in[2][4]);
24
25   source(&a[3][0], data_in[3][0]);
26   source(&a[3][1], data_in[3][1]);
27   source(&a[3][2], data_in[3][2]);
```

```
28    source(&a[3][3], data_in[3][3]);
29    source(&a[3][4], data_in[3][4]);
30
31    source(&a[4][0], data_in[4][0]);
32    source(&a[4][1], data_in[4][1]);
33    source(&a[4][2], data_in[4][2]);
34    source(&a[4][3], data_in[4][3]);
35    source(&a[4][4], data_in[4][4]);
36
37    // transformer
38    transformer(&a[1][1], a[1-1][1], a[1][1-1]);
39    transformer(&a[1][2], a[1-1][2], a[1][2-1]);
40    transformer(&a[1][3], a[1-1][3], a[1][3-1]);
41    transformer(&a[1][4], a[1-1][4], a[1][4-1]);
42
43    transformer(&a[2][1], a[2-1][1], a[2][1-1]);
44    transformer(&a[2][2], a[2-1][2], a[2][2-1]);
45    transformer(&a[2][3], a[2-1][3], a[2][3-1]);
46    transformer(&a[2][4], a[2-1][4], a[2][4-1]);
47
48    transformer(&a[3][1], a[3-1][1], a[3][1-1]);
49    transformer(&a[3][2], a[3-1][2], a[3][2-1]);
50    transformer(&a[3][3], a[3-1][3], a[3][3-1]);
51    transformer(&a[3][4], a[3-1][4], a[3][4-1]);
52
53    transformer(&a[4][1], a[4-1][1], a[4][1-1]);
54    transformer(&a[4][2], a[4-1][2], a[4][2-1]);
55    transformer(&a[4][3], a[4-1][3], a[4][3-1]);
56    transformer(&a[4][4], a[4-1][4], a[4][4-1]);
57
58    // sink
59    for (i = 1; i <= 4; i = i + 1) {
60      for(j = 1; j <= 4; j = j + 1) {
61      sink(a[i][j], &data_out[i][j]);
62      }
63     }
64 }
```

Listing C.5: Source/Inner/Outer unrolled.

C.6 Source/Inner/Outer/Sink Loops Unrolled

```
1 void predictor(int data_in[X][Y], int data_out[X][Y]) {
2
3    int a[X][Y];
4    int i, j, x;
5
6    // source
7    source(&a[0][0], data_in[0][0]);
8    source(&a[0][1], data_in[0][1]);
9    source(&a[0][2], data_in[0][2]);
```

```
10    source(&a[0][3], data_in[0][3]);
11    source(&a[0][4], data_in[0][4]);
12
13    source(&a[1][0], data_in[1][0]);
14    source(&a[1][1], data_in[1][1]);
15    source(&a[1][2], data_in[1][2]);
16    source(&a[1][3], data_in[1][3]);
17    source(&a[1][4], data_in[1][4]);
18
19    source(&a[2][0], data_in[2][0]);
20    source(&a[2][1], data_in[2][1]);
21    source(&a[2][2], data_in[2][2]);
22    source(&a[2][3], data_in[2][3]);
23    source(&a[2][4], data_in[2][4]);
24
25    source(&a[3][0], data_in[3][0]);
26    source(&a[3][1], data_in[3][1]);
27    source(&a[3][2], data_in[3][2]);
28    source(&a[3][3], data_in[3][3]);
29    source(&a[3][4], data_in[3][4]);
30
31    source(&a[4][0], data_in[4][0]);
32    source(&a[4][1], data_in[4][1]);
33    source(&a[4][2], data_in[4][2]);
34    source(&a[4][3], data_in[4][3]);
35    source(&a[4][4], data_in[4][4]);
36
37    // transformer
38    transformer(&a[1][1], a[1-1][1], a[1][1-1]);
39    transformer(&a[1][2], a[1-1][2], a[1][2-1]);
40    transformer(&a[1][3], a[1-1][3], a[1][3-1]);
41    transformer(&a[1][4], a[1-1][4], a[1][4-1]);
42
43    transformer(&a[2][1], a[2-1][1], a[2][1-1]);
44    transformer(&a[2][2], a[2-1][2], a[2][2-1]);
45    transformer(&a[2][3], a[2-1][3], a[2][3-1]);
46    transformer(&a[2][4], a[2-1][4], a[2][4-1]);
47
48    transformer(&a[3][1], a[3-1][1], a[3][1-1]);
49    transformer(&a[3][2], a[3-1][2], a[3][2-1]);
50    transformer(&a[3][3], a[3-1][3], a[3][3-1]);
51    transformer(&a[3][4], a[3-1][4], a[3][4-1]);
52
53    transformer(&a[4][1], a[4-1][1], a[4][1-1]);
54    transformer(&a[4][2], a[4-1][2], a[4][2-1]);
55    transformer(&a[4][3], a[4-1][3], a[4][3-1]);
56    transformer(&a[4][4], a[4-1][4], a[4][4-1]);
57
58    // sink
59    sink(a[1][1], &data_out[1][1]);
60    sink(a[1][2], &data_out[1][2]);
61    sink(a[1][3], &data_out[1][3]);
