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

CAS-2016-03

M.Sc. Thesis

Validation of Performance Estimation, Channel Sizing and Automatic Loop Transformations of High Level Specifications for Polyhedral Process Network Applications

Christopher McGirr B.Sc. Electrical Engineering

Abstract

In this thesis, we present Cprof+, an upgraded version of Cprof. Cprof+ is a lightweight profiling tool for *Polyhedral Process Networks* (PPN) that can estimate the performance of a C-program as a PPN implemented in hardware. Cprof+ improves the set of possible programs that can be successfully profiled, by increasing the performance estimation accuracy to within 94% of RTL simulations. Using the Compaan Compiler and the Xilinix Vivado RTL simulator, Cprof+ was verified against all 29 Polybench Suite benchmarks. The purpose of Cprof+ is to provide a lightweight and rapid performance estimation tool for PPN applications. The simulation run-time of Cprof+ shows on average an execution time of 121 seconds compared to 598 seconds with Vivado RTL simulations. Cprof+ can also estimate channel sizes of the PPN interconnect based on execution profiles of the processes. The estimated channel sizes achieved on average a 77% reduction in communication memory when compared to Compaan.

The Cprof+ profiler can also aid the designer of PPN applications by applying automated source-to-source transformations on C-programs to assist in the design space exploration. These transformations are applied using an optimization technique based on channel sizes of the network interconnect. This technique yields on average a 64% increase in performance of the network latency with a 6x factor increase in hardware resources required.

Validation of Performance Estimation, Channel Sizing and Automatic Loop Transformations of High Level Specifications for Polyhedral Process Network Applications

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Embedded Systems

by

Christopher McGirr B.Sc. Electrical Engineering
born in Chatham, NB, Canada

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 **"Validation of Performance Estimation, Channel Sizing and Automatic Loop Transformations of High Level Specifications for Polyhedral Process Network Applications"** by **Christopher McGirr B.Sc. Electrical Engineering** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 29th November 2016

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 Z. Al-Ars

Abstract

In this thesis, we present Cprof+, an upgraded version of Cprof. Cprof+ is a lightweight profiling tool for *Polyhedral Process Networks* (PPN) that can estimate the performance of a C-program as a PPN implemented in hardware. Cprof+ improves the set of possible programs that can be successfully profiled, by increasing the performance estimation accuracy to within 94% of RTL simulations. Using the Compaan Compiler and the Xilinix Vivado RTL simulator, Cprof+ was verified against all 29 Polybench Suite benchmarks. The purpose of Cprof+ is to provide a lightweight and rapid performance estimation tool for PPN applications. The simulation run-time of Cprof+ shows on average an execution time of 121 seconds compared to 598 seconds with Vivado RTL simulations. Cprof+ can also estimate channel sizes of the PPN interconnect based on execution profiles of the processes. The estimated channel sizes achieved on average a 77% reduction in communication memory when compared to Compaan.

The Cprof+ profiler can also aid the designer of PPN applications by applying automated source-to-source transformations on C-programs to assist in the design space exploration. These transformations are applied using an optimization technique based on channel sizes of the network interconnect. This technique yields on average a 64% increase in performance of the network latency with a 6x factor increase in hardware resources required.

Acknowledgments

First, I would like to thank Dr.Bart Kienhuis for supporting me throughout the year. Thank you for the many skype calls that helped to guide me through the world of polyhedral process networks and proofreading my thesis many times.

I am also thankful to professor Rene van Leuken, who guided me through the thesis process and helped to proofread my thesis. I would also like to thank Carlo Galuzzi, who took an interest in my work.

I am also grateful to my many friends I made here during my studies at TU Delft. You made my studies here much more enjoyable and memorable.

Last, but not least, I would like to thank my family, in particular my parents, Christine and Kevin. Thank you for supporting me throughout the years. I would not have made it to this point in my life without you.

Christopher McGirr B.Sc. Electrical Engineering
Delft, The Netherlands
29th November 2016

Contents

List of Figures

List of Tables

1 Introduction

Modern embedded systems often require implementation of complex and data intensive applications, e.g. applications like object detection[2] and object tracking[3]. Implementing these applications is simplified using *High Level Synthesis* (HLS) tools. HLS aims to increase the productivity of the designer by abstracting the hardware design to high level languages such as C/C++. The C code is then translated to digital hardware. The designer verifies that all requirements are met of the synthesized design using *Register Transfer Level* (RTL) and timing simulations. This verification can be time consuming, taking hours for one simulation[1].

Since a RTL simulation may take hours, the designer needs a tool to give an idea of the performance of the implementation. A tool known as Cprof+, based on Cprof[4], is introduced to aid the designer of *Polyhedral Process Networks* (PPNs). Cprof+ can estimate the performance of the application as a PPN in hardware. The time required for performance estimation is significantly less when compared to RTL simulations.

As the performance of a PPN can be estimated rapidly in software, Cprof+ can be used to explore the design space of an application. Exploring the designs of a program in software increases the productivity and reduces the risk of design errors. The translation of programs into hardware, known as HLS involves converting high level specifications to a hardware implementation that can be mapped to a Field Programmable Gate Array (FPGA), Multiprocessor System on a Chip (MPSoC) or other platforms. In this thesis, Compaan[6] is used in the design flow shown in Figure 1.1.

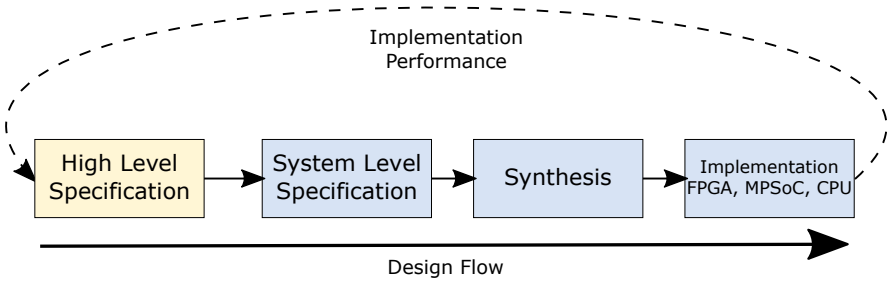


Figure 1.1: The traditional Design Flow in High Level Synthesis

The disadvantage of the design flow in Figure 1.1 is that the designer has to perform synthesis and simulation of the high level specification before a result can be obtained. This process is often time consuming. The designer only finds out at the end of the process if the design has meet the performance requirements. If the design does not fit the requirements, the designer must modify the high level specification and repeat the process again.

Cprof+ aims to modify the traditional design flow and take an iterative design approach in the software domain. This approach abstracts the hardware implementation

and only focuses upon the high level specification. Allowing designers, with no knowledge of hardware, to perform design space exploration. Cprof+ is a profiling tool that analyses execution behaviour of programs. Figure 1.2 highlights the modified design flow, showcasing Cprof+ reducing the feedback loop for the designer. The designer can gain insight early in the implementation phase if a design meets the requirements of the application.

If the design does not fit the requirements, the designer can modify the C-Code to increase parallelism in the program until the desired performance requirement is met. When the designer is satisfied with the result, the design can be committed to the complete design flow to be implemented in hardware.

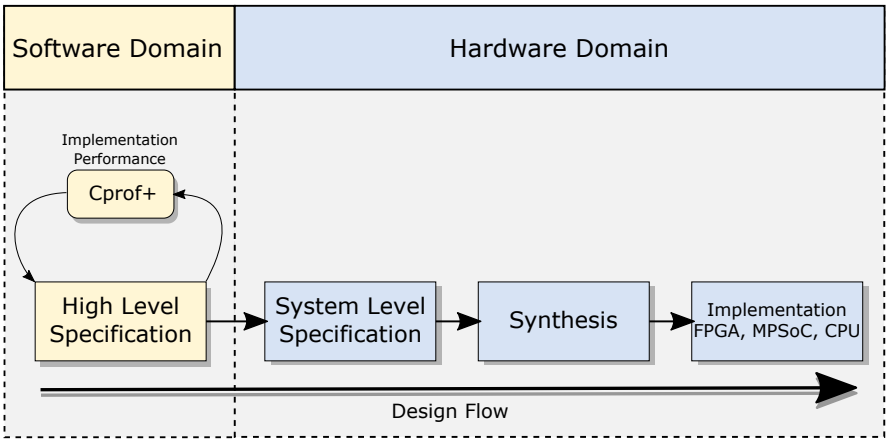


Figure 1.2: High Level Synthesis Design Flow with Cprof+

1.1 Problem Statement

The original Cprof allowed for rapid performance estimation of PPNs with high accuracy for most benchmarks in the Polybench suite. The profiler exhibited inaccuracies in estimation for some benchmarks with a maximum 60% underestimation.

The first problem we address in this thesis is whether we can improve the performance estimation accuracy of Cprof while maintaining the low simulation run time. The second problem is with this improved accuracy, can we utilize the profiling information to estimate the channel sizes of the network and can we use the channel sizes to create an automated optimization strategy for the C programs?

1.2 Contributions

The main contribution of this thesis is the upgraded profiler that we call Cprof+. Based on the profiler created by Teijlingen[4] known as Cprof, Cprof+ can estimate the performance of a program as a PPN with a high accuracy. It has been validated against 29 benchmarks from the Polybench Suite. All benchmarks contain mathematical kernels that are commonly used in data processing applications. Cprof+ allows for a more accurate exploration of the design space in a reduced amount of time, helping to improve the productivity of the designer.

The main contributions of this thesis are as follows:

1. Reduced the maximum performance estimations error from 60% to 6% using 3 methods. (Chapter 4)
   (a) Variable functional latency. (Section 4.2.1)
   (b) Run-time channel type detection. (Section 4.2.2)
   (c) Run-time channel type switching. (Section 4.2.3)
2. Utilized the available execution profile information to estimate channels sizes for the network. Reducing the memory footprint for communication by 77% when compared to Compaan. (Chapter 5)
3. Showed the effects of code source-to-source transformations on the design space and developed an automated optimization strategy. This strategy yielded 64% performance increase on average. (Chapter 6)

1.3 Thesis Outline

The thesis has the following outline. Chapter 2 will discuss the necessary background knowledge and definitions required. In Chapter 3, the related work in the field is proposed and the importance of our solution is highlighted. In Chapter 4, the solution approach and implementation for correcting the estimation inaccuracy is proposed. In Chapter 5, the estimation of channel sizes is proposed with the implementation. In Chapter 6, the code transformations and optimization techniques used by Cprof+ are discussed. The run-time impact of Cprof+ is compared to the hardware simulations in Chapter 7. Finally, we draw conclusions and present future work in Chapter 8.

2 Background

The first comprehensive implementation of the Cprof Profiler was created by Wouter van Teijlingen as presented in his Master Thesis[4]. Cprof was built upon the concept given by Sven van Haastregt who demonstrated in his doctoral thesis methods of transformation and profiling of high level specifications for PPNs[1].

In the following sections, the background knowledge required will be outlined. Section 2.1 will discuss model of computations (MoCs) and Section 2.1.1 will delve into the MoC used by Cprof and Cprof+, the Polyhedral Process Network (PPN). Section 2.3 will discuss how a PPN may be derived from a high-level specification. Section 2.2 will discuss the class of programs that will be used. Ending the chapter with Sections 2.4 and 2.5, which will discuss the use of the LLVM/Clang Frontend compiler in Cprof and Cprof+.

2.1 Models of Computation

There are many models used by designers to help describe a desired behaviour. Process Networks are often the preferred Model of Computation (MoC) for data-flow programs [7]. One of the most common is the Kahn Process Network (KPN). The advantage for process networks is the ability to easily map nodes to Multi Processor System on Chip (MPSocs) architectures due to the similarity in structure and the ability to illustrate task level parallelism.

KPNs help to model task level parallelism and explicitly model communication between nodes within the network[7]. These nodes execute sequential functions where by information is passed between nodes using unbounded FIFO channels with blocking read and non-blocking write.

2.1.1 Polyhedral Process Networks

Data-flow and streaming applications such as those used in image/video processing and mathematical calculations such as matrix multiplication are often simple programs which repeat a set of functions a bounded number of times. The bounded nature of the program means often the programs have static control flow with loop bounds known at compile time.

Similar to KPNs, Polyhedral Process Networks (PPNs) offer to model these types of applications. PPNs model statements within these programs as processes which communicate using a point-to-point method, in this case FIFOs. Unlike KPNs, PPNs have bounded FIFO sizes for communication. In this thesis, we use the following definition of a PPN as given in the work of Haastregt[1] and Teijlingen[4].

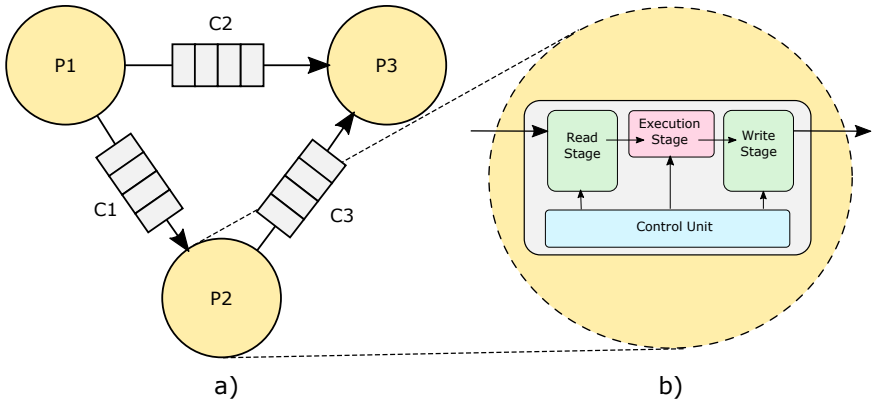


Figure 2.1: A simple PPN (a) with the structure of the implemented process (b) as found in Compaan using the LAURA Virtual Processor Model[1]

Definition 2.1.1. Polyhedral Process Network. A polyhedral process network (PPN) is a directed graph $(\mathcal{P}, \mathcal{E})$ where $\mathcal{P}$ represent the set of vertices of processes and the set $\mathcal{E}$ represents the set of edges or communication channels between these processes.

Each process $p \in \mathcal{P}$ is characterized by:

- a set of read arguments that are inputs to the node which read from a communication channel
- a set of write arguments that are outputs to the node which write to a communication channel
- a function which executes a task based on the read arguments and outputs to the write arguments.

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

- a source process
- a destination process
- a channel type of In-order-memory (IOM) or Out-of-Order memory (OOM)
- a channel size

The name polyhedral is derived from the mathematically description of the functions. Each function is described as a finite set of linear equations that form a polyhedral. The polyhedral description models the iterations and the statement relation between iterations of the function. The mathematical description allows for analysis and optimization using combinatorial and Integer Linear Programming (ILP) algorithms.

Figure 2.1(a) demonstrates a simple PPN composed of three processes with three communication channels. Just like a node in a KPN, a process cannot execute till all tokens on its input ports are present. $P2$ cannot fire till it has received a token from $P1$. $P3$ cannot fire till it has received a token from both $P1$ and $P2$.

When a process within a PPN fires, it is executing an assigned function from the program. Figure 2.1(b) demonstrates the internal structure of a process used in this work. There are three main stages to the process:

- **Read(R)** stage processes all tokens from on the inputs. If not all inputs have tokens present, the process blocks.
- **Execute(E)** stage performs a calculation on the input data and produces an output.
- **Write(W)** stage writes the output to a communication channel, a non-blocking operation.

2.2 Static Affine Nested for Loop Programs

A Static Affine Nested for Loop program (SANFLP) is a program where all statements are enclosed by one or more loops or if-statements[8]. All parameters within the conditional statement and the loop must be affine, meaning it can be represented by a set of linear equations with iterators, static program parameters and constants. All indexes must also be affine and described only by iterators, static program parameters and constants. The flow of data between statements in a loop must be explicit, no hidden shared variables. Cprof+ will focus on this subset of programs.

2.3 Deriving Polyhedral Process Networks

Sequential high level specifications such as C code can be used to describe PPNs which can be implemented in hardware. The translation from C to synthesizable hardware, is accomplished using the Compaan compiler with the *Leiden Architecture Research and Exploration Tool* (LAURA) virtual processor model of processes[9][10]. This model is highlighted in Figure 2.1(b) with the three distinct stages as well as a control unit to ensure the network schedule is executed.

The Compaan compiler fully automates the translation of SANFLPs into PPNs. While the LAURA tool converts PPN specifications to a hardware implementation[11].

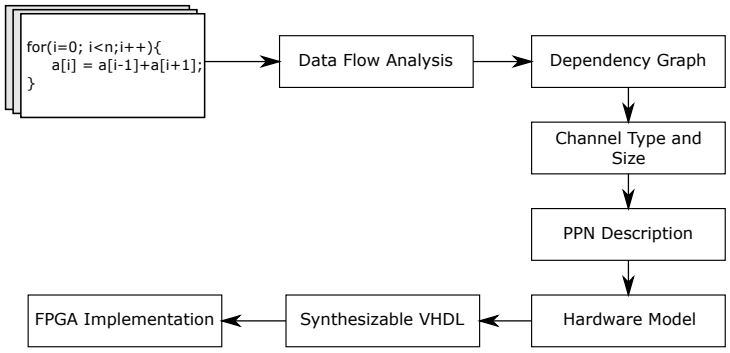


Figure 2.2: PPN synthesis process for a High Level Specification

The process starts with data flow analysis of the C code that translates the specification into a dependency graph. At this point in the synthesis, channel types and sizes are not known. The channel type is determined using a method known as linearization[6].

This process maps the communication type to FIFO channels or reordering channels depending on the producer/consumer relation between processes. The channel information and dependency graphs is used to create a PPN which is taken by the LAURA compiler to generate an abstract hardware model. The abstract hardware model is used to create synthesize VHDL that can be implemented on a FPGA.

2.3.1 Basic Calibration

Each process within a PPN can be correctly modelled and analysed using two metrics: a latency Λ_F and initiation rate $\mathit{\Pi}_F$. Where $\Lambda_F \in \mathbb{N}^+$ is the input-to-output delay and $\mathit{\Pi}_F \in \mathbb{N}^+$ is the initiation interval, both are in clock cycles[1].

The value of Λ_F represents the time between the start of an execution and the end of an execution when an output is created. The initiation rate, $\mathit{\Pi}_F$, helps to determine the throughput of the function as it denotes the time between successive starts of two executions. Figure 2.3 highlights the effect initiation rate has on the throughput of a function. The initiation rate will be assumed to be $\mathit{\Pi}_F = 1$ to enable pipelining of function executions.

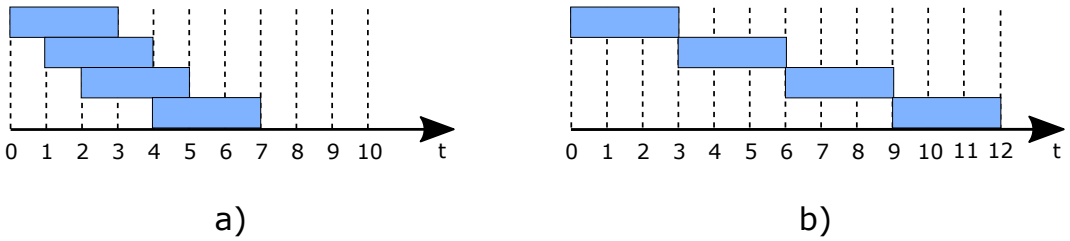


Figure 2.3: Executions of a function with $\Lambda_F = 3$ with (a) $\mathit{\Pi}_F = 1$ and (b) $\mathit{\Pi}_F = \Lambda_F$

The Compaan/LAURA design flow utilizes a process model as shown in Figure 2.1(b). To model a process in Cprof adhering to the LAURA virtual processor model, the read latency (Λ_R) and the write latency (Λ_W) is also required. In the virtual processor successive executions can be pipelined given that $\mathit{\Pi}_F < \Lambda_F$. Figure 2.4 highlights the pipelined nature of a function with a blocking read example that can cause a bubble in the execution pipeline of the process.

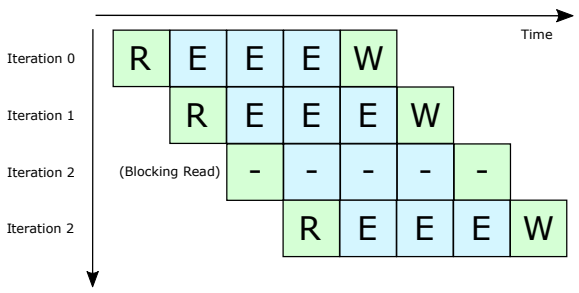


Figure 2.4: Pipeline of an IP Core with $\Lambda_F = 3$ and $\mathit{\Pi}_F = 1$

2.3.2 Types of Communications

The modelling of the program functions as processes means the communication channels between the processes must be defined and classified. During the linearization process, the shared memory variables described in the programs must be mapped to 1-dimensional FIFO units. Linearization requires four types of channel types to accommodate[12].

- In-Order Memory without multiplicity (IOM-) denotes a consumer function which reads in the same order and quantity as the producer.
- In-Order Memory with multiplicity (IOM+) denotes a consumer function which reads the tokens in order, but a token can be read multiple times.
- Out-of-Order Memory without multiplicity (OOM-) denotes a consumer function which reads tokens in an order different from the producer function. Each token is read once.
- Out-of-Order Memory with multiplicity (OOM+) denotes a consumer function which reads tokens in an order different from the producer function. Each token can be read multiple times.

Figure 2.5 demonstrates visually the different types of communication channels in PPNs. The left iteration domain shows the producer iterations and the right iteration domain shows that of the consumer. The arrows highlight how the output of the producer is used in the consumer. The consumer and producer iteration order is shown with blue arrows.

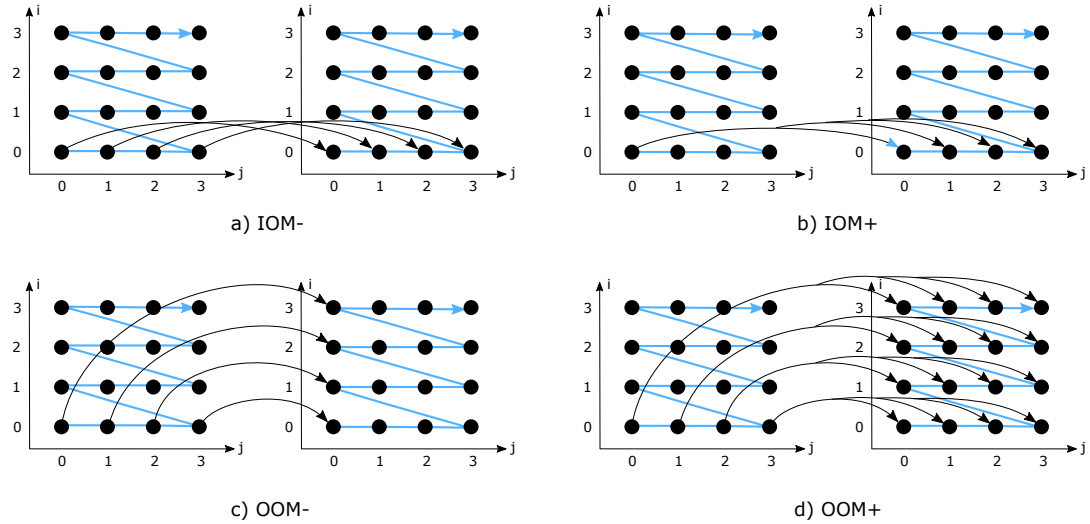


Figure 2.5: Types of Communication Channels in a PPN

2.3.3 Iteration Domain and Dependency

Section 2.1.1 shows the functions of a program in a PPN as a set of linear equations that form a polyhedral. This geometric shape represents the iteration domain of a loop

and the data dependencies between successive iterations. Listing 2.1 demonstrates a SANFLP with a 2-dimensional iteration domain. A mathematical description can be represented in the PPN.

Listing 2.1: Example of a Static Affine Nested For Loop

```
1    for(i=1; i < N; i++){
2        for(j=1; j <M; j++){
3            A[i][j] = foo(A[i][j-1], A[i-1][j]);
4        }
5    }
```

The loop translates to a parametrized rational polyhedron as described in Equation 2.1 where $p \in \mathbb{Q}^d$ [8].

$$P(p) = \{x \in \mathbb{Q}^d | Ax \geq Bp + b\} \tag{2.1}$$

Where d is the dimension of the iteration domain, in the case of Listing 2.1 $d = 2$. With A as an integral mxd matrix, B as an integral mxn matrix and b as an integral vector of size m [8]. Using Equation 2.1 we can describe the SANFLP mathematically.

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

Equation 2.2 demonstrates the iteration domain of Listing 2.1 can be represented mathematically. The polyhedral description can be used to analyse and transform the loops using ILP methods. To perform these transformations and analysis, the dependency between iterations must also be known and this can be visualized in a dependency graph as shown in Figure 2.6

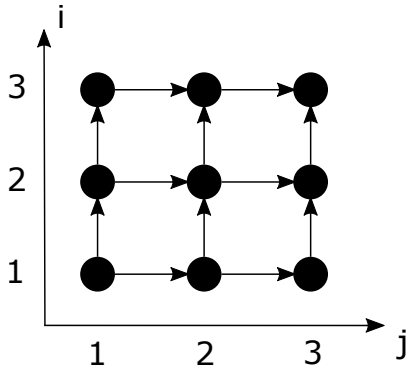


Figure 2.6: Dependencies between iterations of Listing 2.1 with $N = M = 4$

Figure 2.6 demonstrates the dependencies between iterations. The same variable $A[i][j]$ is being written at the end of one iteration but being read at another, creating a self dependency.

2.4 LLVM/Clang

The LLVM and Clang front-end is used to parse the program code to create an Abstract Syntax Tree (AST). This tree can be traversed efficiently and provide all the information necessary to model a PPN and instrument the program code.

LLVM stands for *Low Level Virtual Machine* which is a language independent optimizer and code generator[13]. The goal of LLVM is to build a modular compiler that is platform independent and that can share components across multiple compilers. This allows the designer to select the correct components for the task.

Clang on the other hand is a front-end compiler that parses and creates an AST of the program code. The goal of clang was to develop a front-end that is much more efficient in terms of performance and memory than the standard open source *gcc*. Clang also allows for designers to use components of the tool set. For this work, Cprof utilizes the AST and Parsing tools of Clang to extract the function and variable information. Cprof performs source-to-source transformations on the input programs. Clang utilizes LLVM as back-end to generate the executable[14].

2.5 Cprof

Cprof is an profiler that estimates the performance of a PPN. The profiler can measure the amount of parallelism in the program without deriving the PPN[1]. The original profiler was implemented in the Master Thesis work of Wouter van Teijlingen[4]. The advantage of Cprof is the ability to measure performance of a PPN without deriving it. The performance estimation requires less time than the RTL simulation. Cprof allows for multiple modes of simulation, absolute throughput and unbounded throughput.

Definition 2.5.1. Absolute Throughput. All iterations of a statement are mapped onto a single processing resource with the assumption that an unbounded number of hardware resources are available.

Definition 2.5.2. Unbounded Throughput. Each iteration of a statement is mapped onto a dedicated processing resource with the assumption that an unbounded number of hardware resources are available.

Cprof gives an estimate of the amount of parallelism available. Cprof is also able to give the *maximum degree of parrallelism* which is the measure of simultaneous active processes in the system. While the *average degree of parallelism* measures the average number of active process over the execution of the system. The maximum and average degree of parallelism defines the two extremes of the design space for a PPN.

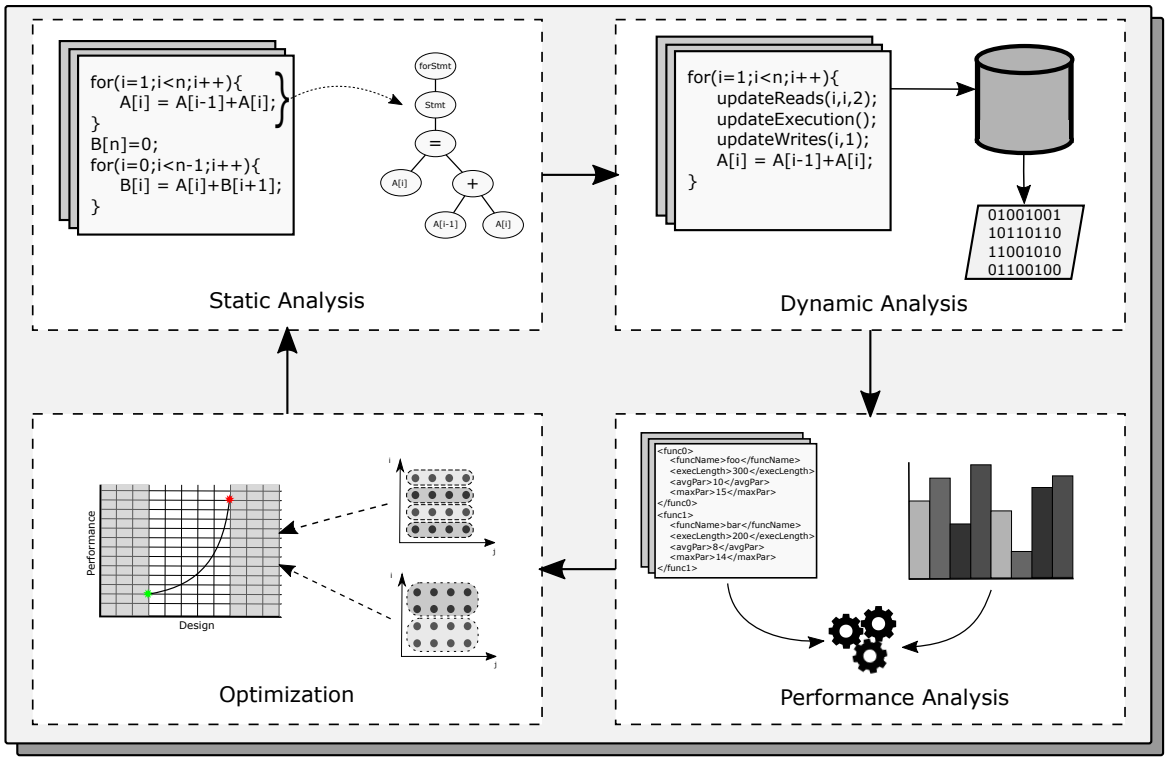


Figure 2.7: The flow of Cprof Execution on a typical program.

Figure 2.7 demonstrates the steps used in Cprof to profile, analyse and optimize a SANFLP. Cprof begins with a static analysis by parsing the program as an AST using the Clang front-end and creates all the necessary objects to represent the variables and statements within the program as a AST. Using this information, Clang inserts instrumentation code around the program statements. An executable of the program is created with this instrumentation and the simulation is run. The results are used to determine where in the program optimization may occur. The final step is to apply the optimizations to the program and repeat the process again and determine the performance gain of the optimizations.

Performance estimation is accomplished using Cprof instrumentation code. The main two concepts used in Cprof to instrument program code to model PPN is the ideas of *Shadow Variables* and *Control Variables*.

2.5.1 Shadow Variables

In Cprof, each variable `v` in the program is given a *shadow variable* `$v` which holds the time that variable `$v` was written. Array variables are handle in a similar way. A shadow array is created for each variable in the program where each index location holds the latest write time of that array index. The shadow variable helps to address the *conditional synchronization* aspect of PPNs. Conditional synchronization is the requirement of a process node to have data ready on all inputs before firing. This means a process may block when reading the inputs if some data is not ready.

Cprof is able to model a PPN without explicitly modelling process nodes and communication channels of a PPN. Unlike sequential programs, PPNs execute concurrently and therefore data dependences are crucial for when a process node fires. Cprof only models *flow dependencies* as performance in a PPN is only affected by *flow dependencies*. Shadow variables are able to capture this behaviour, as they store the latest time-stamp when the variable was written.

2.5.2 Control Variables

Shadow variables are not the only instrumentation needed to estimate the program performance as a PPN. For each statement `s`, a control variable `C$s` is needed to keep track of the earliest time the statement may execute. The earliest time is determined by the conditional synchronization, when all inputs are available and the initiation rate Π_F.

During the read stage, the control variable for `C$s` is updated with the maximum value of all the inputs. The statement `s` can only execute after all the data from its inputs has been read. Taking the maximum of all the inputs allows for simulating of a blocking read advancing the statement time to when all data is ready.

2.5.3 Execution Profiles

Cprof stores the execution behaviour statements and of the program. Statement execution profiles are defined for all three stages in a LAURA process: `R$s[t]`, `E$s[t]` and `W$s[t]` for the read, execute and write stages respectively. For example, `E$s[42] = 4`, denotes 4 concurrent executions in the pipeline of the process at $t = 42$.

Each statement profile is initialized to zero. When an operation is completed for a statement, the profile is incremented by 1 for the range $[t_s, t_f)$, where t_s is the start of the operation and $t_f = t_s + \Lambda$ is the statement finish time. With Λ characterizing the operation latency which can be one of three: read, execute or write depending on the operation.

The statement execution profile contains information such as the start time of the process, which is the first non-zero element of the `R$s` profile. While the finish time can be found by searching for the last non-zero element of `W$s`. The maximum number of concurrent executions can also be found by finding the maximum in `E$s`.

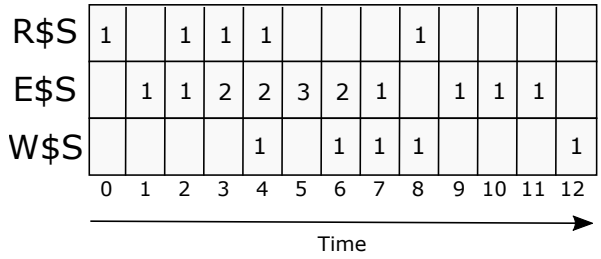


Figure 2.8: An example of the Read, Execute and Write profile of an arbitrary statement with $\Lambda_F = 3$

The `R$s[t]`, `E$s[t]` and `W$s[t]` in Figure 2.8 demonstrate for an arbitrary statement, with $\Lambda_F = 3$, the sequence of reads, executes and writes over time. For example, at $t = 4$ we have the read stage, $R\$S[4] = 1$, denoting the process has 1 active read. At the same time, has a value of 2, indicating 2 operations are passing through the pipeline. The execution profile gives an indication of the pipeline fill as the pattern in Figure 2.8 shows. The pattern shows the first operation at $t = 1$ with $E\$S[1] = 1$. $t = 2$ demonstrates the operation passing through the pipeline as the value is still 1, $E\$[2] = 1$. Reaching a maximum number of operations in the pipeline at $E\$S[5] = 3$ with the pipeline empty at $E\$S[8]$ to indicate no current operations.

The global execution (G$) is another metric that Cprof keeps track of G$ to give an idea of the global behaviour of the program and the maximum amount of parallelism available.

$$G\$[k] = \sum_{i=0}^{|P|-1} R\$i[k] + E\$i[k] + W\$i[k], \quad 0 \leq k < max\{\forall p \in P | f(p)\} \tag{2.3}$$

Where P is the set of all process of the program and $f(p)$ the finish time of a process p as described in the definition from Haagstregt[1]. The global execution profile, `G$`, gives the PPN execution time which is the number of elements in `G$` as well as the *maximum and average degree of parallelism.*

2.5.4 Summary

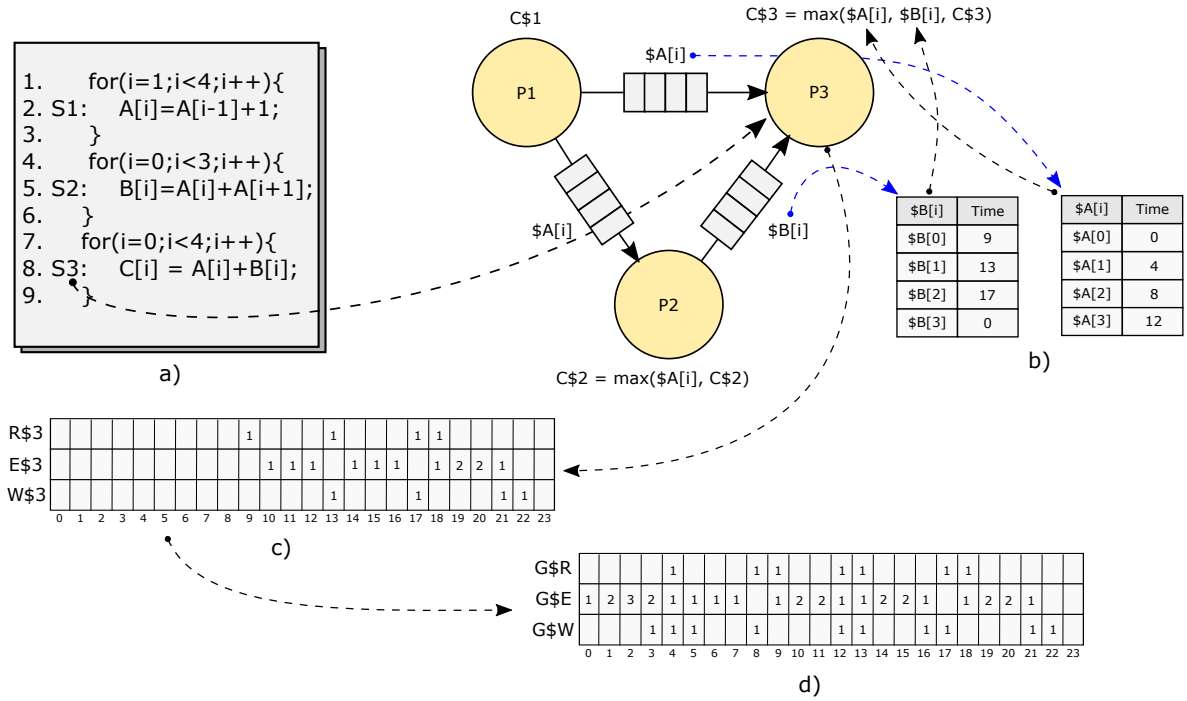


Figure 2.9: Overview of the Cprof profiler mechanics, with $\Lambda_F = 3$ for all processes and $\Lambda_R = \Lambda_W = \Pi_F = 1$

Figure 2.9 summarizes the general operation of Cprof on a SANFLP. Figure 2.9(a) shows a simple example of a three loop program that manipulates array variables A, B and C. The statements in 2.9(a) are modelled as processes in Cprof. As Cprof does not explicitly model the PPN. The arrow from Figure 2.9(a) to 2.9(b) highlights the implicit relation between statement $S3$ and process $P3$.

The execution of $P3$ is regulated using the control variable $C\$3$ and the shadow variables $\$A$ and $\$B$ which are output arguments of $P1$ and $P2$ respectively. Figure 2.9(b) demonstrates the contents of the shadow variables. The process can only execute at the maximum, i.e. when all data inputs are ready and the processor is ready. Take for example iteration $i = 1$ of the loop. The inputs of $P3$ are $\$B[1]$ and $\$A[1]$ which have values of 13 and 4 respectively. The maximum between the two inputs is 13, but before the process may execute $C\$3$ needs to be checked. The previous iteration $i = 0$ of $P3$ started at $t = 9$ and is $\Pi_F = 1$ we find $C\$3 = 10$. The earliest time the process can fire is at $t = 13$ given the maximum of $A\$[1]$,$B\$[1]$ and $C\$3$.

The execution of iteration $i = 1$ for $P3$ is reflected in the statement execution profile in Figure 2.9(c). At $t = 13$, the read profile of $P3$ ($R\$3$) shows a value of 1 indicating there is one active read. The read is followed by the start of an execution at $t = 14$ indicated by $E\$3[14] = 1$. The progress of the execution can be examined in $t = 15$ and $t = 16$ of $E\$3$ as the value is 1 for both times. Indicating 1 active operation in the pipeline. The write operation can be seen at $t = 17$ with $E\$3[17] = 1$ indicating the write stage is writing the output of the process.

The profiles in Figure 2.9(c) also highlight the pipeline ability of the process. Iteration $i = 3$ of $P3$ has input variables $\$A[3] = 12$ and $\$B[3] = 0$. The maximum time on the input variables is 12, but the processor is not ready to fire The control variable, $C\$3$, is 18 at $i = 3$. Taking the maximum of the input variables and the control variable of $P3$ indicate the iteration $i = 3$ of $S3$ can execute at 18. We have $R\$3[17] = 1$ and $R\$3[18] = 1$ indicating two active reads at $t = 17$ and $t = 18$. The consecutive reads translate to two overlapping operations in the pipeline as shown in Figure 2.9(c) at $t = 19$ and $t = 20$ with values of 2.

All profile information is combined into a global execution profile as Figure 2.9(d) demonstrates. The global execution profile is the sum of all read, execute and write profiles of the processes in the PPN as Equation 2.3 Information about the PPN execution time and the maximum degree of parallelism can be obtained. Based on the execution profile, the execution time of the PPN is estimated to be 23 cycles and the maximum degree of parallelism as 3.

3 Related Work

The following chapter will delve into the current world of high level specification profiling for a variety of architectures. The state-of-the art use of profiling and transformations to achieve performance gain in systems will be shown. Section 3.1 will delve into the HLS frameworks while Section 3.2 will investigate in depth the profilers that are often used as a part of these frameworks. Section 3.3 will examine the use of PPN optimization and how it is achieved. Section 3.4 will demonstrate optimizations on high level specifications. While Section 3.5 will discuss the origin of the Cprof profiler and Section 3.6 will summarizes the chapter.

3.1 High Level Synthesis

Converting applications from high level languages to a hardware implementation is not a trivial task. The work by Mazo[15] offers a framework, called MAPS, for such conversion from C to a MPSoC architecture. The frame work aims at reducing the productivity gap between high level exploration and low level implementation. The disadvantage is the MAPS framework supports only the MPSoC architecture and does not support PPN architectures.

The Daedalus[16] is a framework presented by Nikolov et al. as a system level design flow tool for mapping sequential code to MPSoC architectures that uses PPN as a model. The system uses early performance estimations for system-level architectural exploration, to help increase the productivity of the designer. Daedalus show only a 5% error in performance estimations for a JPEG encoder application. Cprof+ aims to profile sequential programs as well and perform design space exploration with source-to-source transformations without knowledge of the target architecture. The disadvantage of the Daedalus framework is that it assumes a MPSoC architecture for the implementation.

3.2 Profilers

The first step in identifying performance bottlenecks in a system is done by simulation and profiling. Execution information can often be obtained using profilers that execute static and dynamic analysis of the application. The following types of profilers exist: memory, software and hardware profilers.

3.2.1 Memory Profilers

Memory profilers identify regions of programs where there is poor temporal or spatial locality of the data and offer ways to improve them.

Cit[17], a gcc-plugin developed in Delft, aims to help parallel programmers by addressing the space of programs which have non-disjoint concurrent accesses which invokes the need for mutual exclusion. The accesses can be classified as Always Conflict or May Conflict. Determining the access type can help increase run-time performance of the program by avoiding the penalty of transactional memory.

StructSlim[18] is another profiler that aims to take advantage of Performance Monitoring Units(PMUs) that are available in most modern processors to sample addresses and their propagation through the execution pipeline. PMUs help to provide an idea of how data is accessed. However, it does require some manual instrumentation of source code and only applies for processors with PMUs.

Both profilers target improvement in the way data is accessed in shared memory systems. The analysis of memory access in a shared memory space is not required, as processes in a PPN communication using a point-to-point method. Cprof+ focuses on memory accesses of point-to-point channels and estimate their size.

3.2.2 Software Profilers

Software profilers examine the affect of programs executed on single or multiple processor architectures. Gprof[19] is a well known general profiler that monitors routine calls in software to estimate bottlenecks in a program. The profiler works by creating a call graph of routines and execution counts by inserting monitoring code into a program. Gprof is not suited for modelling PPNs.

Valgrind is a profiling framework, described as a Dynamic Binary Instrumentation framework[20], which injects instrumentation into the target program during run time. The framework incorporates the concept of shadow values to keep track of a program's variable information. Cprof+ utilizes a shadow variables as well to profile the programs and to keep track of the read and write of processes.

Profilers can also aid the designer in identifying regions of parallelism. Prospector[21], aims to identify areas of parallelism in a program, that are often missed by state-of-the-art compilers. The profiler instruments the code to perform loop profiling and data dependency analysis during run-time. Currently, only targeted for MPSoC architectures, Prospector can be extended to other models such as heterogeneous MPSoC and ASIP designs, but no mention for PPN support is given.

Kismet[22] is a profiling tool for identifying areas of parallelism using Hierarchical Critical Path Analysis (HCPA). The profiler uses HCPA with a target dependent parallelization planner to devise the optimal parallelization strategy. Kismet is able to accurately estimate potential areas of speed-up by partitioning the program into regions and determining a region's self-parallelism. A similar approach is taken in the parallel profiler Parkour[23]. Parkour does not look into region partitioning of a program, rather a finer grain level of functions and loops. Both profilers do not support automated optimization of the program code. The profilers simply identify regions of parallelism for the designer Cprof+ aims to automate the step to allow rapid design space exploration.

A profiling tool that does employ source code transformations is Kremlin[24]. Kremlin detects parallel regions within the program using dynamic execution profile and self parallelism metrics. The information is used to determine an optimal parallelization

plan. Kremlin does support automated optimization of the program. However, the output code is annotated with OpenMP code to facilitate parallelization on MPSoC architectures requiring systems that support OpenMP directives. Support for source-to-source transformations is not implemented as it is in Cprof+.

All the profilers have the ability to describe a program's execution in software and in some cases give optimized code for a given architecture. For applications were source code is translated to hardware, such as PPNs implemented using the LAURA model, a more appropriate profiler is needed.

3.2.3 Hardware Profilers

For a profiler to accurately give performance estimations of the application on a hardware architecture, a basic model is needed. Application Specific Instruction Processors (ASIPs) are one model that are an example of balancing software and hardware design.