62    sink(a[1][4], &data_out[1][4]);
```

```
63
64    sink(a[2][1], &data_out[2][1]);
65    sink(a[2][2], &data_out[2][2]);
66    sink(a[2][3], &data_out[2][3]);
67    sink(a[2][4], &data_out[2][4]);
68
69    sink(a[3][1], &data_out[3][1]);
70    sink(a[3][2], &data_out[3][2]);
71    sink(a[3][3], &data_out[3][3]);
72    sink(a[3][4], &data_out[3][4]);
73
74    sink(a[4][1], &data_out[4][1]);
75    sink(a[4][2], &data_out[4][2]);
76    sink(a[4][3], &data_out[4][3]);
77    sink(a[4][4], &data_out[4][4]);
78 }
```

Listing C.6: Source/Inner/Outer/Sink Unrolled.

PolyBench/C 3.1 Benchmarks

D

Benchmark	Description
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
correlation	Correlation Computation
covariance	Covariance Computation
doitgen	Multiresolution analysis kernel (MADNESS)
durbin	Toeplitz system solver
dynprog	Dynamic programming (2D)
fdtd-2D	2-D Finite Different Time Domain Kernel
fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer
floyd warshall	All-pairs Shortest Path solving
gauss-filter	Gaussian Filter
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition
mvt	Matrix Vector Product and Transpose
mm2	2 Matrix Multiplications (D=A.B; E=C.D)
mm3	3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)
reg-detect	2-D Image processing
seidel	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k operations
syrk	Symmetric rank-k operations
trisolv	Triangular solver
trmm	Triangular matrix-multiply

Table D.1: PolyBench/C Benchmarks.

Support of Control Flow Architectures in Cprof

E

In this appendix, the modeling of anti (WAR) and output (WAW) dependencies in cprof is discussed. In Section E.1, the required modifications are discussed.

E.1 Modification of Algorithms

The algorithms for dynamic analysis presented in Section 5.4.2 are modified to support anti and output dependencies.

E.1.1 Read Operations

Algorithm 1 is used to keep track of read operations. The modified algorithm keeps track at which point in time the variable is read. In line 10 of Algorithm 6, the finish time of the read operation is stored. The `updateReads` is implemented in the `CprofVariable` class and is responsible for updating the read information.

Algorithm 6 Update shadow variables for read access, with support for anti and output dependencies.

Precondition: *indices* is a list of indices for accessing the shadow variables of length n

```
1:  function UpdateReads(indices)
2:      start ← C$s
3:      stop ← C$s + Λ_R
4:      maxStop ← 0
5:      for each argument a in readArguments do
6:          dims ← a.getDimensions()
7:          for y = 0 to dims do
8:              l.push(indices[y])
9:          a.getWrites(l, &maxStop)
10:         a.updateReads(i, max(stop, maxStop))
11:     if UnboundedThroughputEstimate then
12:         C$s ← maxStop
13:     else
14:         C$s ← max(stop, maxStop)
15:     UpdateStatementProfile(start, stop, Λ_R)
```

E.1.2 Write Operations

Algorithm 7 is used to keep track of write operations. In line 4 and 5 of Algorithm 7, two new variables are declared. The `maxWrite` and `maxRead` are used to store the timestamps related to write and read operations. In line 14 and 15 the latest write and read timestamps are stored in the `maxWrite` and `maxRead` variables. It is important to note that the `getWrites` and `getReads` statements only consider preceding reads and writes. It is only possible to write the new variable if all the preceding statements have written and read the variable. From line 14 to 19, this behavior is modeled. With these modifications in place, the write operation is delayed until the anti and output dependencies are solved.