Comet[25], is a design flow tool for creating VHDL models of ASIPs from C code. The tool uses a custom instruction set simulator (ISS) to profile an intermediate representation (IR) that is created from the C description. The IR is optimized a number of times using profiling data from the ISS, before it is converted to a VHDL model of the architecture. Comet allows for description of the target architecture, code instructions and memory hierarchy. It does not allow for descriptions other than ASIP and does not perform source-to-source transformations as Cprof+ does.

CoEx[26] is a profiler intended to reduce the productivity gap of ASIP DSE, by allowing for rapid iteration in the software domain. It uses a multi grained profiler approach by finding program hotspots and then tracing variable histories and memory accesses within the hotspots to give the designer a better idea for optimization while leveraging performance costs of basic blocks. The disadvantage is the profiler assumes an ASIP architecture and does not support others like PPNs.

In the paper of J.F. Eusse et al.[27] pre-architectural performance estimation on abstract processor models is used to reduce the productivity gap. The estimation is achieved with static analysis of LLVM-IR combined with a dynamic analysis using the CoEx[26] profiler to instrument source code producing a semi-static analysis technique. Currently, the tool is limited to data-path optimizations of ASIPs and does not look at memory architectures of the custom processors.

TotalProf[28], is a source code profiler that aims to deliver a tool that can estimate performance of a program on many different architectures. The profiler works by creating a virtual compiler backend in LLVM to emulate the execution on the target architecture. TotalProf can estimate performance of VLIW, MPSoC and ASIP architectures with an error rate of 5% to 15%, but support for PPN architectures is not implemented.

None of the profilers address the need for rapid exploration of PPN designs which utilize the LAURA model. Most profilers and HLS frameworks target a specific implementation model. Cprof+ aims address the lack of profilers targeted at PPNs and PPN optimization.

3.3 Polyhedral Process Network Optimization

To improve the productivity of the designer, tools may be created to reduce DSE or ease the exploration. These tools require transformations of the source code or the system description to achieve design exploration.

CLooG[29], a tool used for source-to-polydedra-to-source transformations, can apply transformations to source code in an analytical manner for PPNs. With speedups of 4.05 in terms of run time as compared to other code generators, CLooG is a useful tool[29]. However, it does not allow for automated DSE of programs and the inherent intensive memory usage and time complexity hinder rapid DSE.

Polly[30] attempts to address the issue of Polyhedral transformations by manipulating the Intermediate Representation (IR) of programs as Polytopes using LLVM. Polly supports a range of transformations. It does not support source-to-source transformations, as the transformations are applied only to the the LLVM-IR description and then converted to an executable.

Loopy[31], performs loop transformations with verification using Polly[30]. It does not allow for automatic optimization, as the designer must specify the transformations to be used. The tool only looks at GPP applications and not hardware. Loopy achieved significant speed up of programs with all the transformation scripts for each program of the Polybench suite. The disadvantage is the optimization is not automated. The transformation scripts for all benchmarks were written in one week[31].

So far compile time optimizations of PPNs have been discussed. Runtime optimization of a PPN using the process splitting technique has also been proposed. Meloni et al.[32] propose a runtime profiler that detects when a process can be optimized using the process splitting method. The detection is achieved using an execution table located in shared memory, whereby a helper process is able to scan the table and select the correct process for splitting. The results demonstrate for the MPJEG case study, the performance gain was negligible due to the overhead costs of the decision, splitting and merging phases.

3.4 Optimizing High Level Specifications for Hardware

HLS today is standard practice for many applications due to size and complexity of the problem. Each modern HLS tool often uses a compiler to generate the HDL. The compilers use optimizations on the code such as dead code elimination and constant propagation. Huang et al.[33] present a HLS-directed compiler optimization for FPGA implementations By investigating different compiler optimization techniques and compiler passing directions, Huang et al. were able to present a set of compiler optimizations that in HLS produce on average 16% performance increase[33]. The paper did not suggest if the these optimizations can be applied to other hardware architectures, but it does highlight the importance of source transformations to induce performance in hardware.

Accelerating kernel loops of programs in reconfigurable hardware at the high-level language level using Loop Skewing and Loop Unrolling is another problem. The paper[34] took a look a data dependencies between iterations to determine whether

to unroll a loop. Determining the unroll factor based on memory accesses and area estimations as well as comparing estimated software and hardware implementations.

3.5 Cprof

Cprof is a lightweight profiling tool that estimates the performance of a PPN based on an application source code. The concept was conceived by Haagstregt[1] in his PhD thesis and suggested the tool for profiler PPNs. The profiler does not explicitly model the PPN, rather it constructs a simulation based on the statements of a program. The implementation of the tool was done in the master thesis of Teijlingen[4]. The results of Cprof demonstrated for most programs in the Polybench Suite, accurate performance estimations can be achieved. However, for some benchmarks the estimation was up to 60% inaccurate[5].

3.6 Summary and Conclusion

Improving source code for applications is fast under way for MPSoC and ASIP architectures to enable better designs. Each employing frameworks with a range of profiler types to minimize the overhead for profiling thereby increasing the productivity of the designer. Some frameworks even allow for automated DSE to generate optimal implementations.

Work towards a rapid profiler for programs implemented in a PPN has been achieved using Cprof. The previous work of Teijlingen[4] gave an efficient profiler that can be used to help explore the DSE of a PPN rapidly using the Compaan/Laura tool chain. Inaccuracies and lack of features hinder the ability for widespread use as an early prototyping tool for hardware PPNs implementations. We present Cprof+ that will help to reduce the gap of the profiler as a stable and lightweight tool for profiling PPNs.

4 Cprof Validation

The ultimate goal for Cprof+ is to provide the designer a tool that can find design points efficiently to assist in DSE. The design points given by Cprof+ can aid the designer in finding the optimum design for their application. Finding appropriate design points in the design space requires Cprof+ to maintain an accurate execution profile of the network.

With an accurate execution profiles, Cprof+ will be able to estimate the performance of a network comparable to that of the hardware performance. The Cprof+ performance estimations will be compared to the RTL simulations of the PPN using the Vivado HLS Suite to ensure their validity.

4.1 Problem

The master thesis of Teijlingen[4] showed that Cprof estimates the performance of most Polybench programs accurately. Table 4.1 demonstrates for 22 of the 29 benchmarks in the Polybench Suite the performance estimations are well within 1% of the RTL simulation values. There are exceptions which include the benchmarks: *dynprog, fdtd_ampl, floyd_warshall, gemm, lu, reg_detect and syr2k*. Each exhibit a difference between the Cprof estimation and RTL simulation ranging from 5% to 60%.

Verification was performed on the benchmarks and the following problems were identified.

1. **Static functional latency.** The execute stages of processes in Cprof are modelled as a 3 stage execution pipelines, no matter the complexity of the function expression implemented.

2. **Channel communication type.** Process communication channels in Cprof are not modelled. The communication channel types are not known a priori and no distinction is made between a IOM and OOM channel.

3. **Static channel type.** When a process in Cprof is reading an input, the channel type is always static and does not change during run time. Static channel types are not guaranteed for Cprof. As Cprof simplifies the communication topology of the PPN.

Benchmark	Cprof (Cycles)	Vivado (Cycles)	Difference (%)
adi	5590	5606	0.29
atax	19948	19963	0.075
bicg	1989	719911	0.070
cholesky	90552	91604	1.14
correlation	337392	338579	0.35
covariance	340876	342014	0.33
doitgen	181028	181047	0.010
durbin	10743	10909	1.52
dynprog	187874	253593	25.92
fdtd_2d	2125	2140	0.70
fdtd_apml	4412	4609	4.27
floyd_warshall	746	2086	64.24
gemm	33762	99251	65.98
gemver	54972	55050	0.14
gramschmidt	4510	4721	4.47
jacobi_1d_imper	1031	1045	1.34
jacobi_2d_imper	1938	1951	0.67
lu	10743	31571	65.97
ludcmp	122656	122773	0.095
mvt	10649	10724	0.70
mm2	519096	521222	0.41
mm3	627683	631730	0.64
reg_detect	3649	4571	20.17
seidel_2d	181228	181242	0.0077
symm	279435	281367	0.69
syrk	508958	508979	0.0041
syr2k	1016878	108038	25.88
trisolv	9497	9518	0.22
trmm	418625	420323	0.40

Table 4.1: Comparison of the original performance estimations of Cprof against Vivado RTL Simulation results.

4.2 Solution Approach

To solve these 3 problems we propose 3 solutions. The first solution introduces the capability for variable pipeline depths. The second solution implements a run time check for channel communication type. The third solution demonstrates dynamic channel type switching during run time.

4.2.1 Variable Pipeline Depths

The original design of Cprof implements the pipeline depth of a function as a constant. Where the function latency (Λ_F) for any function is set to 3 cycles with a read latency (Λ_R) and write latency (Λ_W) of 1 cycle. A functional latency of 3 cycles is not the case

for many functions.

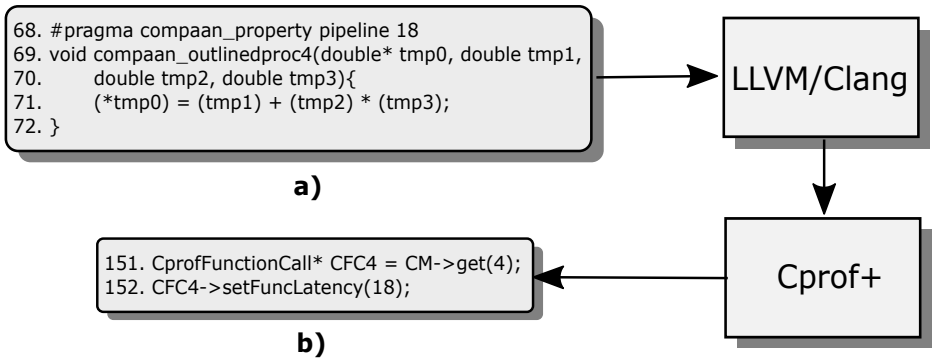


Figure 4.1: Typical Flow for Pipeline Depth Detection. a) original source code of atax.c b) instrumented source code of ATAX, atax.c_cprof.cpp, with Cprof Instrumentation statements inserted.

Figure 4.1 demonstrates a function from the *atax* benchmark with a functional latency of 18 cycles. The functional latency or pipeline depth was calculated by Compaan using Vivado HLS 2015.4 from Xilinix[5], for each function within the Polybench suite. The original Cprof did not take the calculated pipeline depths into consideration even though the infrastructure existed.

For Cprof+ to include the variable pipeline depths, the LLVM/Clang was modified to include a pragma handler to catch the `#pragma compaan_property pipeline` on a function. The pipeline depth is then passed to the CprofFunctionCall object representing the function call in Cprof+. The function latency is also saved directly to the instrumented program file as shown in Figure 4.1(b). Cprof+ can adjust firing times of the node accordingly and simulate the actual execution of the node in the network. This allows for more accurate measurement of the network performance.

4.2.2 IOM and OOM Communication

Communication between PPN nodes take the form of IOM and OOM with multiplicity or without. For most cases we can see in Table 4.1 the type of communication is FIFO based (IOM-) as Cprof modelled single latency reads. Benchmarks like *gemm* shows the read latency can have an affect on the performance estimation.

Figure 4.2 shows the PPN for *gemm* with streaming nodes, ND_2 and ND_3. Both nodes communicate using Out of Order Memory to ND_7. As the original Cprof cannot distinguish between communication types, tt assumes a constant cycle cost of 1 cycle for each read. A read latency of 1 is not the case for Out of Order Memory. By examining the RTL simulations, generated by the Compaan/LAURA tool, we observe that it takes 3 cycles to complete a read from an OOM communication type. This is further support by the HDL description of the OOM.

The HDL description shows the Finite State Machine (FSM) used. It does indeed have three states and for one complete read it must pass through each state. Hence the delay of 3 cycles when a token is read from OOM.

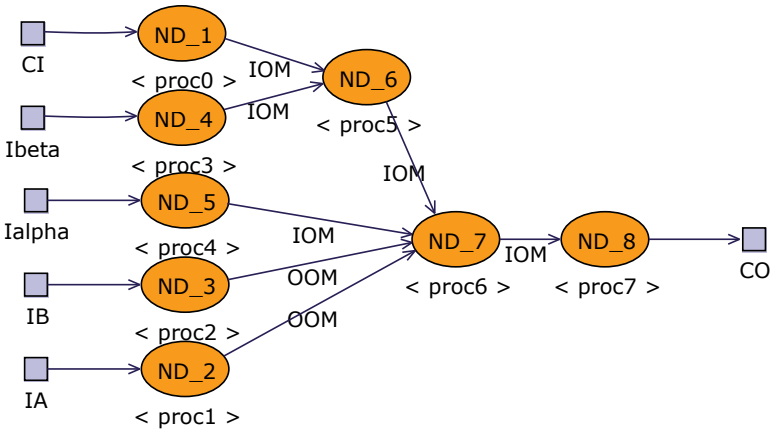


Figure 4.2: The PPN graph of the benchmark GEMM. With OOM edges: ND_3 to ND_7 and ND_2 to ND_7

Listing 4.1: The Finite State Machine used in the RTL description to handle the read from an OOM

```
1  FSM: process(s_clk, s_rst)
2      variable cntr: natural;
3  begin if rising_edge(s_clk) then
4      if(s_rst='1') then
5          state <= s_idle;
6      else
7          case (state) is
8              when s_idle => if (read_variable='1') then state <= s_load;
                   end if;
9              when s_load => if (FSL_S_Read='1') then state <= s_update;
                   end if;
10             when s_update => state <= s_idle;
11             when others => state <= s_idle;
12         end case;
13     end if;
14 end if;
15 end process;
```

The next step is to implement this behaviour within Cprof+. This requires the detection of an OOM channel. OOM detection is not a trivial task because the original implementation for communicating variable write times between functions in Cprof was done using Shadow Variables, which act as a simple Look Up Table (LUT). When the function is executed, the indices are used to check the latest write time of the variable in the shadow variable. The shadow variable method does allow for modelling of flow dependencies in a PPN, but it does not have the capability to model the communication channel type.

To identify the types of communication, we introduce the concept of Lexicographic Order.

Definition 4.2.1. Lexicographic Order. An element a is lexicographically less than an element b if $a_i < b_i$ for the first dimension i in which the elements differ [1]. Or more concisely,

$$a \prec b \equiv \bigvee_{i=1}^{n} (a_i < b_i \wedge \bigwedge_{j=1}^{i-1} a_j = b_j) \tag{4.1}$$

For example, let $a = (1, 2, 1)$ and $b = (1, 2, 2)$. First, the left most index ($i = 0$) is checked and it is found that the element is equal, e.g. 1. Then the middle indexes ($i = 1$) are checked and again are found to be equal, e.g. 2. It is only with the last index ($i = 2$), do the elements differ. With $a_3 < b_3$ we find that $a \prec b$.

The idea of lexicographical order is utilized through out the OOM Detection algorithm. A read is checked for OOM using the lexicographical order test on variety of metrics that are stored within Cprof. Section 4.2.2.1 will explain the infrastructure needed for the OOM detection algorithm. The algorithm can be partitioned into 3 main parts. Figure 4.3 shows the control flow of the program with 3 main parts highlighted: A.1, A.2 and A.3.

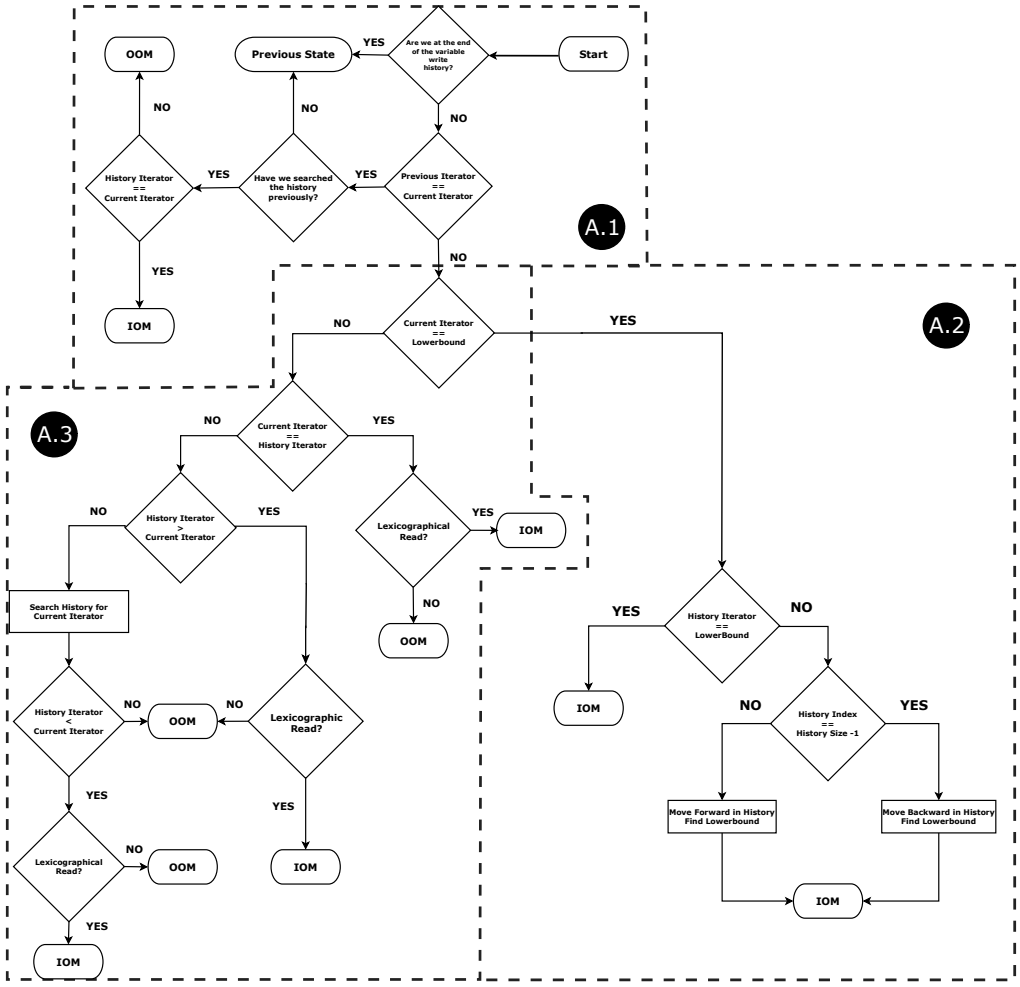


Figure 4.3: The control flow of the OOM detection algorithm

4.2.2.1 OOM Detection Infrastructure

The original Cprof cannot model channel communication types. Cprof lacked the infrastructure to detect an OOM channel. When a variable is read, the channel type can be determined by maintaining four metrics.

1. Current value of the read iterators. Cprof+ will need to know which iteration the loop is in to understand if the current read operation is in-order or out-of-order.

2. A variable's write history. The original Cprof did not maintain a history of variable write times and write indices. The read order of variables are assumed to traverse the iteration domain in lexicographical order. The lexicographic order of the read compared to the current write in the history can be used to determine if a channel is OOM. If the lexicographical order is negative, $read \succ write$, then the read is OOM.

3. The previous value of the iteration. Keeping track of the previous iteration can help to determine if the variable is being read in a lexicographic order. If $read \prec read_{prev}$ then the variable is reading out of order. The result can help to determine if the channel is indeed out of order.

4. Lower and upper bound of the iteration values. The original Cprof lack the ability to know during run time, the minimum and maximum values the iterators will take during the loop execution. Cprof+ is required to know the maximum and minimum values of the iterators as it can indicate if the loop is beginning or ending a pattern of reads.

4.2.2.2 A.1 OOM Detection

The first part of the algorithm checks whether we have reached the end of the variable's write history. The algorithm works by incrementing the history's array index and examining the write at that index. The array index is incremented during each read operation of the process. If during a read, we find the write history index has reached the end of the array, it can signal two outcomes. The channel may have multiplicity, meaning tokens are being read multiple times, or due to parts A.2 and A.3 we have advanced the write history to the end. In each outcome, the previous iteration's channel type is used to classify the current channel type. The assumption is the channel type will not change, if there are no more tokens to read from the write history.

If the write history has not reached the end, the previous read iterators are used to check the current read iterators. If the read iterators are equal to the previous, it can indicate we are reading in order with multiplicity or out-order with multiplicity.

To determine whether it is IOM or OOM, we check if the write history was searched. The search is carried out by A.2 or A.3 of the algorithm. If the history was not searched, we maintain the previous channel type. The search could indicate the channel is reading in IOM or OOM depending on the result of the search. We need to confirm the result with the write history. If the write iterators match the current read iterators, then it is IOM. If not, it is OOM.

4.2.2.3 A.2 OOM Detection

Beginning the loop at the initial iterator value can present a problem for detecting the channel type. There are cases within the Polybench suite were benchmarks have variables that do not reach the loop maximum or minimum condition which is caused by dynamic initial loop conditions.

Listing 4.2: Source Code of Lu. An example of dynamic loop conditions

```
1  for (i = 0; i < _PB_N; i++) {
2      for (j = 0; j < _PB_N; j++) {
3          proc0( &A[i][j],AI[i][j] );
4      }
5  }
6  for (k = 0; k < _PB_N; k++) {
7      for (j = k + 1; j < _PB_N; j++) {
8          proc1( &A[k][j],A[k][j],A[k][k]);
9      }
10     for(i = k + 1; i < _PB_N; i++) {
11         for (j = k + 1; j < _PB_N; j++) {
12             proc2( &A[i][j],A[i][j],A[i][k],A[k][j]);
13         }
14     }
15 }
16 ...
```

Table 4.2: Showing the write and read iterator histories of variable A for *lu*'s proc2 with channel types as detected by Cprof+. The first 6 iterations.

Write History		Read History (Proc2)					
(Proc1) $A[k][j]$	(Proc0) $A[i][j]$	$A[i][k]$	Type (Proc)	$A[k][j]$	Type (Proc)	$A[k][j]$	Type (Proc)
(1,0)	(0,0)	(1,1)	IOM(2)	(1,0)	IOM(2)	(0,1)	IOM(1)
(1,1)	(0,1)	(1,2)	IOM(2)	(1,0)	IOM(2)	(0,2)	IOM(1)
(1,2)	(0,2)	(2,0)	IOM(2)	(2,0)	OOM(2)	(0,0)	OOM(1)
(2,0)	-	(2,1)	IOM(2)	(2,0)	OOM(2)	(0,1)	OOM(1)
(2,2)	-	-	-	-	-	-	-
...	-	...	...	...	...	...	...

Table 4.2 shows the first 5 iterations of function `proc2` from the benchmark *lu* shown in Listing 4.2. The two left columns of the table show the write history of the variable A as seen from two functions, `proc0` and `proc1`. The right columns of Table 4.2 highlights the read iterations of `proc2`.

The first read iteration of `proc2` has $(1, 1)$ for variable $A[k][j]$. The lower bound on the iterator values for $A[k][j]$ is calculated to be $(1, 1)$ as both k and i have initial loop conditions of 1 for this iteration. The variable $A[k][j]$ is IOM as calculated by Compaan. If the write history of `proc2` were compared to the initial read of $A[k][j]$ it would be a mismatch. As $A[k][j] = (1, 1)$ and $A[k][j] = (1, 0)$ for `proc2` and `proc1` respectively. The history of $A[k][j]$ from `proc1` will have to be incremented to $(1, 1)$.

As the read history of $A[k][j]$ of `proc2` is in lexicographical order. For subsequent reads of $A[k][j]$ of `proc2` will follow the write history of `proc1`. This is shown in Table 4.2 with the subsequent reads of $A[k][j]$ described as IOM.

4.2.2.4 A.3 OOM Detection

The third part of the algorithm takes a look that value of the iterators in the write history to verify the channel type. If the current iterator values are equal to that in the history, then it must signify we are reading in step with what was written. To ensure this assumption, the previous read iterator values are checked. The check is done by examining if the current read is lexicographical positive and offset by 1. This is accomplished by comparing the previous iteration with the current iteration to see if the distance between the two are 1. If both conditions are not met, the channel is classified as OOM.

There are instances where the value of the write history does not match the current iteration. If the history iterator values are lexicographically greater than the current iterators, it may mean we are reading in order. Only if the lexicographical read order is positive and current iteration is ahead of the previous iteration by one. When the read order is lexicographically negative the channel is classified as an OOM.

For example, take the iterators (i, j) where we assume the minimum and maximum values of (i, j) to be $(0, 0)$ and $(2, 2)$ respectively. Take the current iteration value as $(2, 0)$. If the previous iteration value was $(1, 0)$ then the read is lexicographically positive, but the distance between the two points is greater than one using lexicographical read order. Meaning the current iteration is not an in order read.

An example of an in order read, is where the current iteration, $(2, 0)$, is both lexicographically positive and succeeds the previous iteration by 1. In this case $(1, 1)$.

The lexicographical read check can also be used when the history iterator values are lexicographically less than the current iterator values. In this case the history may be a number of iterator values behind the current read iterator values and needs to be advanced. If the current iterator values is found within the history, then it is checked to see if the read order is lexicographical to determine if the channel is IOM or OOM. If the iterator values are not found, then it is assumed to be OOM as the iterator may lie in the past.

Listing 4.3: Source Code of GEMM

```
1  for (i = 0; i < N; i++) {
2      for (j = 0; j < N; j++) {
3          proc1(&A[i][j],IA[i][j] );
4      }
5  }
6  for (i = 0; i < N; i++) {
7      for (j = 0; j < N; j++) {
8          proc2(&B[i][j],IB[i][j] );
9      }
10 }
11 ...
12 for (i = 0; i < N; i++) {
13     for (j = 0; j < N; j++) {
```

```
14      proc5( &C[i][j],C[i][j],beta );
15          for (k = 0; k < N; ++k) {
16              proc6(&C1[i][j],C[i][j],alpha,A[i][k],B[k][j]);
17          }
18      }
19  }
```

Table 4.3: Showing the write and read iterator histories of variables A and B of *gemm*'s compaan_outlinedproc6 with channel types as detected by Cprof+. The first 12 iterations.

Write History		Read History				
A	B	Iteration	A[i][k]	A_{type}	B[k][[j]	B_{type}
(0,0)	(0,0)	(0,0,0)	(0,0)	IOM	(0,0)	IOM
(0,1)	(0,1)	(0,0,1)	(0,1)	IOM	(1,0)	OOM
(0,2)	(0,2)	(0,0,2)	(0,2)	IOM	(2,0)	OOM
(1,0)	(1,0)	(0,1,0)	(0,0)	OOM	(0,1)	OOM
(1,1)	(1,1)	(0,1,1)	(0,1)	OOM	(1,1)	OOM
(1,2)	(1,2)	(0,1,2)	(0,2)	OOM	(2,1)	OOM
(2,0)	(2,0)	(0,2,0)	(0,0)	OOM	(0,2)	OOM
(2,1)	(2,1)	(0,2,1)	(0,1)	OOM	(1,2)	OOM
(2,2)	(2,2)	(0,2,2)	(0,2)	OOM	(2,2)	OOM

The benchmark *gemm* highlights the third part of the algorithm. The source code is given in Listing 4.3. The read iterations of `proc6` in *gemm* are given in Table 4.3. The variable $B[k][j]$ of `proc6` is an example of an out of order channel. The initial iteration is classified as IOM because the value in the write history of B, $(0,0)$, matches that of $B[j][k]$ in `proc6`. The in-order read does not hold for the next iteration of `proc6`. The read value of B is $(1,0)$, but the value in the history is now $(0,1)$. The current iterators are lexicographically greater than the write history and the read is not following the lexicographic order. The read of iteration (0,0,1) for `proc6` is OOM.

4.2.2.5 Summary

Figure 4.3 demonstrates a run time check for detecting if a read is OOM. The accuracy of the check is not 100% as Tables 4.3 and 4.2 show. The variable B of `proc6` in Table 4.3 is an OOM channel. Cprof+ cannot determine the channel is OOM for the first iteration because it lacks information on the read and write history creating an inaccuracy. The inaccuracy can also be seen in Table 4.2 for variable $A[k][j]$ of `proc2`. The first two iterations are indeed following the lexicographical read order and the write history. The error is not correct till the lexicographical read order is broken.

The algorithm may not be 100% accurate, but as the results in Section 4.3 demonstrate, the run-time check for OOM channels can handle most Polybench benchmarks to give an accurate estimation of performance.

4.2.3 Dynamic Channel Type Detection

Detecting the communication type is crucial to achieve a more accurate tool for profiling SANFLPs. Simply assuming that when OOM is detected, that the channel is consistently OOM is not always true. As explained in Section 5.1.1, Cprof+ cannot model all the edges present between nodes explicitly. Cprof+ only examines the relations between variables and derives the channels from the write and read profiles, rather than from a mathematical description of the input and output relation.

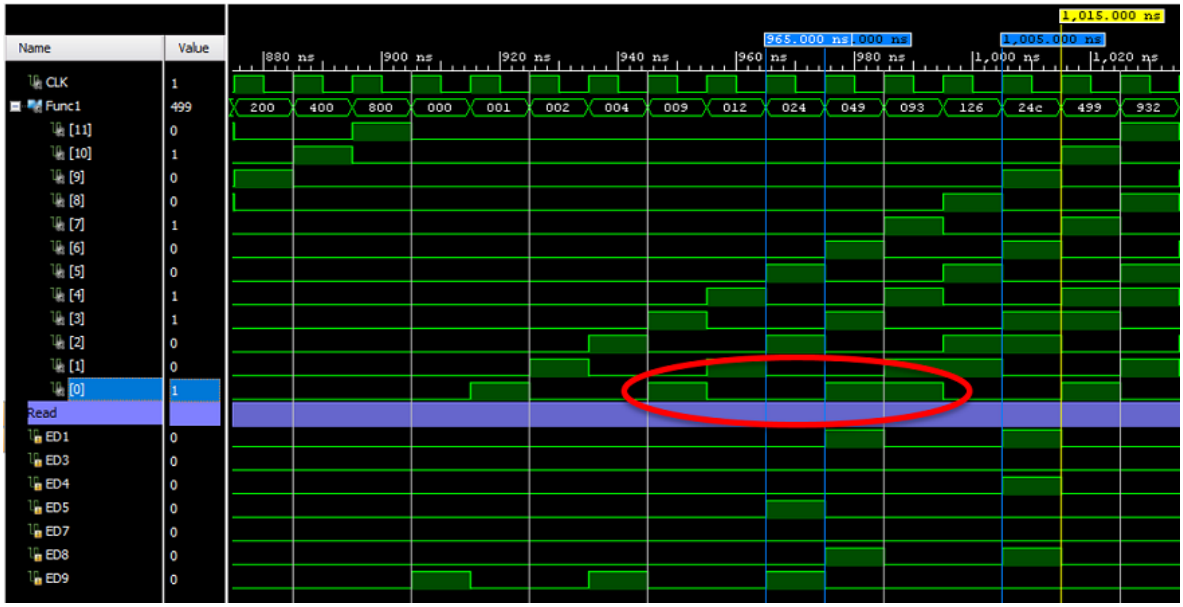


Figure 4.4: RTL Simulation of the Floyd Warshall Benchmark. Showcasing the execution of ND_2 (*Compaan Calculation Function*) and the IOM and OOM nature by highlighting the read signals from the self loop edges.

As Figure 4.4 highlights, during the execution of the Floyd Warshall benchmark, depending from which channel the token is read, it will incur a read penalty due to a read from an OOM channel. In some instances, there is no read penalty incurred because two tokens are read in sequence from two different IOM channels. Causing two operations to execute in sequence as highlight in Figure 4.4

The PPN graph of the Floyd Warshall bechmark in Figure 4.5 demonstrates the complex nature of the self loops of Node 2. Node 2 has 7 separate self loops when the variables are read, the self loop that is selected is based on the schedule of the network. The schedule is derived mathematically. In the case of the Floyd Warshall benchmark, the iteration domain is divided to accommodate for 7 different access patterns. These access patterns are manifested as 7 self loops.

Cprof+ can only examine the high-level specification, inferring the communication channels by examining the relation between variable input and output. Figure 4.6 highlights the function of node 2 with 3 input variables and 1 output variable, all of the same name. Given this variable information, Cprof+ can only model 3 self loops. Leading to a model where 4 out of the 7 edges are not recognized. The Cprof+ model

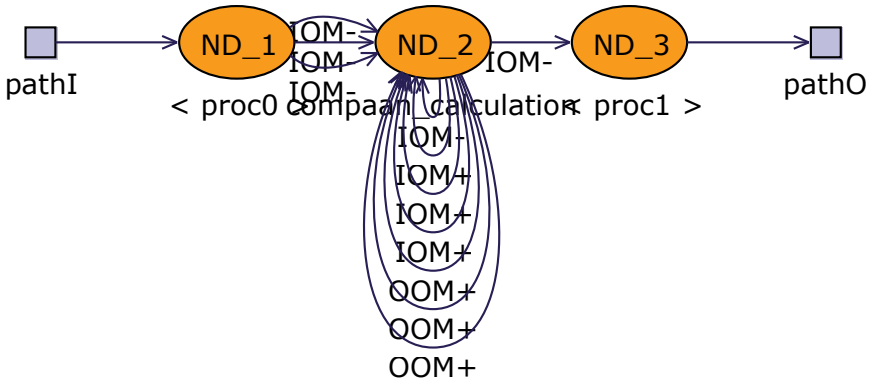


Figure 4.5: PPN of the Floyd Warshall Benchmark.

```
1     for (k = 0; k < _PB_N; k++) {
2         for(i = 0; i < _PB_N; i++) {
3             for (j = 0; j < _PB_N; j++) {
4             path[i][j] = compaan_calculation(path[i][j], path[i][k], path
                  [k][j]);
5             }
6         }
7     }
```

Figure 4.6: Function Description for ND_2 in the KPN of the Floyd Warshall Benchmark

of the PPN for the Floyd Warshall benchmark can be seen as a simplification of the actual model.

The simplified model can lead to a situation where two tokens are read in Cprof+ in sequence from the same Cprof+ communication channel. When in the true PPN model, the tokens are being read from two different communication channels that may have different types. The solution is to have Cprof+ constantly checking during run time for the communication type.

4.3 Results

The results of the modifications demonstrate a much more accurate estimation of the actual hardware implementation of the programs in a PPN. Table 4.4 compares the RTL Vivado simulation results to the estimations given by Cprof+. The maximum error of the simulation results is now brought to the range of -2.20% to 5.88% for the Polybench suite of programs. A much more accurate result when compared to the original simulation results which saw a maximum error of 66% error as shown in Table 4.1. Errors for some benchmarks still exist, e.g. *floyd_warshall* and *syr2k* show an error of 4.12% and 5.88% respectively. The source of the errors is further discussed in the following section.

Benchmark	Cprof (Cycles)	Vivado (Cycles)	Difference (%)
adi	5590	5606	0.18
atax	19948	19963	0.08
bicg	19897	719911	0.07
cholesky	91500	91604	0.11
correlation	338498	338579	0.02
covariance	341992	342014	0.01
doitgen	181028	181047	0.01
durbin	10803	10909	0.97
dynprog	242398	241801	-0.25
fdtd_2d	2187	2140	-2.2
fdtd_apml	4508	4609	2.19
floyd_warshall	2000	2086	4.12
gemm	99172	99251	0.08
gemver	55034	55050	0.03
gramschmidt	4746	4721	-0.53
jacobi_1d_imper	1031	1045	1.34
jacobi_2d_imper	1899	1951	2.67
lu	31364	31571	0.66
ludcmp	123680	122773	-0.74
mvt	10711	10724	0.12
mm2	521142	521204	0.00
mm3	631713	631730	0.00
reg_detect	3649	4571	0.52
seidel_2d	181228	181242	0.01
symm	281351	281367	0.01
syrk	508958	508979	0.00
syr2k	1016878	108038	5.88
trisolv	9497	9518	0.22
trmm	418625	420323	0.40

Table 4.4: Comparison of Cprof+ with OOM detection against Vivado RTL Simulation results.

4.4 Limitations

As Table 4.4 demonstrates for two of the benchmarks, *floyd-warshall* and *syr2k*, the error is still substantial. Section 4.2.2.5 highlights the ability to check for a channel type during run time can lead to a run in error. Cprof+ cannot detect with little or no information that a channel is OOM.

The source of the Cprof+ estimation errors can be categorized in two parts.

1. **Channel State**. As part A.1 of Figure 4.3 demonstrates, when the algorithm has reached the end of the write history, the previous channel type is used. The previous channel type may in some cases not match the current channel type.

2. **Initial Channel Type**. Finding the initial channel type is difficult with no read

history. As Section 4.2.2.5 demonstrates for some benchmarks a initial error is encountered due to the lack of information and read pattern. This initial error can lead to both over and underestimations in the case of OOM or IOM respectively.

Further investigation into the inaccuracies will have to be conducted in order to obtain a more stable estimation of the PPN. Cprof+ shows that with run time OOM detection estimations can be signifigantly reduced for benchmarks such as *lu* which rely on OOM channels.

The ability to detect channel type is not a trivial one. ILP methods used previously in Compaan using Multiplicity Test and Reordering Test which are memory intensive and time consuming[35]. At the current moment, a polynomial technique is employed to detect types in Compaan using the mapping matrix. The advantage of Cprof+ is the ability to detect the type just using the execution profiles during run time. The comparison of run times of Cprof+ vs. Compaan/LAURA design flow is shown in Chapter 7.

5 Channel Sizing using Cprof+

The calculation of channel sizes for communication channels in a PPN is to choose channel sizes such that a network does not deadlock. Often times heuristics [15]or simulation runs[36] are used to calculate appropriate channel sizes with varying results. A deadlock in a PPN occurs when the network cannot progress forward due to one or more function nodes blocking on a write operation.

One solution is to select arbitrary large channel sizes to ensure that deadlocks do not occur in the PPN. This approach is inefficient in terms of hardware used, as it increases memory costs unnecessarily.

In the thesis of Haastregt[1] buffer size computation is performed by PNGen[37]. PNGen takes the global schedule of the network and determines the channel sizes based on this schedule [1]. The schedule provides a relative order of iteration pairs. Given a read and write iteration pair with the number of reads so far and the number of writes before, the size of communication channel can be calculated. The maximum size is determined by examining all sizes over the whole set of iteration pairs for that channel.

Examining the iteration pairs is not the only approach. When the PPN is non-parametric, the channel size can be calculated using a symbolic approach by computing the maximum on a quasi-polynomial. PNGen employs both techniques to find a set of channel sizes that guarantee a deadlock free schedule. This is done by using a greedy algorithm for computation of channel size.

Cprof+ utilizes the execution profiles of each function node available to determine the channel size. A similar approach to that of PNgen.

5.1 Solution Approach

The solution for calculating the sizes of channels is to model the channels explicitly. Section 5.1.1 explains the explicit definition of communication channels and how they are implemented in Cprof. Section 5.1.2 will demonstrate the method used to calculate the channel size based on the write and read history of that channel.

5.1.1 Modelling Communication in Cprof+

The current method of passing variable information between function nodes in Cprof is done through the Shadow Variables. Similar to a LUT, the table is used to store the write time of a variable. This method of communication is extremely effective for performance modelling of the PPN. The original method of modelling does not accommodate for channel size estimation. The following section will discuss the CprofFIFO object, that will be used to represent the communication channels in Cprof+.

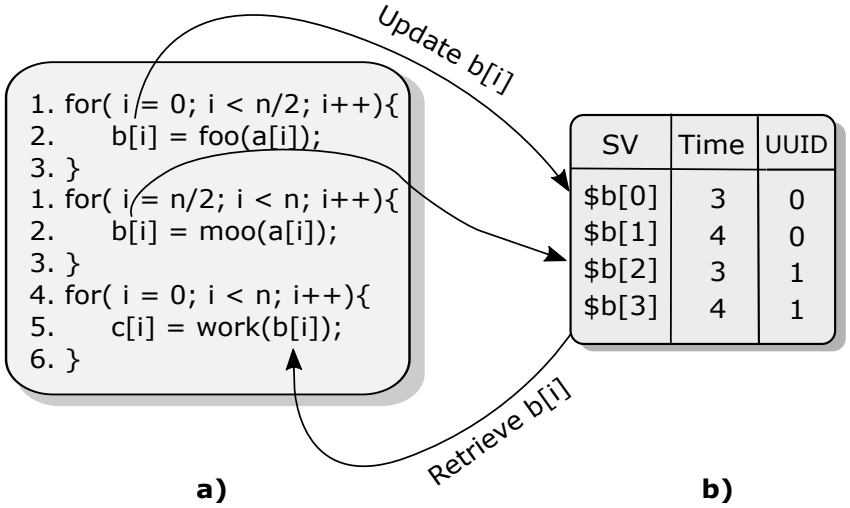


Figure 5.1: A trivial example highlighting the new information stored at runtime in a Cprof simulation. That is the UUID of the function call that wrote the variable.

To construct the channels in Cprof+, information on which node wrote the token needs to be obtained. This is obtained by adding another dimension to the Shadow Variable, that is the Universally Unique Identifier (UUID) of the writing node. For example, Figure 5.1 demonstrates the new function of the Shadow Variable. As variable `b[i]` is being written to by two different functions we can see that in the Shadow Variable of `b[i]` that its is reflected by the different UUIDs. Where UUID 0 and 1 represent functions `foo` and `moo` respectively.

Each variable in a function can have multiple sources when read. As Figure 5.1(b) highlights the Shadow Variable of `b[i]` has two sources. When modelling `b[i]` as a `CprofVariable`, a list of sources must be stored. The list is a collection of `CprofFifo` objects which contain all the information regarding the source. When the variables are read during simulation, the matching `CprofFifo` object is updated. The correct `CprofFifo` object is selected based on a UUID. When the `CprofFifo` object does not exist within the `CprofVariable`, a new `CprofFifo` object is created with the UUID.

With a dynamically created list of channels, Cprof+ has the ability to derive the communication topology during run time. `CprofFIFO` objects are used to store the read and write information of that variable and contains all the logic for channel type identification. Figure 5.2 highlights the data structure of the `CprofFIFO` object and its relation with the `CprofVariable` object. The data structure of the `CprofFIFO` object shows the `UUID` of the writing function as well as the UUID of the reading function (`myUUID`).

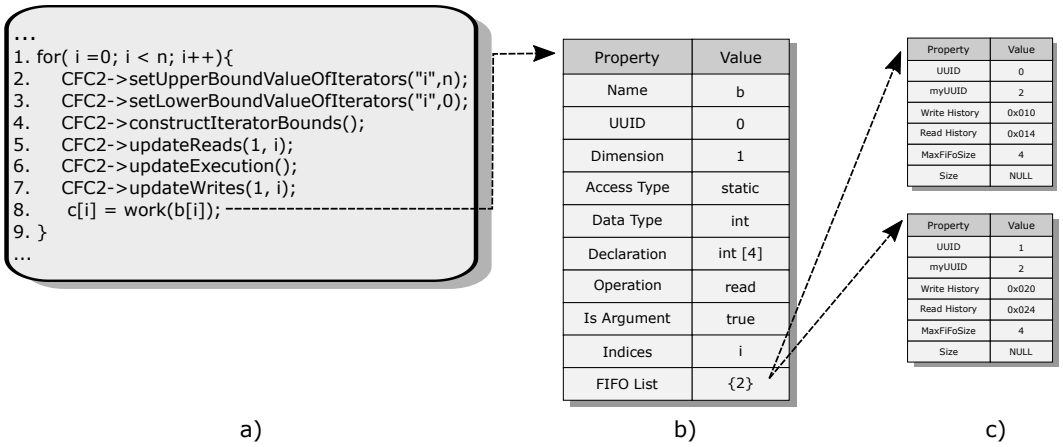


Figure 5.2: Demonstrating the data structures of CprofVariable (b) and those of CprofFifo (c) used in an instrumented version of the code (a)

5.1.2 Calculating Channel Size

Calculating channel sizes are the next critical step in the implementation. Each `CprofVariable` within a `CprofFunctionCall` contains a list of `CprofFifo` objects with its read and write history. Based on this history, the channel size can be calculated using Algorithm 1.

The size of the channels calculated using Algorithm 1. The calculation is done after the simulation is complete. Post-simulation analysis is required as some self loops do not have all the information available at run time to calculate the correct channel size. As a token in the self loop may be used later in an iteration causing the channel to exhibit multiplicity. To calculate the amount of iterations till the token is used again in the self loop requires all iterations to be known.

The calculation of channel sizes can be separated into two separate algorithms. Section 5.1.2.1 will describe the algorithm used for calculating sizes of channels that do not contain self loops. Section 5.1.2.2 will deal with the self loop case.

5.1.2.1 Calculating channel size for non self Loops

Algorithm 1 demonstrates the method for calculating the size of channels which do not contain self loops. The method compares the read and write times of an iteration. Each iteration is saved to memory during run-time with the write time and read time pair of that variable. The read times are used to traverse the write history of the `CprofFifo`.

Line 2 of Algorithm 1 is used to check for multiplicity. Multiplicity is found when the same write time is found twice within the history. Line 11 gives the condition for such an instance. If multiplicity is found, Lines 21-40 help to ensure the calculated channel size does not exceed the maximum calculated size.