Algorithm 7 Update shadow variables for write access, with support for anti and output dependencies.

Precondition: *indices* is a list of indices for accessing the shadow variables of length n

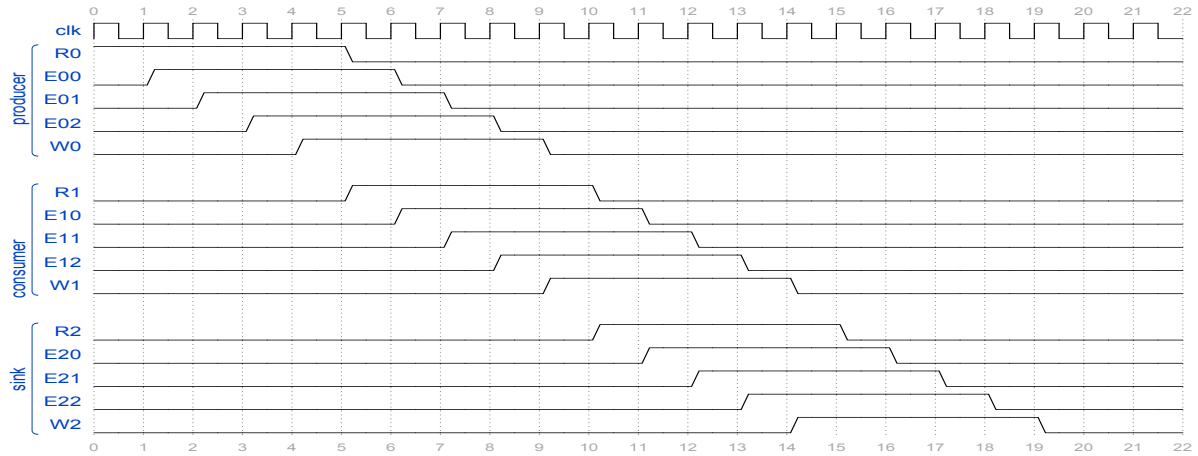
```
1:  function UpdateWrites(indices)
2:      start ← C$s + Λ_F
3:      stop ← C$s + Λ_F + Λ_W
4:      maxWrite ← 0
5:      maxRead ← 0
6:      if AbsoluteThroughputEstimate then
7:          start ← start − II_F
8:          stop ← stop − II_F
9:      maxStopValue ← 0
10:     for each argument a in writeArguments do
11:         dims ← a.getDimensions()
12:         for y = 0 to dims do
13:             l.push(indices[y])
14:         a.getWrites(indices, &maxWrite)
15:         a.getReads(indices, &maxRead)
16:         if maxReadValue ≥ stop then
17:             start ← maxRead
18:             stop ← maxRead + Λ_W
19:         stop ← max(stop, maxWrite)
20:         a.updateWrites(l, stop)
21:     UpdateStatementProfile(start, stop, Λ_W)
```

Verification Waveforms

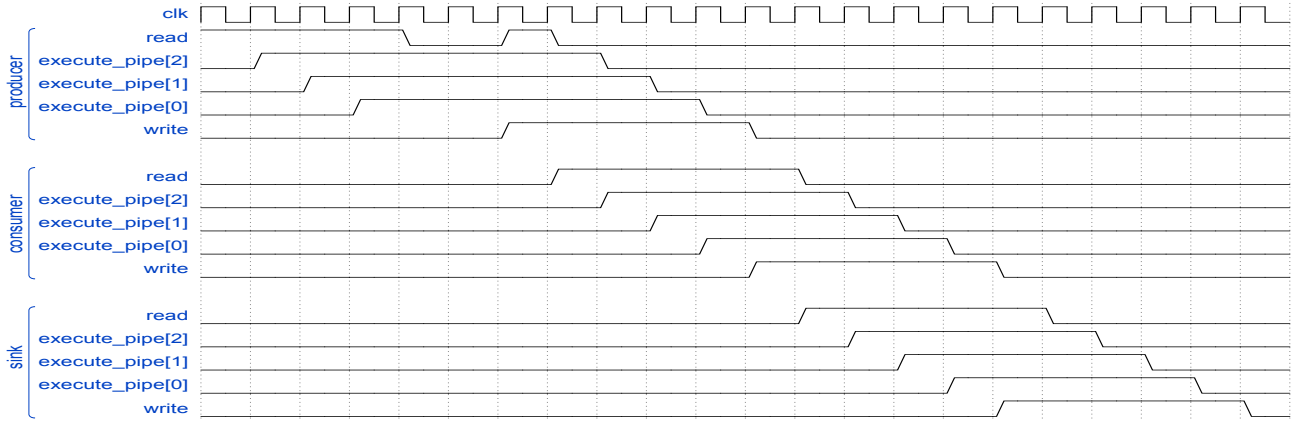
F

In this appendix, the waveforms used for verification of the communication models in PPNs are shown. In Section 6.2, we have discussed the generated waveforms.