Algorithm 1 Cprof FIFO Sizing

Input: Read and Write History of the Variable
Output: Unsigned Integer of FIFO Size

```
1: done = false
2: checkWrite = writeHistory[0];
3: for i = 0 To Read History Size do
4:     size = 0;
5:     read = ReadHistory[i];
6:     for j = 0 To Write History Size do
7:         write = WriteHistory[j]
8:         if write ≤ read then
9:             size++
10:        end if
11:        if !done && write == checkWrite && j != i then
12:            done = true
13:        end if
14:    end for
15:    checkWrite = write;
16:    size = size - i;
17:    if size > maxSize then
18:        maxSize = size
19:    end if
20: end for
21: if done == true then
22:    maxSizeTemp = maxSize
23:    if maxFifoSize > 1 then
24:        if maxSize < History Size then
25:            if maxFifoSize > maxSize then
26:                maxSize = size
27:            else
28:                maxSize = maxSizeTemp
29:            end if
30:        else
31:            maxSize = maxSizeTemp
32:        end if
33:    else
34:        if History Size < maxFifoSize then
35:            maxSize = History Size
36:        else
37:            maxSize = maxFifoSize
38:        end if
39:    end if
40: end if
41: return maxSize
```

Listing 5.1: C code of the function compaan_outlinedproc3 of the *atax* benchmark.

```
1 ...
2 for (i = 0; i < _PB_NY; i++) {
3     for(j = 0; j < _PB_NX; j++){
4         compaan_outlinedproc3(&tmp[i], tmp[i], A[i][j], x[j]);
5     }
6 }
7 ...
```

The maximum channel size is calculated based on the largest iterator value seen and the dimension of the variable. For example, a variable defined as $A[8][8]$ will have a maximum channel size of 64 as each iterator dimension is size 8 and it is 2-dimensional.

If the maximum channel size is not reached, the algorithm increments through each read time ($ReadHistory[i]$) in the history and compares it with all write times ($WriteHistory$). If the write time is less than or equal to read time the size is incremented. To ensure the correct size is calculated, the size is subtracted by the read index (Line 16). We assumed the values that have been read are not stored any more.

The time complexity of the algorithm is $\mathcal{O}(nm)$ where n is the length of the read history and m the write history. The space complexity is $\mathcal{O}(1)$ as the memory requirements do not increase given the read and write history.

5.1.2.2 Calculating Self Loop Channel Size

Algorithm 2 gives a method for calculating the channel size of a self loop channel. The read pattern of the self loop is different to other channels as the data is either read write after it is written or in a certain number of iterations later.

Line 3 of Algorithm 2 demonstrates when the write and read times of an iteration for a variable are equal, it indicates the variable is being read as it is being written. As the variable is being read the iteration after it is written, it will only need a channel size of 1. The function `compaan_outlinedproc3(&tmp[i], tmp[i], A[i][j], x[j])` of *atax* demonstrates the direct read after write self dependency. The variable `tmp[i]` is read just after it is written. Listing 5.1 shows the C code implementation of *atax*

When the write time of a variable and the read time do not match, the history of the self loop is searched. The search of the read history, Lines 6-30, looks for the same read indices, but later in time. The distance between the two indices determines the amount of tokens that need to be stored before the current token is used.

If the current token is not found, line 29 will try to calculate a channel size based on the Algorithm 1. The `historyIndex` is used to reduce the amount of elements in the write history array that need to be searched. Start position for each search iteration of the token is based on the `historyIndex` which is incremented after each iteration as shown in Line 4 and 11 of Algorithm 2.

The time complexity of Algorithm 2 is $\mathcal{O}(nm)$ where n and m are the read history size and write history size respectively. The space complexity of the algorithm is $\mathcal{O}(1)$ as no other storage is required.

Algorithm 2 Cprof Self Loop FIFO Sizing

Input: Read and Write History of the Self Loop
Output: Unsigned Integer of FIFO Size

```
1: for i=0 to Read History Size do
2:     size = maxSize
3:     if writeTime[i] == readTime[i] then
4:         historyIndex++
5:     else
6:         sizeBefore = size
7:         for j = historyIndex to Write History Size do
8:             if WriteIndices[j ] == ReadIndices[i] then
9:                 if index != j then
10:                    found = true
11:                    historyIndex++
12:                    maxsize = max(size, maxSize)
13:                    if size > maxSize then
14:                        size = maxSize return size
15:                    else
16:                        size = 1
17:                    end if
18:                end if
19:                break
20:            else
21:                size++
22:                if j > Write History Size then
23:                    size = sizeBefore
24:                end if
25:            end if
26:        end for
27:        if !found ∧ historyIndex < ReadWrite History then
28:            calculateChannelSizeUsingTime(size, historyIndex)
29:        end if
30:    end if
31: end for
32: return maxSize
```

5.1.2.3 Channel Size Calculation Example

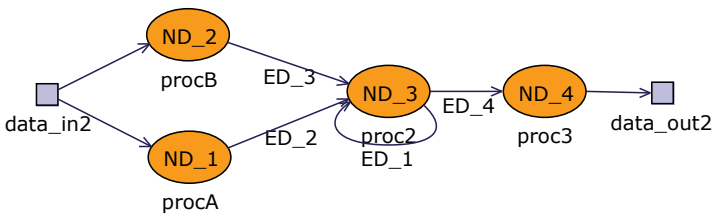


Figure 5.3: PPN of example channel size calculation

Cprof+ calculates the channel sizes using Algorithms 1 and 2. The PPN in Figure 5.3 will illustrate the channel size calculation. Listing 5.2 demonstrates the C code description of the PPN. The functional latency, Λ_F, of `proc2` is 3. While `procA`, `procB` and `proc3` are given a functional latency of 1.

Listing 5.2: Source Code of an example IP Core

```
1  // Stream data into the design
2  for (i = 0; i < 3; i++) {
3      for(j=0; j< 3; j++){
4          procA(&a[i][j], data_in2[i][j]);
5      }
6  }
7  // Stream data into the design
8  for (i = 0; i < 3; i++) {
9      for(j=0; j< 3; j++){
10         procB(&b[i][j], data_in2[i][j]);
11     }
12 }
13 for(t=0; t < 3; t++){
14     for (i = 0; i < 3; i++) {
15         for(j = 0; j < 3; j++){
16             proc2(&a[i][j], a[i][j], b[j][i]);
17         }
18     }
19 }
20 for (i = 0; i < WIDTH; i++) {
21     for(j=0; j< WIDTH; j++){
22         proc3(&data_out2[i][j], a[i][j]);
23     }
24 }
```

Table 5.1: Communication Channel, ED_2, showing the read and write history of Variable A.

Iteration	Write Time	Read Time	Channel Size
(0,0)	3	3	1
(0,1)	4	4	1
(0,2)	5	5	1
(1,0)	6	6	1
(1,1)	7	7	1
(1,2)	8	8	1
(2,0)	9	9	1
(2,1)	10	10	1
(2,2)	11	11	1

Table 5.1 shows the channel history of ED_2 for variable A in Figure 5.3. The first 9 iterations of `proc2` use the variable written by the streaming function `procA`. The table shows that the variable is read as soon as it is written. Algorithm 1 yields a channel size of 1 for all cases. As the variable is read and then never used again, a case of IOM-communication.

Table 5.2: Communication Channel, ED_3, showing the read and write history of Variable B with channel size updated as the algorithm parses the history. Only iterations for $t = 0$ and $t = 1$ are shown.

Iteration	Write Time	Read Time	Channel Size
(0,0)	3	3	3
(1,0)	6	6	12
(2,0)	9	9	21
(0,1)	4	10	24
(1,1)	7	11	27
(2,1)	10	12	27
(0,2)	5	13	27
(1,2)	8	14	27
(2,2)	11	15	27
(0,0)	3	16	27
(1,0)	6	17	27
(2,0)	9	18	27
(0,1)	4	19	27
(1,1)	7	20	27
(2,1)	10	21	27
(0,0)	5	22	27
(1,0)	8	23	27
(2,0)	11	24	27
...	...	...	...

The ED_3 highlights a OOM channel between node 2 and node 3. The channel sends the variable B and as Table 5.2 shows the calculation of the channel size is not trivial. Function `proc2` is contained within a 3 nested loop. Each loop dimension has size of 3 and as Table 5.2 demonstrates the calculated channel size is 27. The true channel size is 9 and this is corrected in Algorithm 1 with lines 21-40. If a read occurs more than once in the channel history, e.g. iteration (0,0) with 3 at read times 3 and 16, it indicates the channel has multiplicity. The algorithm will correct by setting the maximum channel size. Variable B has a maximum size of 9 as it is a 2-dimensional 3x3 array.

Table 5.3 highlights the calculation of the channel size for the self loop of `proc2`, i.e. ED_1. Algorithm 2 is used. The first iteration $(0, 0)$ calculates a size of 9 as the write and read times do not match. The read iteration history is search till the next token $(0, 0)$ is found. The distance between the two tokens in the iteration read history is 9 and for the first iteration a value of 9 is calculated. The same method is used for the next rows of Table 5.3. When iteration $(0, 0)$ with read time of 26 is reached, the algorithm stops incrementing the channel size as the write and read times are equal.

Table 5.3: Communication Channel, ED_1, showing the read and write history of Self Loop Variable A with channel size updated as the algorithm parses the history.

Iteration	Write Time	Read Time	Channel Size
(0,0)	13	16	9
(0,1)	14	17	9
(0,2)	15	18	9
(1,0)	16	19	9
(1,1)	17	20	9
(1,2)	18	21	9
(2,0)	19	22	9
(2,1)	20	23	9
(2,2)	21	24	9
(0,0)	26	26	9
(0,1)	27	27	9
(0,2)	28	28	9
(1,0)	29	29	9
(1,1)	30	30	9
(1,2)	31	31	9
(2,0)	32	32	9
(2,1)	33	33	9
(2,2)	34	34	9

5.2 Validation

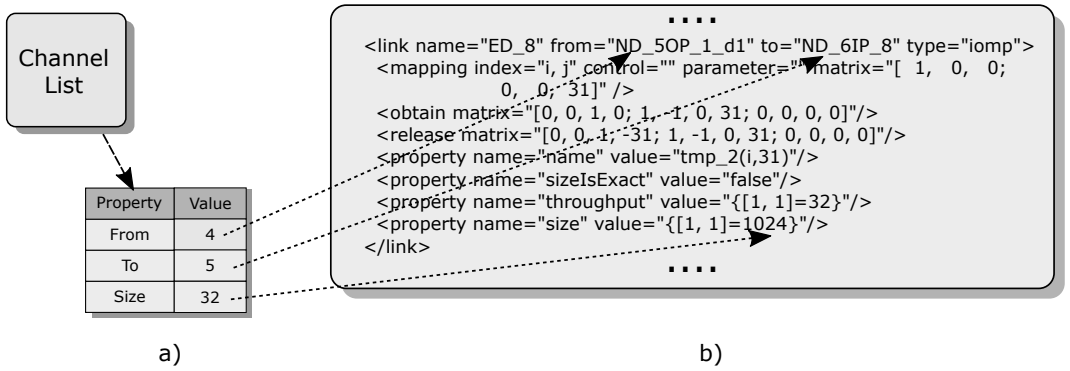


Figure 5.4: Matching the FIFO description in Cprof+ (a) to that of the KPN in Compaan (b) for ATAX.

To ensure the channel sizes computed by Cprof+ are correct, the sizes were checked against the RTL implementation. By mapping the communication channels in Cprof+, to those in the PPN model used in the Compaan Tool Chain, we can run the RTL simulations with Cprof+ values. If the channel sizing is done incorrectly, the RTL simulations will block.

A parsing program was created to check the PPN model of the program against

the Cprof+ channel sizes calculated. Figure 5.4 demonstrates visually how the Cprof+ value is compared and matched with the corresponding PPN edge in file. Notice, that the naming of the nodes in Cprof+ is based on 0 and those in the PPN on 1. Cprof+ node values are corrected for this offset.

Algorithm 3 Cprof+ Mapping of Channel Sizes to Compaan PPN File

Input: CprofFunctionCalls C and PPN Graph P
Output: Modified PPN Graph

```
 1: channels ← ∅
 2: for all c ∈ C do
 3:     for all v ∈ c do
 4:         for all f ∈ v do
 5:             if f ∉ channels then
 6:                 f ∪ channels
 7:             else
 8:                 updateSize(f, channels)
 9:             end if
10:         end for
11:     end for
12: end for
13: for all lines ∈ P do
14:     if checkLine(line, "<link name") then
15:         for all f ∈ channels do
16:             if compareNodes(f, line) then
17:                 incrementLineBySeven(P)
18:                 replaceStringBetweenWith("]=", "}", size(f))
19:             end if
20:         end for
21:     end if
22: end forreturn P
```

Algorithm 3 describes the method of modifying a PPN file with the values calculated by Cprof+. Figure 5.4(b) shows the file structure of the PPN description for Compaan. The mapping algorithm works by searching the PPN file for a edge using the marker, "<link name". Once a edge has been found, it will search through a compact list of channel sizes created from Cprof+ to match node names. The edges from Cprof+ and the PPN are matched by checking the strings "from=..." and "to=" on the line containing "¡link name=". The matching is done on Line 16 of the mapping algorithm. The PPN is updated once a match is found by replacing the string contained between "]=" and "}" in the line "<property name="size"". The mapping process is then repeated for all channels in the list.

The mapping algorithm has one caveat. As the communication channels for the PPN were calculated symbolically by Compaan, there exists some instances were there are multiple channels between nodes. In this case, Cprof+ cannot model all channels as it creates channels based on the reading and writing of variables. To overcome this problem, the size calculated by Cprof+ for an edge between two nodes is applied to all

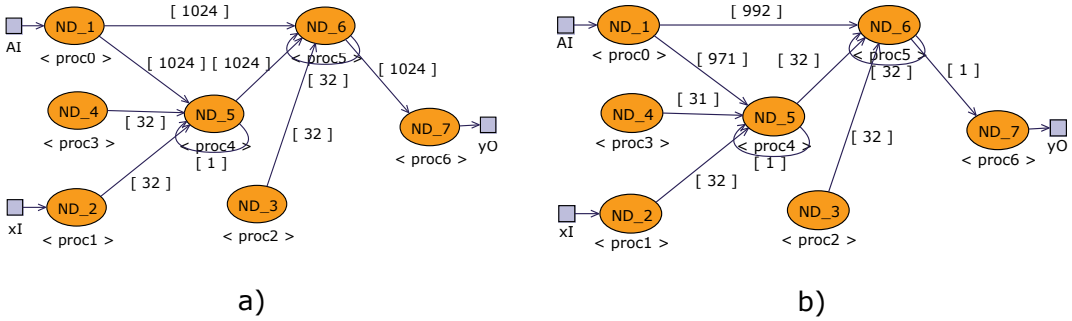


Figure 5.5: Comparison of ATAX KPNs with (a) original fifo sizes and (b) with Cprof calculated sizes.

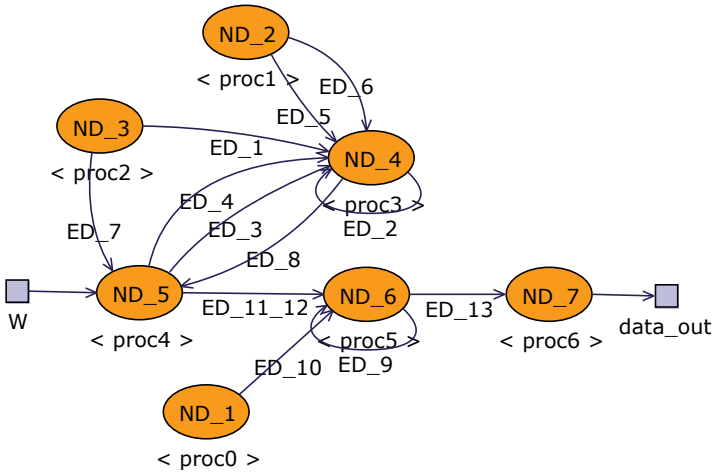


Figure 5.6: PPN of the *dynprog* benchmark

channels between the same two nodes in the PPN. In some cases, this leads to a less optimal solution, as some of the channels do have a smaller size.

The Compaan/LAURA tool chain is executed using the modified PPN with Cprof+ channel sizes. The result of the simulation is saved to a text file which is compared to the original RTL simulation results.

5.3 Results

Cprof+ yields channel sizes much less than those calculated in the Compaan Tool Chain. In most cases, a 50% to 96% reduction in the memory footprint was achieved with Cprof+. While maintaining a deadlock free network. In some cases, the performance of the resulting network is degraded due to the channel sizes from Cprof+. For example, in *fdtd_2d, dynprog, mm2, mm3* and *mvt* experience performance degradation due to the new channel sizes. A closer look at *dynprog*, shown in Figure 5.6 reveals that the Cprof+ channel size of ED_8 caused the network to bottleneck. Cprof+ calculates a channel size of 32. A channel size of 32 for ED_8 causes blocking writes, but a size of 33 does not. The reason for the mismatch in size is unclear as the data available to Cprof+

indicates a maximum value that can be calculated for the size is 32 for ED_8. Though, the channel size leads to a deadlock free network, the size inhibits the network to run at full speed. The results also showed that benchmark fdtd-apml causes a error in a pipeline of a function with the modified channel sizes. This yielded no result for the benchmark as shown in Table 5.4. The reason for the error is unclear as the synthesis and design of the process pipeline is carried out by Compaan/LAURA tool flow.

Case	Original	Modified	Diff(%)	Old Mem	New Mem	FifoSaving(%)
adi	5606	5699	1.6	7056	2185	69.0
atax	19963	19962	0.0	4225	2124	49.7
bicg	19911	19910	0.0	4257	1104	74.1
cholesky	91604	91597	0.0	12130	1252	89.7
correlation	338579	353917	4.3	43270	9127	78.9
covariance	342014	359305	4.8	40034	5712	85.7
doitgen	181047	181043	0.0	12011	1997	83.4
durbin	10909	10876	-0.3	1994	670	66.4
dynprog	253593	286447	11.5	94893	3540	96.3
fdtd_2d	2140	5673	62.3	34891	7467	78.6
fdtd_apml	4609	N/A	N/A	23623	3401	85.6
floyd_warshall	2086	2141	2.6	842	566	32.8
gemm	99251	99244	0.0	35874	3076	91.4
gemver	55050	56970	3.4	5412	2371	56.2
gramschmidt	4721	4713	-0.2	1902	493	74.1
jacobi_1d_imper	1045	1043	-0.2	8469	1428	83.1
jacobi_2d_imper	1912	1912	0.0	21384	5106	76.1
lu	31571	31568	0.0	49202	5986	87.8
ludcmp	122773	122739	0.0	27249	2769	89.8
mm2	521222	553889	5.9	37956	8163	78.5
mm3	631730	694188	9.0	70755	12284	82.6
mvt	10724	11684	8.2	4226	2069	51.0
reg_detect	4571	4568	-0.1	2971	157	94.7
seidel_2d	181242	182981	1.0	24731	25441	-2.9
symm	281367	281364	0.0	57368	6664	88.4
syrk	508979	508971	0.0	36867	3074	91.7
syr2k	1080382	1080382	0.0	12288	5125	58.3
trisolv	9518	9512	-0.1	2737	597	78.2
trmm	420323	435200	3.4	24591	3008	87.8

Table 5.4: Comparison of the original performance estimations of Cprof+ against Vivado RTL Simulation results.

5.4 Limitations

The main limitation of the algorithm used to calculate channel sizes in Cprof+, is the inability for Cprof+ to model every edge explicitly. Compared to the mathematically PPN description of the network. Cprof+ generates channels based on the write and read iteration pairs. This method of channel generation, leads to an oversimplification for some cases.

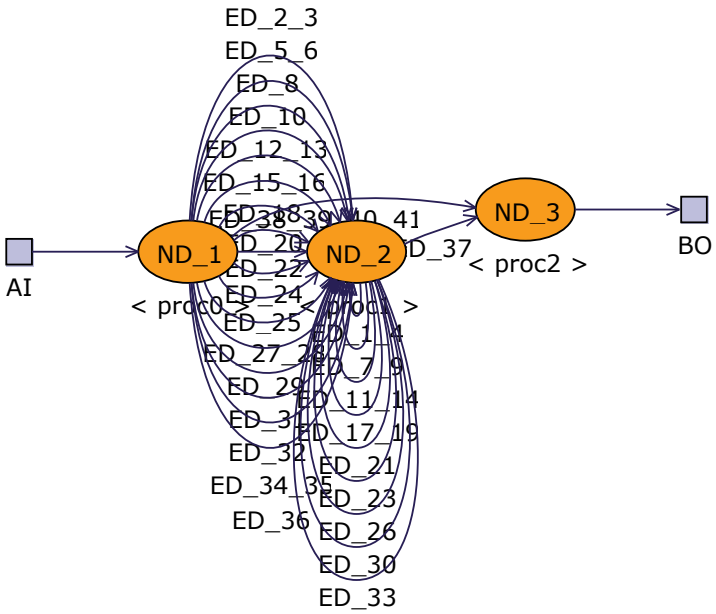


Figure 5.7: PPN of the Siedel 2D Benchmark showcasing the diversity in communication channels

The PPN in Figure 5.7 highlights the high amount of edges between nodes in *siedel-2d*. Cprof+ simplifies the edges between `ND_1` and `ND_2` to 1. A single edge is modelled due to the single output/input relation between `proc0` and `proc1`. When Cprof+ applies the channel sizes to the PPN, the maximum channel size is used for all edges. Even when some channels may have a smaller size. As Table 5.4 shows seidel_2d yields a memory footprint that is larger than the original which leads to a worse solution when compared to the Greedy Algorithm used in KPNRateMatcher of Compaan.

Code Transformation and Optimization

6

Cprof+ can determine the performance for a c-program as a PPN accurately. By transforming the C-program, we can explore different design points. How to apply source-to-source transformations to the C-code is crucial for Cprof+ to understand. Unnecessary code transformations can cause inefficiency in terms of hardware resources used. The goal is to apply common loop transformations in an intelligent and efficient manner.

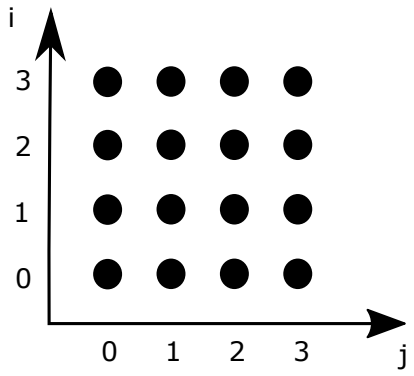
Cprof implemented two types of transformations for finding other design points of an application. Cprof+ will investigate the transformations and propose a more intelligent method of finding optimal design points for the designer.

6.1 Types of Transformations

There are many types of code transformations that can be applied to programs to induce performance increase[38][39]. The transformations may lead to increased hardware use. The transformations discussed in the following sections include: Modulo Unfolding and Plane Cutting.

6.1.1 Modulo Unfolding and Plane Cutting

The first transformation technique that was implemented in the original Cprof was Modulo Unfolding and Plane Cutting. Two code transformation techniques increase performance by adding hardware resources that can run in parallel by partitioning the iteration domain over multiple hardware resources.



b) Iteration Domain

```
1. for( i = 0; i < 4; i++){
2.     for( j = 0; j < 4; j++){
3.         b[i, j] = foo(a[i,j]);
4.     }
5. }
```

b) Original Code

Figure 6.1: Example of a simple loop with no dependencies between iterations.

Figure 6.1 highlights a simple example of a for loop 6.1(b) and its iteration domain 6.1(a). The iteration domain shows, all iterations within the loop are independent to one another.

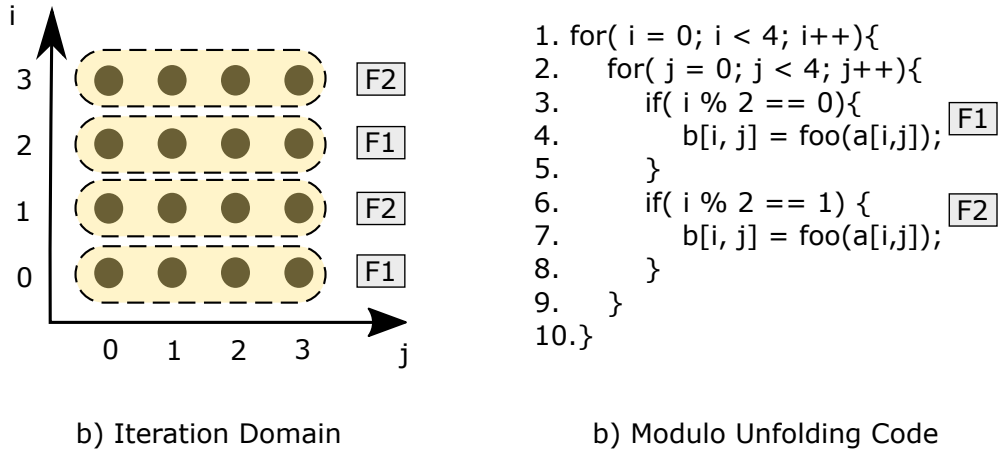


Figure 6.2: Example of Modulo Unfolding with factor 2.

The first transformation that can be applied to the code in Figure 6.1 is Modulo Unfolding. Figure 6.2(b) highlights the modified code. The code demonstrates an unrolling factor of 2 by replacing the function with 2 conditional statements. The condition of the statements are based on the iterator i. The conditional statements help to schedule the iteration domain to the correct compute node. The modulo function (%) is used to execute iterations that have remainder 0 to a single compute node `(F1)` and remainder 1 to another `(F2)`. Thereby increasing the throughput of the system as both compute nodes can execute in parallel.

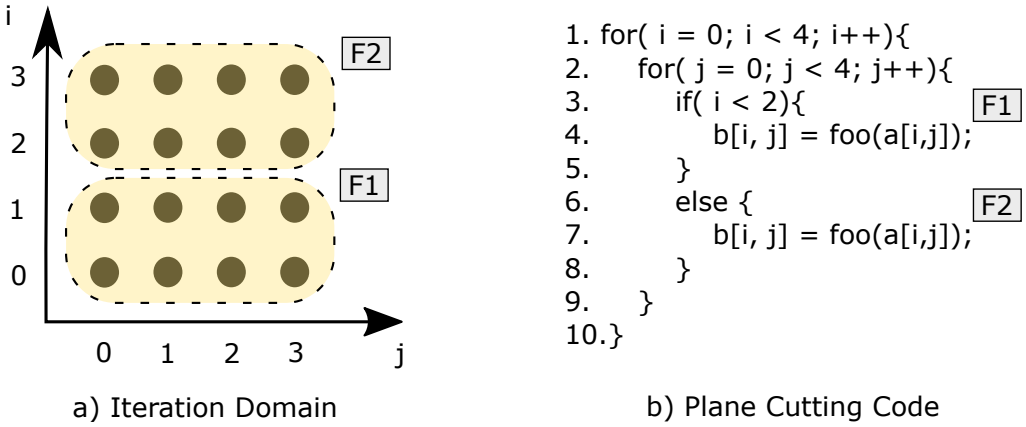


Figure 6.3: Example of Plane Cutting with factor 2.

The same operation can be accomplished with plane cutting in Figure 6.3. Rather than implement a modulo conditional statement, an equality is added, e.g. $i < 2$. Figure 6.3(b) line 3 highlights the equality. The consequence of the equality is the iteration domain is partitioned into two regions as shown in 6.3(a). One compute node

(F1) executes the bottom half of the iteration domain and compute node (F2) the top part. Compute nodes F1 and F2 can execute in parallel.

In this particular case, both loop transformations accomplish the same goal. That is, they divide the iteration domain to increase parallelism. This division leads to a better performance. One of the advantages of using the transformations is the lack of data flow dependency checks. As the transformations do not affect the execution order of the program. If for example, there were inter iteration dependencies that occurred across a transformation border, the compute node will simply stall until the data is ready.

6.1.1.1 Case Study: ATAX

To understand how the transformations affect the performance of a PPN, we conducted a case study on *atax*. *atax* is a matrix transpose and vector multiplication benchmark. *atax* was analysed using Cprof and the values were validated with the values generated by RTL simulations. It was found that for the *atax* benchmark, values calculated by Cprof differed at most by 2.65% with those computed by RTL simulations.Table A.1 shows the estimated Cprof+ estimated performance values with the RTL simulation values.

PC Factor	Cprof	Vivado RTL
2	10524	10539
4	5812	5827
8	3456	3471
16	2278	2239

Table 6.1: Plane Cutting results for compaan_outlineproc3 around iterator j up to a factor of 16 with Cprof and Compaan comparison.

The case study was conducted using -MINI_DATASET where the values of NX and NY are set to 32. Figure A.1 in Appedix A demonstrates, the whole DSE can be traversed by Cprof simply using the Modulo Unfolding code transformation. Teijlingen[4] demonstrated the bounds of the design space with the absolute and unbounded throughput calculations. The red lines indicate the absolute and unbounded throughput of the network. By unfolding a function such as compaan_outlinedproc3, as shown in Figure 6.6, by a factor of 32. The transformation can induce a performance gain close to the unbounded throughput. The unbounded throughput case yields a performance estimation of 1286 cycles. What is more interesting, is that the critical section of the network is compaan_outlinedproc3 as unfolding or plane cutting other nodes while leaving compaan_outlinedproc3 will not cause performance gain.

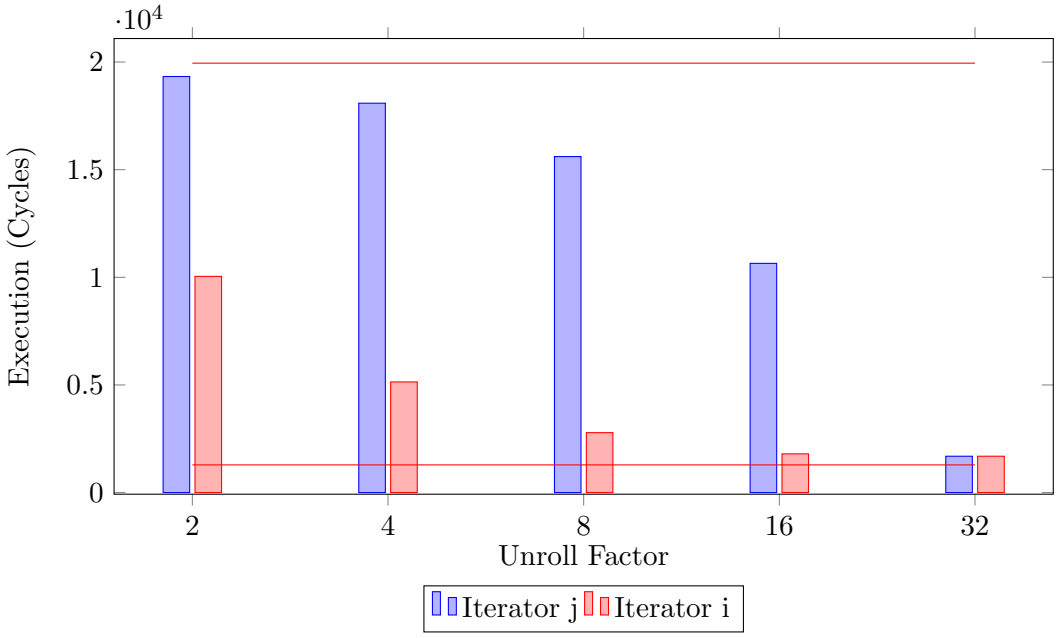


Figure 6.4: Modulo unfolding results for compaan_outlinedproc3 around iterators i and j up to a factor of 32.

The unrolling of non-critical nodes in the network and their lack of effect is supported by Table 6.2. The Table demonstrates, no performance gain is achieved by unrolling `compaan_outlinedProc4`. The gain is only achieved through unrolling `compaan_outlinedProc3`. The reason can be explained through the PPN graph of *atax* in the Figure 6.5

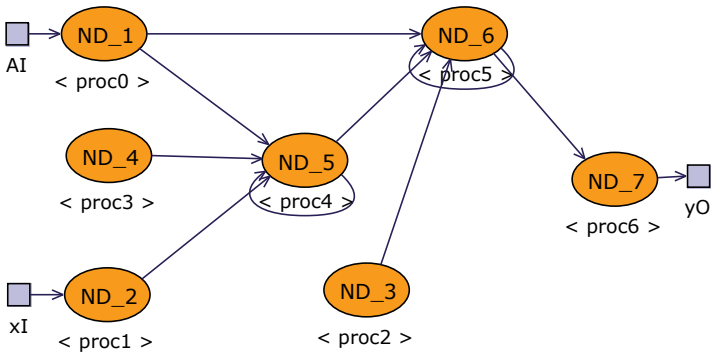


Figure 6.5: The PPN graph of atax

```
1     for (i = 0; i < _PB_NY; i++) {
2         for(j = 0; j < _PB_NX; j++){
3             compaan_outlinedproc3(&tmp[i], tmp[i], A[i][j], x[j]);
4         }
5     }
6     for (i = 0; i < _PB_NX; i++) {
7         for (j = 0; j < _PB_NY; j++) {
8             compaan_outlinedproc4(&y[j], y[j], A[i][j], tmp[i]);
9         }
10    }
```

Figure 6.6: C code of the ATAX benchmark.

compaan_outlinedProc4	compaan_outlinedProc3	Cycles
0	8	15608
2	8	15608
4	8	15608
8	8	15608

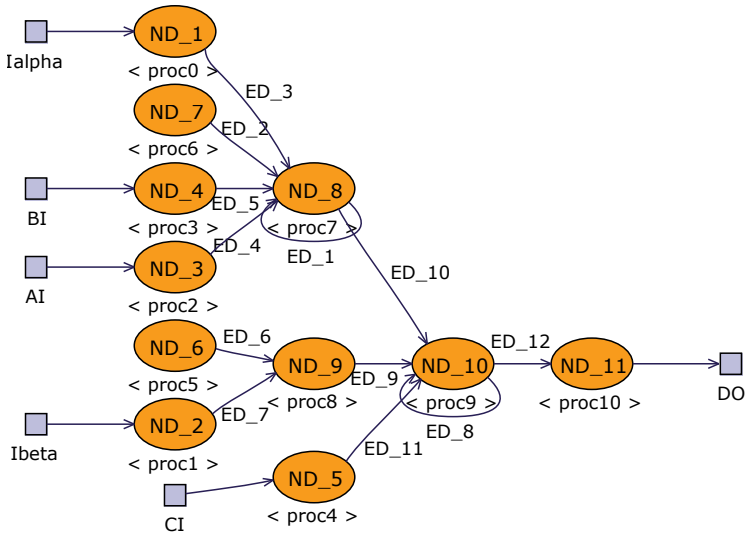
Table 6.2: Modulo unfolding both compaan_outlinedProc3 and compaan_outlinedProc4 around iterator j.

The dependency between the two functions can be clearly seen in Figure 6.5. `compaan_outlinedProc4` depends on the output of `compaan_outlinedProc3`. It must wait till the whole row calculation is complete before it can begin its execution. Figure 6.6 shows that the variable `tmp[i]` must wait till `compaan_outlinedproc3` has completed an entire loop of j before it can start execution. Cprof+ will have to incorporate a method for identifying the correct iterator for optimization. Section 6.2.3 will explain the method used in Cprof+.

6.1.1.2 Case Study: MM2

The next benchmark we studied using the modulo unfolding and plane cutting transforms was *mm2*. The *mm2* benchmark is a 2-dimensional matrix-matrix multiplication program. mm2 is an important operation that is often times used in all kinds of applications. Figure 6.7 shows the PPN graph of *mm2*.

The previous case study of *atax* showed that only two nodes affect the performance. In the case of *mm2*, those are nodes `ND_8` and `ND_10`. The effect of `ND_9` is negligible even though the latency of the procedure is comparable to `ND_8` and `ND_10`. The size of the iteration domain for `ND_9` is a magnitude smaller than the critical nodes. The reduced domain size indicates the node executes less than the critical nodes. Potentially reducing the effect on performance.



a) MM2 KPN

Nodes	Latency
1-7 & 11	1
8	14
9	7
10	8

b)Function Latencies

Figure 6.7: PPN Diagram of the MM2 benchmark (a) and the corresponding Function Latencies (b)

Both `ND_8` and `ND_10` are required for maximum performance gain. Completely unfolding only one of the nodes around any of the 3 iterators: i, j or k leads to a maximum performance gain of 25% and 1% for `ND_8` and `ND_10` respectively. This is demonstrated in Figures A.3 and A.2 of Appendix A.

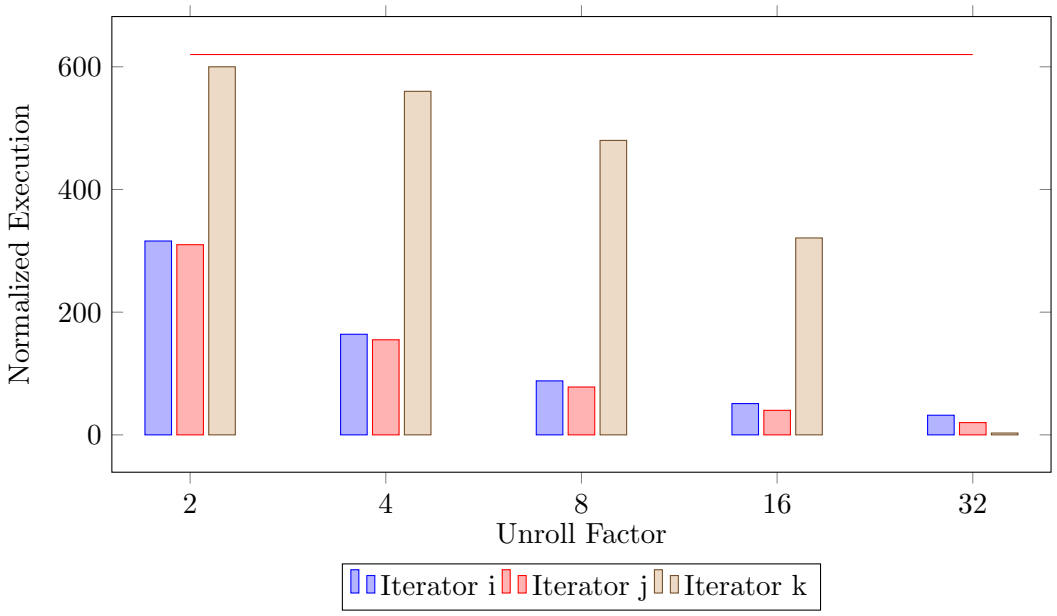


Figure 6.8: Modulo unfolding results ND_8 and ND_10 around iterators i, j and k up to a factor of 32 normalized to the Lowerbound of 838 cycles.

The choice of iterator when unfolding or cutting a loop is an important choice.

Figure 6.8 demonstrates the effect of the iterator and unfold factor has on the network. Unfolding around k, for example, results in little performance gains when compared to i or j. The gain of unfolding around k exceeds i and j once the iterator k is fully expanded, approaching the unbounded throughput case. Fully unfolding a node around an iterator causes a large increase in hardware resources required. The cost effective method would be to unfold with a factor of 2 around loop iterator i or j. As this transformation would only introduce 2 more processes to the network. For an approximate 50% performance gain for the network as shown in Figure 6.8.

6.1.2 Effects of Function Latency on Performance Gain with Transformations

The latency of a function can be important in some cases for the amount of performance. The effects on varying function latency are investigated in this Section to understand the amount of performance gain that can be obtained using the modulo unfolding transformation. Understanding how functional latency (Λ_F) contributes to the performance of a network can be useful for Cprof+ when identifying functions to optimize.

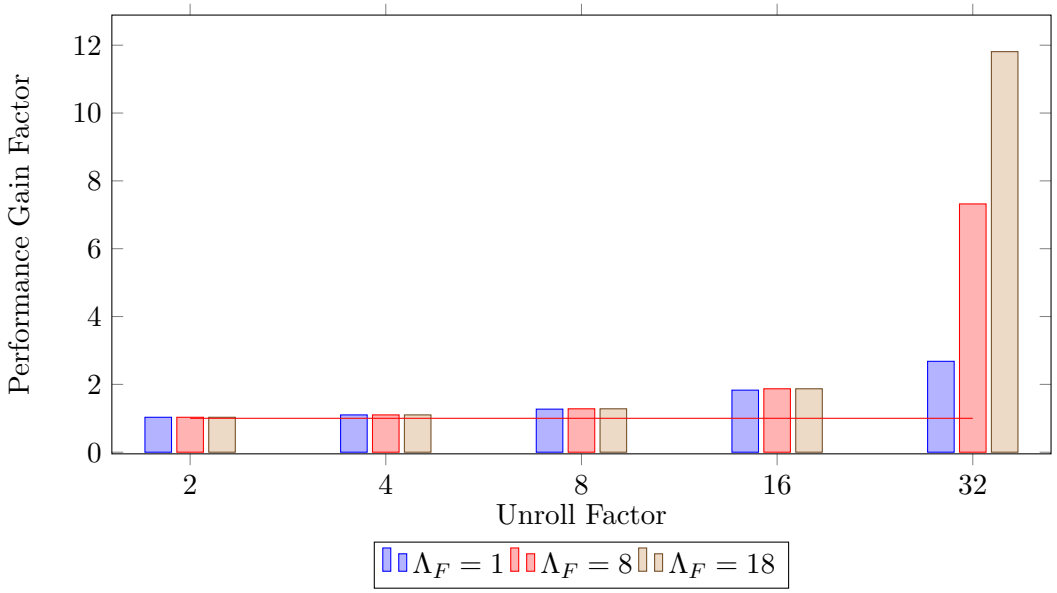


Figure 6.9: Modulo unfolding results for compaan_outlinedproc3 around iterators j with varying Function Latency (Λ_F) for ATAX.

As identified earlier, the critical process for the *atax* PPN is the `compaan_outlinedproc3`. The function latency was modified to understand how the effect of the transformation changed. Figure 6.9 shows the amount of performance gain achieved through unrolling the critical node does not change drastically until the iterator has been fully unrolled. The effects of different function latencies become apparent only when fully unrolled. These effects suggests that the latency of the function or pipeline depth does not affect the amount of performance that can be

obtained. The difference between function latencies becomes apparent when the function is full unfolded as the data dependencies between iterations do not exist any more The results for Figure 6.9 was obtained by unfolding around Iterator j, the most ineffective iterator as Figure A.1 indicates.

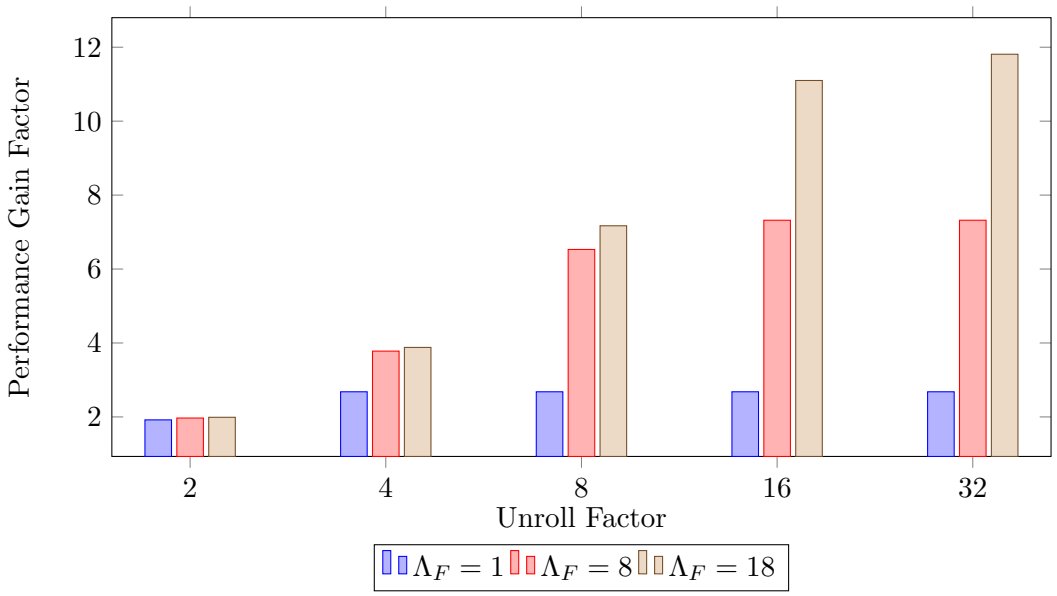


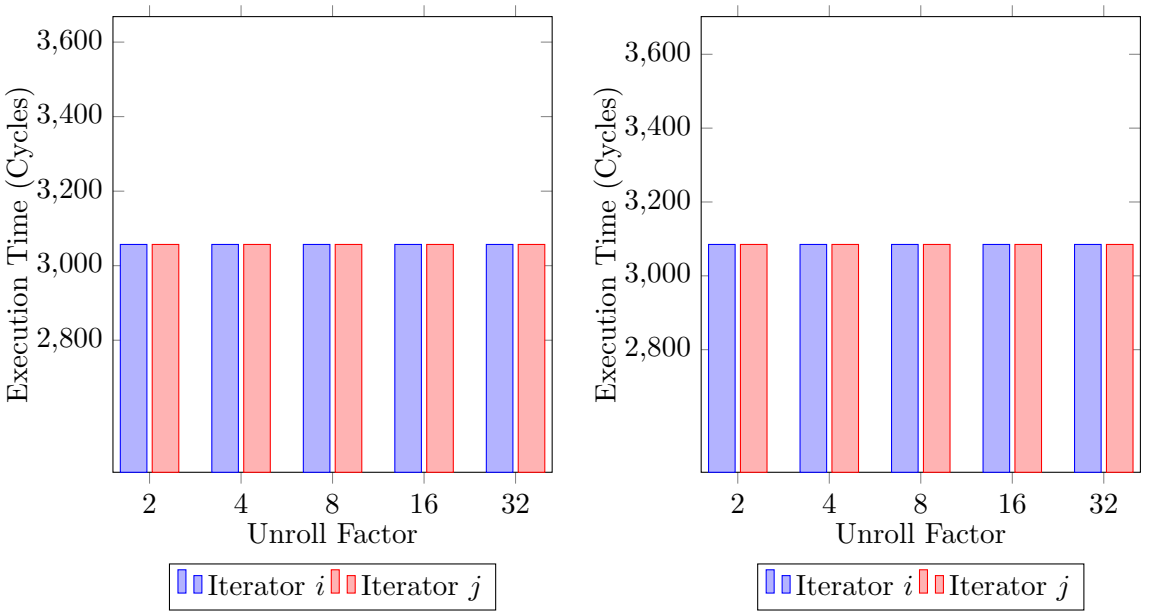
Figure 6.10: Modulo unfolding results for compaan_outlinedproc3 around iterators i with varying Function Latency (Λ_F) for *atax*.