F.1 IOM- Waveforms



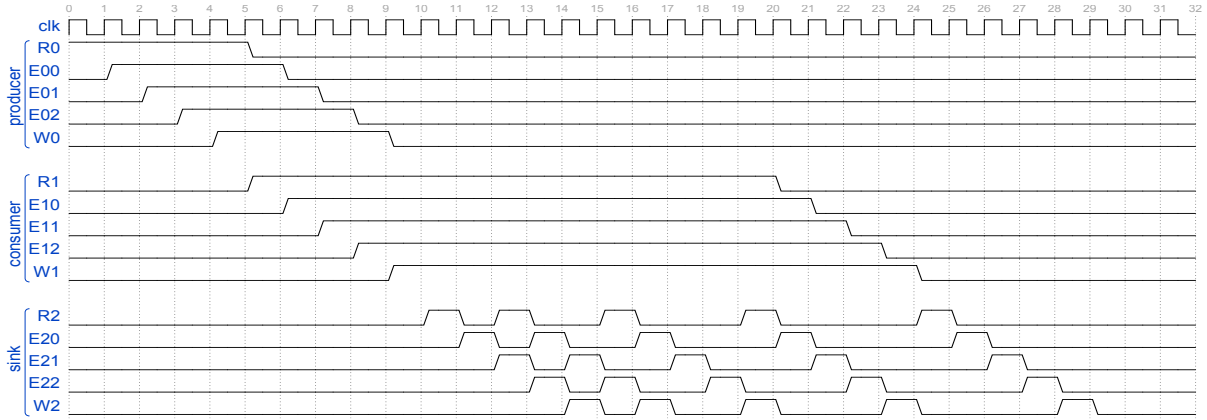
(a) Cprof generated waveform for the IOM- communication model.



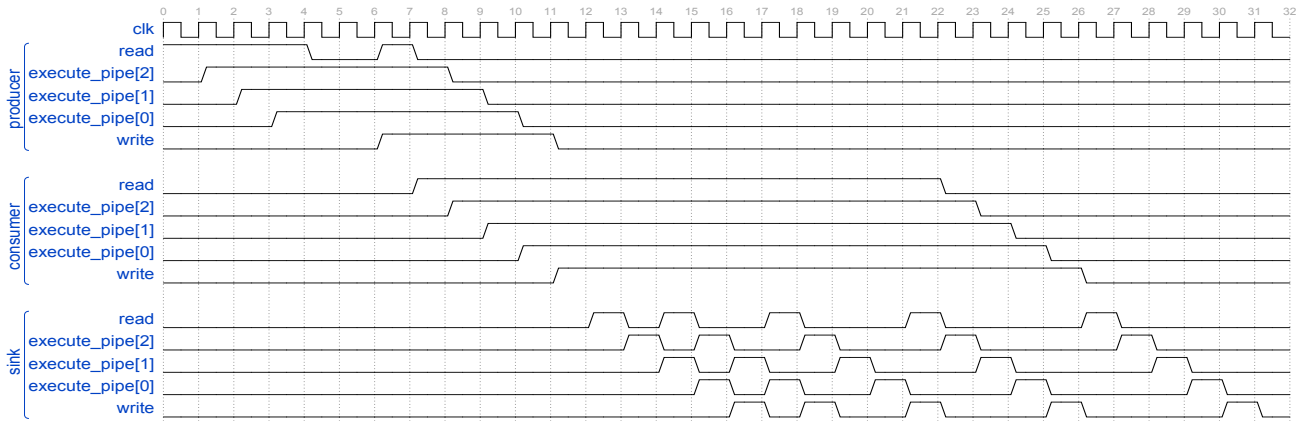
(b) Xilinx ISE generated waveform for the IOM- communication model.