Figure 6.10 shows when the function latency for `compaan_outlinedproc3` is changed and the transformation is applied around iterator i. The amount of performance gain varies with function latency and the factor of unfolding. With a $\Lambda_F = 1$, the critical node does not obtain a performance increase after an unroll factor of 4.

As the function latency increases, the amount of performance that can be extracted from the function also increases. This can be important for selecting critical nodes within a PPN for optimization. Though the node may be a bottle neck in the network, a small function latency may indicate that the amount of performance gain available is small. Even with a small function latency the critical node in ATAX can provide a 1.98x factor performance gain even with an unroll factor of 2. Here the performance gain factor is calculated using the following equation.

$$PerformanceGainFactor = (Absolute\ Throughput\ Case)/(Unfolded\ Case) \quad (6.1)$$

Given that the critical node can obtain gains only with a small unroll factor, can the effect be reduced by changing the function latency of another node? The node executing `compaan_outlinedproc4` was modified, in order to investigate whether a large pipeline depth compared to `compaan_outlinedproc3` will change the dynamics of the network.



(a) Proc3 with $\Lambda_F = 1$ and Proc4 with $\Lambda_F = 8$ (b) Proc3 with $\Lambda_F = 1$ and Proc4 with $\Lambda_F = 36$

Figure 6.11: Modulo unrolling Proc4 with varying pipeline depths while having $\Lambda_F = 1$ for Proc3 within *atax*

Figure 6.11 highlights the result the effect functional latency of `compaan_outlinedproc4` on the network. With a functional latency many times greater than Proc3, no performance gain is obtained by unrolling Proc4. Demonstrating that the data dependency between Proc3 and Proc4 is crucial for the network and the amount of performance that can be extracted.

The results demonstrate that the function latency of a node does not indicate necessarily that the node will affect the performance of the network. Selecting a node that is non-critical and has a large functional latency does not mean a transformation will cause performance gain. The critical node still is required to obtain performance gains.

6.2 Optimization Techniques

The code transformations that are currently implemented in Cprof allow for quickly assessing design points as shown in Sections 6.1.1.1 and 6.1.1.2. The next challenge is to identify functions within a program that will allow for maximum performance gain given the cost of the transformation. The cost is measured in terms of hardware resources used. The amount of hardware resources used is measured by counting the number of processes in the network. The measurement helps to give a first order approximation of the hardware resources required and the change in resources after a transformation.

Before hardware and performance measurements can be made. Performance metrics are needed to help classify critical functions in a program. The following Section 6.2.1 will examine the metrics that can be extracted from the execution profile Cprof+ generates for each function call in the program. With these metrics Sections 6.2.2 and 6.2.3 will look at two approaches used in optimization and compare the results.

Listing 6.1: Excerpt from the XML Cprof Simulation Results of ATAX

```
<name>compaan_outlinedproc4</name>
<metrics>
    <selfLoop>tmp</selfLoop>
    <rbse>993</rbse>
    <rde>31</rde>
    <used>17905</used>
    <avail>2976</avail>
    <pipeUtil>85.7478104</pipeUtil>
    <iAvgPipeUtil>8.41471386</iAvgPipeUtil>
    <avgStagesExec>1.51464856</avgStagesExec>
    <ExecLength>19888</ExecLength>
    <StreamFunction>false</StreamFunction>
    <UnrollIterator>i</UnrollIterator>
    <latencyFunc>18</latencyFunc>
    <UUID>4</UUID>
</metrics>
```

6.2.1 Function Call Metrics

Cprof+ has the ability to capture a variety of metrics of a function within a program. The metrics are derived from the execution profiles of the functions, that were implemented in Cprof. The amount of operations in a pipeline and the pipeline utilization can be determined with the profiles.

Metric	Description
selfloop	Self Loop Variable Name
rbse	Number of reads before a start of execution. (Pipeline is empty)
rde	Number of reads during an execution
used	Number of Cycles Executing
avail	Number of Cycles Idle
pipeUtil	Percentage Pipeline Utilization
iAvgPipeUtil	Average Pipeline Fill
avgStagesExec	Average Number of Executions in the Pipeline
ExecLength	Execution Length of the Node
StreamFunction	Input/Ouput Streaming Function
UnrollIterator	Iterator Selected for Optimization
latencyFunc	Function Latency
UUID	Function UUID

Table 6.3: Description of XML Metric Values

Figure 6.1 and Table 6.3 demonstrate the new metrics Cprof+ has incorporates to measure the function. With metrics `rbse` and `rde`, Cprof+ can determine whether function nodes wait on data to arrive or data arrives during one or more executions. However, these are not the only metrics gathered by Cprof+. With the ability to

Listing 6.2: C Code of the function described in *atax* as seen in Figure 6.1

```
for (j = 0; j < _PB_NY; j++) {
    compaan_outlinedproc4( &tmp[i],tmp[i],A[i][j],x[j] );
}
```

Listing 6.3: Input Variable Metrics from the XML Cprof Simulation Results of *atax*

```
<inputs>
<input0>
    <name>tmp</name>
    ...
    <avgCondSync>9620</avgCondSync>
    <maxCondSync>19840</maxCondSync>
</input0>
<input1>
    <name>A</name>
    ...
    <avgCondSync>496</avgCondSync>
    <maxCondSync>992</maxCondSync>
</input1>
<input2>
    <name>x</name>
    ...
    <avgCondSync>9292</avgCondSync>
    <maxCondSync>19200</maxCondSync>
</input2>
</inputs>
```

model communication channels as described in Chapter 5, Cprof+ can measure the conditional synchronization time of a variable to help identify the variables that require optimization.

Conditional Synchronization time is the measure between the write and read of a token. Take the function given in Figure 6.2. If the write time of $t_{tmp[1]} = 3$, $t_{A[1][0]} = 4$ and $t_{x[1]} = 5$, then the conditional synchronization for all 3 variables will be $0, 1$ and 2 respectively. The *conditional synchronization* time can be an indicator for functions to focus on. As the variable with the maximum conditional synchronization could be optimized to reduce the time the reading function waits. The maximum *conditional synchronization* time may not always indicate a variable needs to be optimized. Take the case of *atax* and `compaan_outlinedproc4`.

Figure 6.3 demonstrates the variables all have large *conditional synchronization* times. If the optimization strategy was to select the variable with the highest *conditional synchronization* time and optimize its writing function. Then the variable *tmp* would be selected. In this case it is the correct variable to select. The performance gain will not come from the transforming the writing function, but from the self loop.

The final set of metrics which is saved for a function node is the communication channel sizes for each variable as calculated in Chapter 5. The channel sizes of the variables can be useful in identifying the amount of data available. A function can be

Listing 6.4: Input Variable FIFO Metrics from the XML Cprof Simulation Results of *atax*

```
<inputs>
    <input0>
    <name>tmp</name>
    <numFifos>2</numFifos>
    <Fifos>
        <Fifo0>
            <UUID>3</UUID>
            <Size>31</Size>
        </Fifo0>
        <Fifo1>
            <UUID>4</UUID>
            <Size>1</Size>
        </Fifo1>
    </Fifos>
    ...
    </input0>
    <input1>
    <name>A</name>
    <numFifos>1</numFifos>
    <Fifos>
        <Fifo0>
            <UUID>0</UUID>
            <Size>971</Size>
        </Fifo0>
    </Fifos>
    ...
    </input1>
    <input2>
    <name>x</name>
    <numFifos>1</numFifos>
    <Fifos>
        <Fifo0>
            <UUID>1</UUID>
            <Size>32</Size>
        </Fifo0>
    </Fifos>
    ...
    </input2>
</inputs>
```

unrolled given the size of the input channels. Figure 6.4 demonstrates the results of the channel sizing for *atax* described in Figure 6.2. The sizes demonstrate variable $A[i][j]$ has the largest size of 971 due to the fact that the tokens are written long before the function enters 53rd of the 1024 iterations.

With all of the metrics, Cprof+ can begin to identify relevant and critical function nodes for optimization. In this work, two approaches were tackled and compared.

6.2.2 Naive Approach

Cprof+ operates in two modes of operations for simulation. One, running the program using absolute throughput and the other using unbounded throughput. The two values give a bound on the maximum benefit in terms of performance.

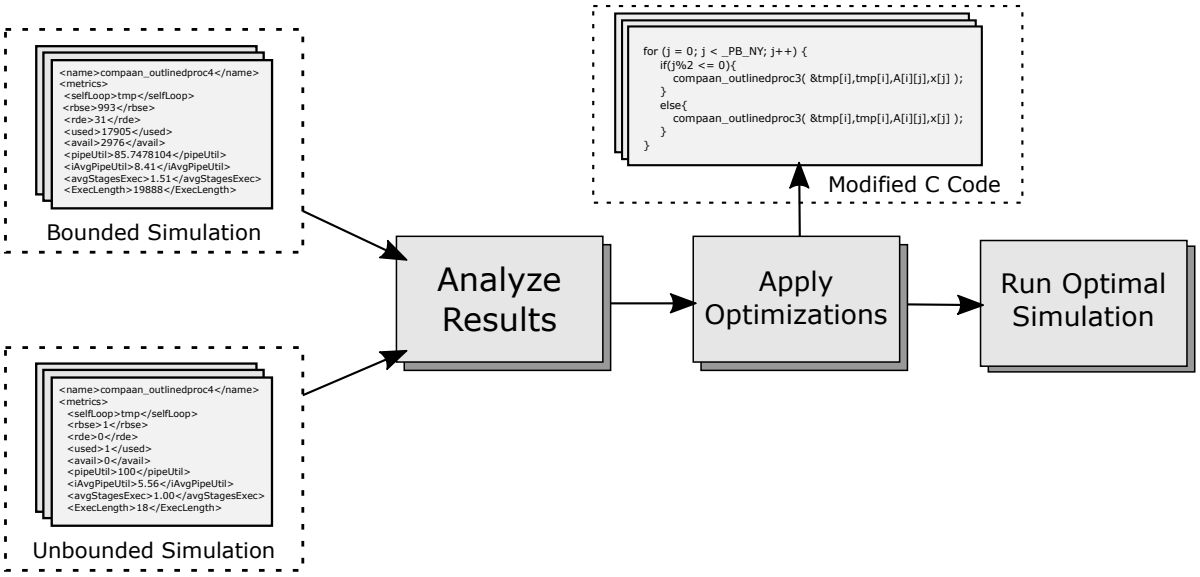


Figure 6.12: Overview of Cprof Optimization Flow

The self loop of a node is a crucial for regulating the firing of a node [12].Self loops are the consequence of a data dependency between iterations of the process. The Naive approach using Cprof+ will use the self loop metric for optimization.

6.2.2.1 Algorithm

The algorithm used to select the functions for optimization is given in Algorithm 4. A list of functions is identified an iterator and unroll factor must be added. The naive approach takes the inner loop iterator for the unroll operation and sets the unroll factor to the maximum for that iterator for a function containing a self loop.

Algorithm 4 Naive Optimization Technique

Input: List of Function Calls S
Output: List of Function Calls to optimize O

1: **for all** $s \in S$ **do**
2: **if** s contains Self Loop **then**
3: setUnfoldIterator(s, i_{inner})
4: setUnfoldFactor(s, $max(i_{inner})$)
5: $s \cup O$
6: **end if**
7: **end for**

The Algorithm 4 takes the list of all statements within a program and selects only

the statements containing self loops. The statements are transformed using modulo unfolding. The unfold factor is set to the maximum value, $max(i_{inner})$, for the innermost iterator,i_{inner}, of the loop. The time complexity of the technique is $\mathcal{O}(c)$ where c is the number of statements in the program. The space complexity is defined to be $\mathcal{O}(O)$ where O is the set of statements selected to be optimized and is a subset of S, defined as all statements contain in the program.

6.2.2.2 Results

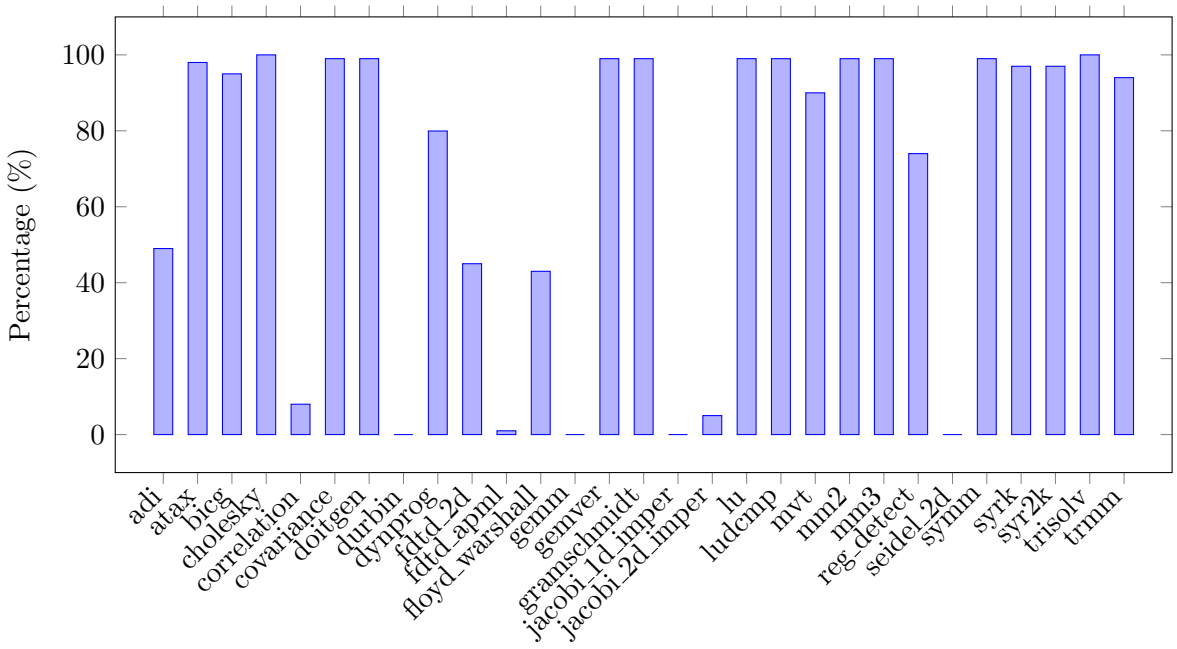


Figure 6.13: Percentage of possible performance gain achieved using the Naive approach for optimization

As the results in Figure 6.13 demonstrate, by simply optimizing the functions which contain a self loop we can reach for most benchmarks the maximum available performance gain. However, the unroll factor chosen was the maximum possible value for the given iterator and as Figure 6.10 demonstrates, unrolling a function around an iterator by its maximum value can lead to the maximum performance gain regardless of iterator direction.

One of the disadvantages of this optimization technique is the hardware requirements. Since the inner iterator of the loop is expanded to its maximum value, the number of nodes needed increases dramatically. Take the `MINI_DATASET` for the Polybench suite which on average takes iterator sizes of 32. Expanding a program and all the functions containing a self loop can increase hardware usage dramatically.

Therefore, an optimization approach that takes into account the iteration direction and optimal unrolling factor is ideal for maximizing performance gain while maintaining low hardware resource requirements.

6.2.3 Channel Size Approach

Selecting the correct unfolding factor for modulo unfolding or determining when to use plane cutting can be difficult. The work by Zissulescu-Ianculescu[12] proposed examining the size of a self loop to increase pipeline utilization. As self dependencies indicate a data dependency between iterations and cause low pipeline utilization values. By examining the amount of data available at the input to a process, an unroll factor can be calculated for that process to unlock the optimal amount of performance.

$$\Delta = min(ChannelSize_i), i = 1, .., n \tag{6.2}$$

$$F_{SL} = \left\lceil \frac{\Delta}{\Lambda_F} \right\rceil \tag{6.3}$$

Where Δ is the minimum self loop channel size of a function containing $i = 1, .., n$ self loops. There are two rules given by Zissulescu-Ianculescu[12] for when to optimize a function with a self loop.

1. $\Delta > \Lambda_F$. In this case, modulo unfolding will benefit the network as the minimum amount of data available on the self loops is greater than the functional latency. Meaning the pipeline achieves full utilization with data still ready to be processed. Adding another process to split the domain will help to increase the performance. The unfolding factor F_{SL} for modulo unfolding functions with self loops is given by Equation 6.3.

2. $\Delta \leq \Lambda_F$. When the minimum self loop size is less than the functional latency, transformations are needed to fill the pipeline with independent data. As the pipeline of the function can be hindered by a number of blocking read operations[12]. Techniques to increase the independent data in the pipeline include loop skewing[39]. The optimization factor for this technique is given as $P = \Delta - \Lambda_F$. Where P is the number of independent operations to be added.

The original Cprof possessed only two transformation techniques. Cprof+ will employ the use of channel sizes to estimate an unroll factor that will give optimal balance between hardware resources and performance using the transformations implemented by Cprof.

6.2.3.1 Algorithm

Algorithm 5 optimizes a function with a self loops based on the minimum channel size the self loops. The results of Figure 6.13 demonstrate that for some benchmarks, optimizing for the self loop does not cause performance gain. To accommodate, lines 13-21 of Algorithm optimize functions without self loops who are not defined as streaming functions.

Streaming functions are processes in a network that stream in or out the data to the network. Listing 6.5 shows an example of a input streaming function from *atax*. The implemented expression for this function is $(\&A)[j][i] = AI[j][i]$, with a functional

Algorithm 5 PPN self loop optimization using channel size

Input: List of Function Calls S
Output: List of Function Calls to optimize O

1: **for all** $s \in S$ **do**
2: **if** $!streamFunction(s)$ **then**
3: **if** s contains Self Loop **then**
4: $\Delta \leftarrow INT_{MAX}$
5: **for all** $f \in Channels_{Self}$ **do**
6: **if** $size(f) < \Delta$ **then**
7: $\Delta = size(f)$
8: **end if**
9: **end for**
10: $F_{SL} \leftarrow \left\lceil \frac{\Delta}{\Lambda_F} \right\rceil$
11: $setUnrollFactor(s, F_{SL})$
12: $O \leftarrow s$
13: **else**
14: $\Delta_{\overline{\text{stream}}} \leftarrow INT_{max}$
15: **for all** $f \in Channels$ **do**
16: **if** $!streamFunction(f) \wedge size(f) < \Delta_{\overline{\text{stream}}}$ **then**
17: $\Delta_{\overline{\text{stream}}} = size(f)$
18: **end if**
19: **end for**
20: $F_{\overline{\text{SL}}} \leftarrow \left\lceil \frac{\Delta_{\overline{\text{stream}}}}{\Lambda_F} \right\rceil$
21: $setUnrollFactor(s, F_{\overline{\text{SL}}})$
22: $O \leftarrow s$
23: **end if**
24: **end if**
25: **end for**

Listing 6.5: Example of a streaming function from *atax*

```
1 for (j = 0; j < _PB_NX; j++) {
2     for (i = 0; i < _PB_NY; i++) {
3         compaan_outlinedproc0(&A[j][i], AI[j][i]);
4     }
5 }
```

latency $\lambda_F = 1$. The streaming functions do not affect the performance of the network and can be filtered out. Algorithm 5 demonstrates this on line 2.

The channel sizes from streaming functions are disregarded for functions without self loops. The reason is the streaming functions do not have any dependencies and run at full capacity. As opposed to other functions which have data dependencies and input channels. The channels sizes from these functions give a better idea of the unfold factor that should be used. Equation 6.4 highlights the unfold factor used for functions without self loops. Where $\Delta_{\overline{\text{stream}}}$ is the minimum channel size of all channels that do

not originate from a streaming function.

$$F_{\overline{\text{SL}}} = \left\lceil \frac{\Delta_{\overline{\text{stream}}}}{\Lambda_F} \right\rceil \tag{6.4}$$

The functions that do posses a self loop are optimized using the factor calculated in lines 4-12 of Algorithm 5. Using the Equations of 6.2 and 6.3 an ideal unfold factor is calculated.

For functions containing both self loops and no self loops, the modulo unfolding and plane cutting transformations are used. When the unfold factor is greater than 2 the modulo unfolding transformation is used. When the factor is less than the plane cutting transformation is used. Cprof cannot currently utilize transformations that help to insert independent data into the pipeline as suggested in Section 6.2.3. In both cases when $\Delta > \Lambda_F$ and $\Delta \leq \Lambda_F$, the modulo unfolding or plane cutting transformations are used.

6.2.3.2 Results

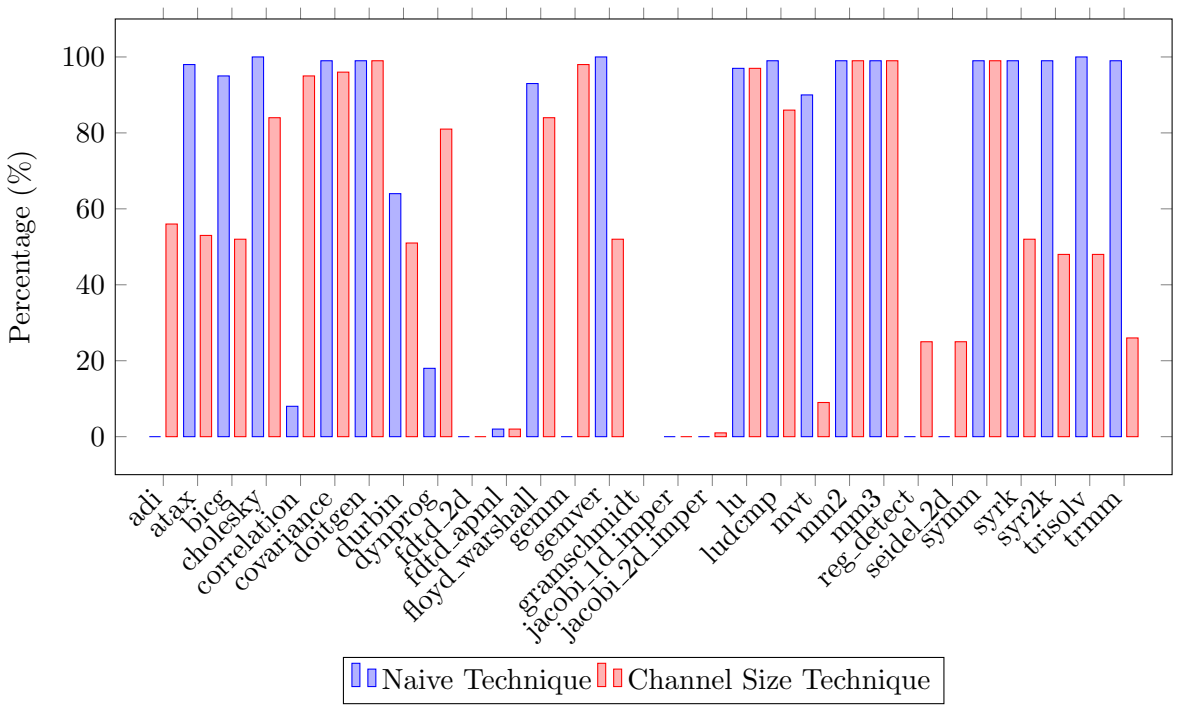


Figure 6.14: Percentage of Possible Performance Gain Achieved using Channel Size Technique compared to Figure 6.13

The results of the optimization using minimum channel size of a function is shown in Figure 6.14. The graph demonstrates the amount of possible performance achieved is in some case similar to that of the naive technique. In some cases only 54% of the possible performance is achieved when compared to the previous technique, e.g. benchmark *atax*. The percentage of possible performance is calculated by the equation, $Performance(\%) = 100 * (Absolute - Unbounded)/(Absolute - Optimized)$. For

example, *atax* obtained a value of 10030 cycles using the channel size technique. The absolute and unbounded cases yielded 19948 and 1286 cycles respectively. Resulting in the total amount of possible performance achieved to be $Performance(\%) = 100 * (19948 - 10030)/(19948 - 1286) = 54\%$.