Figure F.1: Waveforms representing the execution of the program implementing the IOM- communication model.

F.2 IOM+ Waveforms



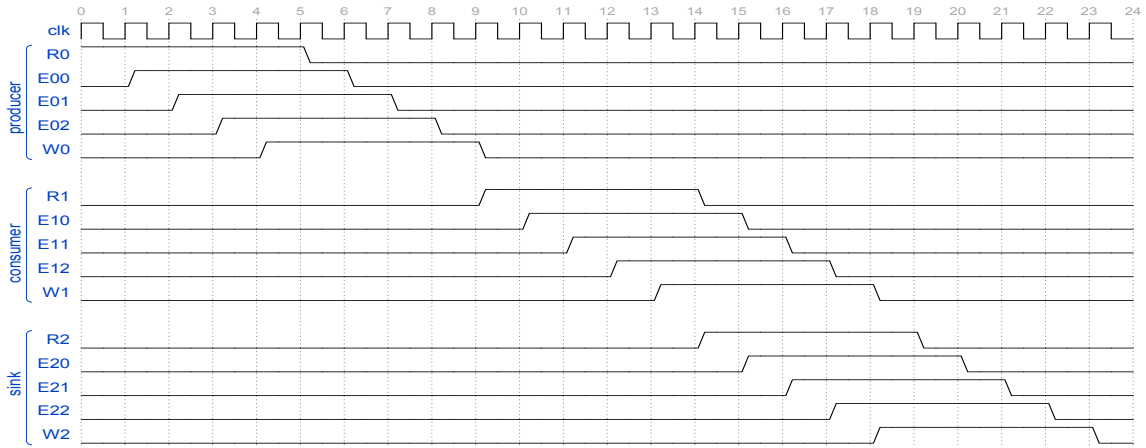
(a) Cprof generated waveform for the IOM+ communication model.



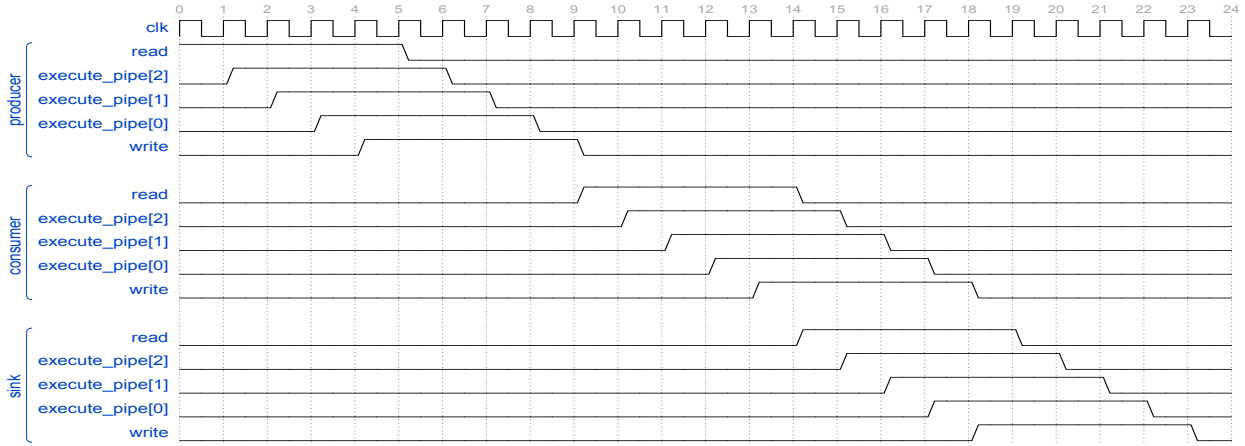
(b) Xilinx ISE generated waveform for the IOM+ communication model.

Figure F.2: Waveforms representing the execution of the program implementing the IOM+ communication model.

F.3 OOM- Waveforms



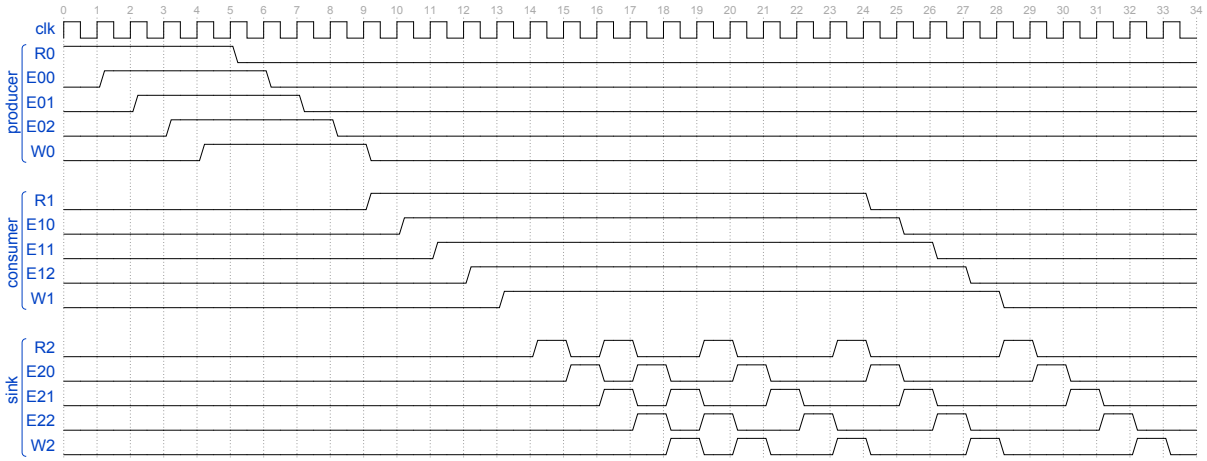
(a) Cprof generated waveform for the OOM- communication model.



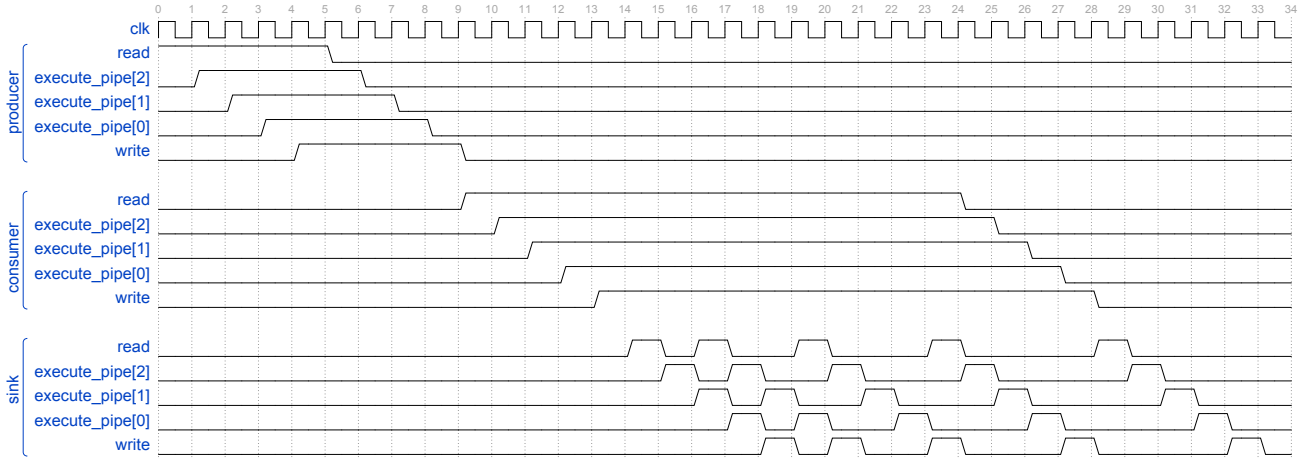
(b) Xilinx ISE generated waveform for the OOM- communication model.

Figure F.3: Waveforms representing the execution of the program implementing the OOM- communication model.

F.4 OOM+ Waveforms



(a) Cprof generated waveform for the OOM+ communication model.



(b) Xilinx ISE generated waveform for the OOM+ communication model.

Figure F.4: Waveforms representing the execution of the program implementing the OOM+ communication model.