One point to note, for the benchmark *gramschmidt*, the optimal version has a smaller execution time in cycles than the unbounded. The optimized version gives a cycle count of 2275 while the estimated unbounded case gives 3335 cycles. The case of the error was not studied as the unbounded case is by definition the upper limit on performance.

To compare the performance gains of the two techniques, another metric must be introduce. To give an idea on the hardware resources, Cprof+ counts the amount of processes in the network to give a first approximation of hardware costs. Figure 6.15 demonstrates the amount of processes added to the network due to the two different optimization techniques. The graph demonstrates for most cases the amount of hardware added using the channel size optimization technique is less than the naive optimization.

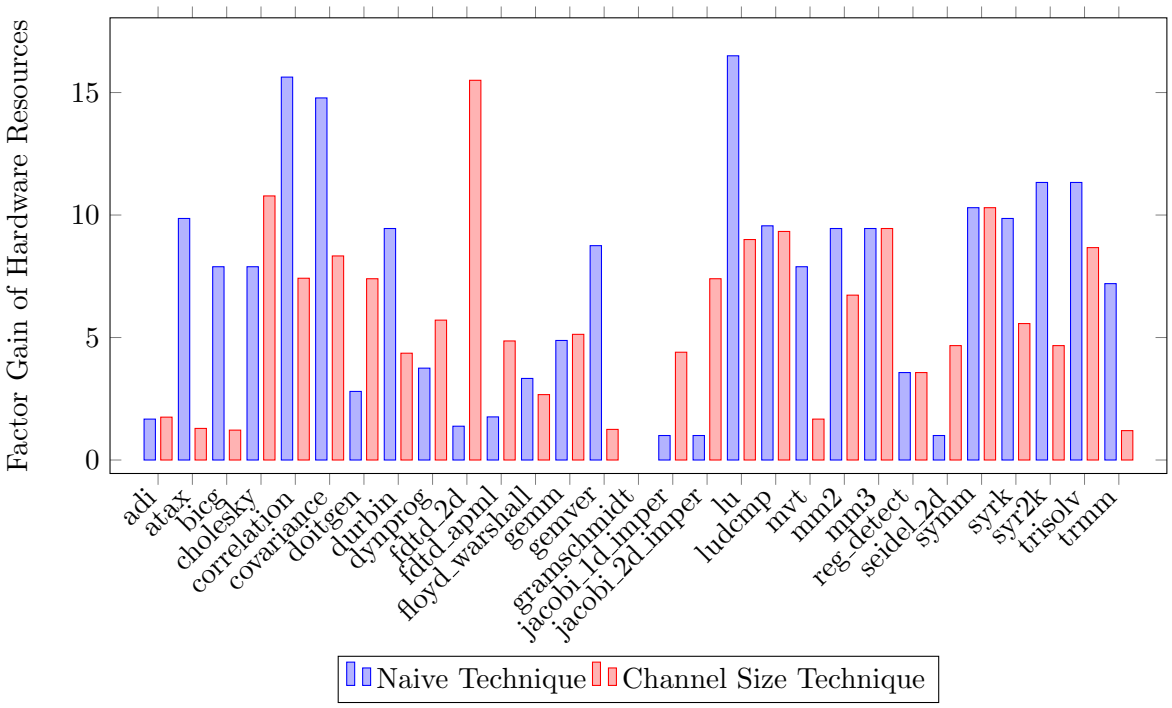


Figure 6.15: Factor gain of Processes added using Channel Size Technique and Naive Approach

The benefit of using the minimum channel size of a process for an optimization factor can be seen in the average cost and performance gain. Figure 6.15 shows this by demonstrating the Factor Gain of Hardware Resources (FG_{hw}).

$$FG_{hw} = (\#\ processes\ in\ optimized)/(\#\ processes\ in\ absolute) \tag{6.5}$$

On average, using the channel size to estimate a unroll factor yields, a 64% performance increase with an average of 5.91x increase in hardware resources when compared

to the absolute case. While the naive approach utilizes 7.12x more hardware resources with only 63% performance increase on average.

The selection of the iterator for optimizing is crucial. The naive approach considered the inner loop index, which in most cases allows for significant performance gain when optimizing a function with a self loop. The channel size technique selected the iterator based on the inner iterator of the writing variable. For example, a function $foo(\&A[i][j], B[k], A[j][i])$, will have the j iterator selected as the write variable is $\&A[i][j]$. While the naive approach may select the iterator k if this function is surrounded by three loops, with k as the inner most loop.

Benchmark *gemm* is an example of the importance of iterator selection for optimization. The hardware resources used in *gemm* for the channel size approach is 41, while the naive approach utilizes 39 processes. The performance gain is 98% and 0% for the channel size and naive approach respectively. Highlighting the importance of the iterator optimization direction.

6.3 Summary

This chapter presents Cprof+ as a profiler for selecting functions within a program to optimize. Sections 6.1.1.1 and 6.1.1.2 highlight the correct selection of process nodes within the network is crucial for obtaining the most performance per hardware resource. While Section 6.1.2 highlights the importance of data dependencies between nodes for a PPN. As an arbitrary small or large functional latency does not mitigate the effect of a crucial node on the network performance, nor does it enhance the effect of a node that is not on the critical path.

Finally, employing the source-to-source transformations in Section 6.1 demonstrate that a program can be quickly transformed using modulo unfolding and plane cutting. The first technique unfolded an iterator to the maximum value that yielded designs that obtain on average positive performance gains. Using the minimum channel size of a process to estimate the unroll factor and selecting the inner iterator of the write variable showed similar performance gains with reduced resource costs.

Both techniques utilized only 4 metrics of the function for selection: function latency, self loop, channel sizes and if the function was a streaming function. Section 6.2.1 demonstrated that a variety of metrics for a function execution can be given. Further investigation will have to be given to determine which metrics are relevant for identifying functions to optimize.

7 Cprof Simulation Time

To allow a designer to quickly evaluate a program implementation, Cprof+ needs to be efficient and quick. Haastregt[1] compared different types of PPN performance estimation methods including RTL, SystemC, Mean Cycle Method and Cprof. It was suggested that Cprof could provided accurate estimations with a small runtime and effort compared to RTL and SystemC simulations[1].

The following sections demonstrate the run times of Cprof+ and the RTL simulation. To show that Cprof+ in real cases can indeed provide accurate simulation results within a short amount of time.

The experiments shown in the following sections have all been run using a Intel i7 Q720 operating at 1.60Ghz with 8GB of internal memory using a single thread. The RTL simulator used for comparison was the Xilinx ISE Simulator (ISE) included in the Webkit Vivado HLS Suite 2015.4 edition. All benchmarks were run using the `MINI_DATASET`.

7.1 Cprof Run Times

Table 7.1 demonstrates the various run times of Cprof+ for the different benchmarks of the Polybench Suite. As suggested, for most benchmarks, the simulation time of Cprof+ is within seconds yielding an accurate performance estimation with little effort.For some of the benchmarks the small run-time does not hold and there exists large simulation times, with a maximum of 790 seconds and an average of 122 seconds.

Although it was suggested in the work of Haastregt that Cprof can execute in the range on seconds when compared to RTL simulations, Table 7.1 suggests that with the modifications, this is not the case. The discrepancy can be attributed to the feature that Cprof+ currently has implemented, that is channel size estimation based on execution profiles. In the original work of Haastregt, Cprof was not intended to calculate channel sizes of the PPN, but with the available information Cprof+ can. Naturally, this adds overhead to the instrumentation code that is added to any benchmark and the time complexity is a function of the variable write history as shown in Section 5.1.2. Thus, large iteration domains can lead to longer computation time for Cprof+.

This can be clearly seen in the values of the Optimal Simulation that is run using the technique given in Section 6.2.3. With the increased number of process given to one function of the program we can see the time to simulate them has decreased. For example, benchmark *gemm* in Table 7.1 has an Absolute Throughput simulation time of $368s$ while the optimized version has a run-time of $161s$. This is attributed to the fact, that the variable write histories are reduced in size. Therefore, the computational effort for channel size estimation is reduced as the number of elements to be searched is reduced. Even though the number of channels sizes to calculate has increased.

Table 7.1: Simulation Times of Cprof+ in 3 different modes

	Cprof Simulation Type			Vivado
Benchmark	Absolute (s)	Unbounded (s)	Optimal (s)	RTL (s)
adi	3	3	4	666
atax	4	3	3	114
bicg	5	3	4	128
cholesky	15	4	6	181
correlation	72	6	24	419
covariance	66	4	19	185
doitgen	19	4	11	114
durbin	3	3	3	199
dynprog	626	13	43	161
fdtd_2d	7	3	11	160
fdtd_apml	9	3	8	11988
floyd_warshall	3	2	3	110
gemm	368	8	161	148
gemver	7	3	6	243
gramschmidt	3	2	6	174
jacobi_1d_imper	3	3	3	100
jacobi_2d_imper	4	3	3	100
lu	13	4	8	105
ludcmp	28	4	13	116
mm2	447	10	47	95
mm3	547	13	64	132
mvt	4	2	2	208
reg_detect	3	2	3	155
seidel_2d	11	2	14	242
symm	127	6	28	100
syrk	277	6	29	135
syr2k	790	10	208	251
trisolv	3	2	3	251
trmm	63	5	41	363

7.2 Comparison with RTL Simulation Times

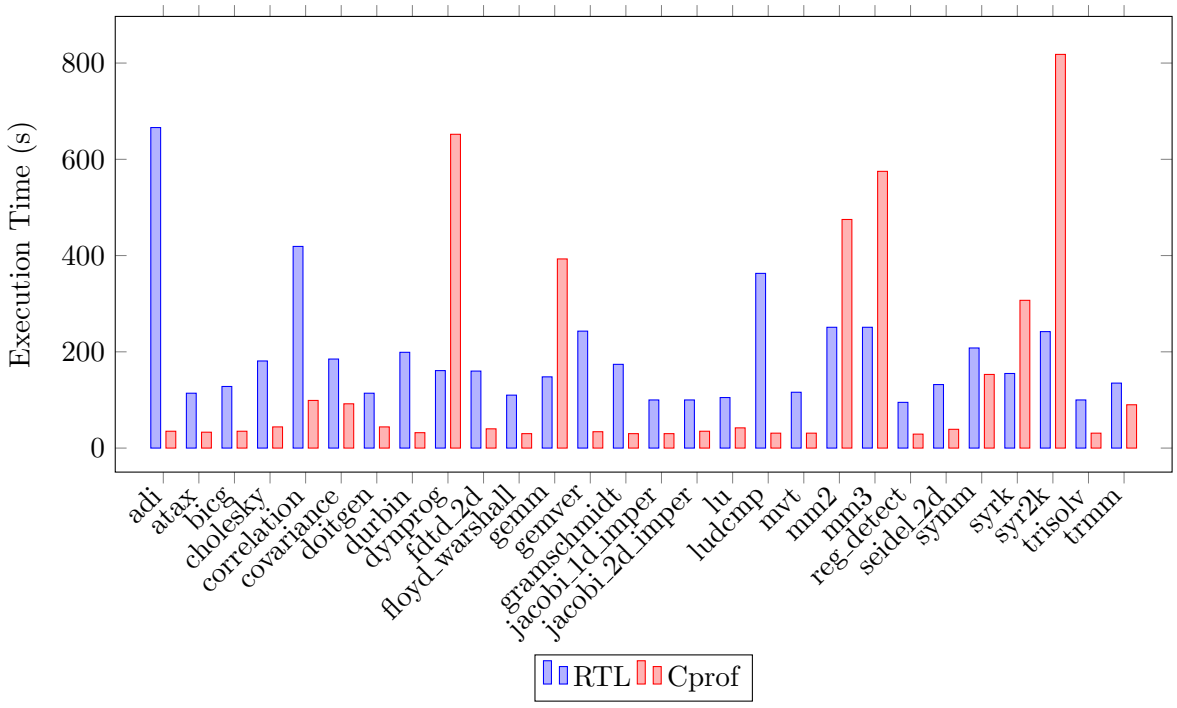


Figure 7.1: Comparing Cprof+ and Compaan+RTL Simulation Times for the Polybench Suite

The comparison of the RTL simulations with Cprof+ was conducted by executing the complete design flow as described in Figure 2.2. The Compaan/LAURA tool chain was used to generate a PPN from the benchmark C code and then RTL simulations were run using the generated RTL description of the PPN. The results of Figure 7.1 demonstrate that Cprof+ for the most part executes much quicker than the RTL simulations. With Cprof+ simulations requiring 122s and Vivado RTL simulations requiring 598s on average to complete. The values of Cprof+ used in Figure 7.1 is an aggregate of the static analysis, dynamic anaylsis and the compile time of the simulation executable. Only for benchmarks such as *dynprog, gemm, mm2, mm3, syrk and syr2k* do we see Cprof+ executes much slower than the RTL counterpart.

Another interesting experimental point, is that for the benchmark, *fdtd apml* the execution time for Cprof+ was 42s while the RTL simulation took 11988s. The benchmark was ommitted from Figure 7.1 due to the large range. An odd abnormality, but it does demonstrate the hardware complexity of the implementation can cause enormous simulation time.

8 Conclusion

In this thesis, we present Cprof+. This profiler is capable of simulating polyhedral process networks without explicitly deriving the PPNs. Cprof+ is designed as tool that will allow the designer to explore the design space rapidly for a PPN. The PPN that Cprof+ models, is derived from high-level specifications. The C programs used to validate Cprof+ and Cprof were the Polybench suite benchmarks. This a collection of 29 data streaming and matrix transformation programs which contain static control parts.

The original Cprof showed an accurate performance estimation of the network execution to within 1%[5], but not for all benchmarks. The performance estimations for *gemm* and *dynprog* showed 65% and 26% underestimation. To correct this inaccuracy, Cprof+ had to model the Out of Order Memory channel between processes which the original Cprof did not. The OOM channel in the case of the Compaan/LAURA PPN description was found to have a longer read latency of 3 cycles instead of 1 cycle. Incorporating the read latency of an OOM channel in Cprof+ was accomplished with run time analysis of the read time, write time and the indices history. The results yield network performance estimations errors of less than 6% for all Polybench Suite benchmarks. In other words, Cprof+ is able to achieve 94% accuracy in performance estimations compared to RTL simulations

The accurate performance estimation allows Cprof+ to profile a large set of programs. The Cprof+ simulation keeps profile of each statement execution including read, execute and write times. These statement profiles allow for calculation of new metrics, including the *conditional synchronization* time and the estimation of channel sizes between processes. One of the difficulties for PPNs is the ability to calculate minimum channel sizes while maintaining a deadlock free schedule. Cprof+ profiler calculates channel sizes that, which not proved to be minimum, are significantly less than the sizes calculated by Compaan.

With the channel size estimations, Cprof+ is able to perform a novel optimization technique. Using the channel sizes as an estimate for the optimization factor of a function, Cprof+ is able to achieve on average a 64% increase in performance using modulo unfolding and plane cutting source-to-source transformations. While maintaining an average factor of 6x increase in hardware resources as compared to the absolute throughput case. A naive optimization technique yielded similar performance gains, but with 7x factor increase in hardware resources.

The addition of channel size estimation and run-time checks for OOM adds overhead to the Cprof+ profiling simulation. To ensure Cprof+ maintains reduced simulation time, the Cprof+ and RTL simulations were compared. The results show that on average Cprof+ executes within 121 seconds, while RTL simulations yield a time of 598 seconds. This result demonstrates the capability of Cprof+ as a rapid performance estimation tool for PPN implementations in hardware.

8.1 Contributions

The contributions made in this thesis are as follows.

1. **A profiler capable of modelling variable functional latency.** Cprof+ now has the ability to model different functional latencies to accurately model the execution of a process in a PPN.

2. **Run time analysis of channel type.** To improve the performance estimations of PPNs with OOM channels, Cprof+ analyses the read and write behaviour to adjust in run time whether a read incurs an extra latency.

3. **Channel size calculation based on simulation profile.** Cprof+ can calculate the channel sizes for a process of the PPN using the read and write profiles of the process input variables. The size gives a first order approximation of the memory requirements of the channels of a PPN. These channels can be implemented as FIFOs or re-ordering buffers.

4. **Automated source-to-source optimization technique for assisting in design space exploration.** We have presented an automated method for assisting the designer in exploring the design space by utilizing the input channel sizes of a process to estimate an optimization factor. The factor is used to determine whether to use modulo unfolding or plane cutting transformation techniques.

5. **Validation of performance estimation and channel sizes.** The results of the modification were validated using the Compaan/Laura tool chain. The Xilinix Vivado HLS Suite was used to verify the hardware implementation. The simulation results were then compared to the values computed by Cprof+.

8.2 Future Work

In the following section suggestion for improvement will be given.

Expanding Transformation Tools

Cprof+ maintains two transformation techniques that it can apply to the source code in order to change the PPN. There exist many more loop transformations that can help increase the performance of the network without incurring additional hardware resources. These loop transformations include *loop skewing, loop interchange* and *stream multiplexing*. Chapter D of the appendix gives a detailed background of the transformations.

Investigating Optimal Transformation Techniques

The automated transformation technique given focused only upon two transformations that are well suited for processes with high pipeline utilization. To perform intelligent design space exploration a combination of code transformations must be used to increase the amount of data available to the process. The paper of Meijer[38] showed

performance increases by utilizing a combination of process splitting and process merging techniques on PPNs for MPSoC applications.

Improving Programming Language Support

Cprof+ supports only programs with statements modelled as functions. Supporting programs with non-functional call statements allows for larger set of programs that can be profiled. To increase the set of programs that can be profiled, Cprof+ can be extended to also support parallel programming languages as other profilers such as Parwiz[40] do. Supporting parallel languages may also alleviate the demand of Cprof+ to identify parallel regions as the designer can already specify parallel code. Allowing Cprof+ to give a more accurate estimation of the design space.

Improving Channel Sizing Calculation

Cprof+ currently implements channel sizing calculations post-simulation using arrays of read and write histories. The memory and performance requirements of storing and searching the histories to obtain a channel size increase with problem size and network complexity. Compression techniques and adoption of a run-time size calculation can aid in reducing the time complexity of Cprof+.

Bibliography

[1] S. van 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, Leiden, Dec. 2013.

[2] R. Sun, X. Wang, and X. Ye, "Real-time pedestrian detectionl using opencl," in *2014 International Conference on Audio, Language and Image Processing*, pp. 401–404, July 2014.

[3] A. Yilmaz, O. Javed, and M. Shah, "Object tracking: A survey," *ACM Comput. Surv.*, vol. 38, Dec. 2006.

[4] W. van Teijlingen, "Determining performance boundaries and automatic loop optimization of high-level system specifications," msc. thesis, TU Delft, Delft, The Netherlands, Nov. 2014.

[5] W. van Teijlingen, R. van Leuken, C. Galuzzi, and B. Kienhuis, "Determining performance boundaries on high-level system specifications," in *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, (New York, NY, USA), pp. 90–97, ACM, 2016.

[6] B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: deriving process networks from matlab for embedded signal processing architectures," in *Hardware/-Software Codesign, 2000. CODES 2000. Proceedings of the Eighth International Workshop on*, pp. 13–17, May 2000.

[7] S. S. Bhattacharyya, *Handbook of signal processing systems*.

[8] S. Meijer, *Transformations for Polyhedral Process Networks*. PhD thesis, Leiden University, Leiden, Dec. 2010.

[9] C. Zissulescu, B. Kienhuis, and E. Deprette in *Proceedings of the 14th International Conference on Field- Programmable Logic and Applications*.

[10] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using kahn process networks: The compaan/laura approach," in *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '04, (Washington, DC, USA), pp. 10340–, IEEE Computer Society, 2004.

[11] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette, "System design using khan process networks: the compaan/laura approach," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 1, pp. 340–345 Vol.1, Feb 2004.

[12] C. Zissulescu-Ianculescu, *Synthesis of a parallel data stream processor from data flow process networks*. PhD thesis, LIACS Embedded Research Center, Leiden, Nov. 2008.

[13] C. Lattner, "Llvm and clang: Next generation compiler technology." Poster, 2008.

[14] S. Naroff, "New llvm c front-end." Poster, 2007.

[15] J. C. Mazo, *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. PhD thesis, RWTH Aachen, Aachen, Apr. 2013.

[16] 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 *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 574–579, June 2008.

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

[18] P. Roy and X. Liu, "Structslim: A lightweight profiler to guide structure splitting," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, (New York, NY, USA), pp. 36–46, ACM, 2016.

[19] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, pp. 120–126, June 1982.

[20] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, pp. 89–100, June 2007.

[21] H. K. M. Kim and C. Luk, "Prospector: A dynamic data-dependence profiler to help parallel programming," in *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*, HotPar'10, 2010.

[22] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor, "Kismet: Parallel speedup estimates for serial programs," *SIGPLAN Not.*, vol. 46, pp. 519–536, Oct. 2011.

[23] C. L. D. Joen, S. Garcia and M. Taylor, "Parkour: Parallel speedup estimates for serial programs," in *Proceedings of the 3nd USENIX Workshop on Hot Topics in Parallelism*, HotPar'11, 2011.

[24] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: Rethinking and rebooting gprof for the multicore age," *SIGPLAN Not.*, vol. 46, pp. 458–469, June 2011.

[25] R. Urban, M. Schlzel, H. T. Vierhaus, E. Altmann, and H. Seelig, "Compiler-centred microprocessor design (comet) - from c-code to a vhdl model of an asip," in *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2015 IEEE 18th International Symposium on*, pp. 17–22, April 2015.

[26] J. F. Eusse, C. Williams, and R. Leupers, "Coex: A novel profiling-based algorithm/architecture co-exploration for asip design," in *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*, pp. 1–8, July 2013.

[27] J. F. Eusse, C. Williams, L. G. Murillo, R. Leupers, and G. Ascheid, "Pre-architectural performance estimation for asip design based on abstract processor models," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pp. 133–140, July 2014.

[28] 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*, CODES+ISSS '09, (New York, NY, USA), pp. 305–314, ACM, 2009.

[29] 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*, PACT '04, (Washington, DC, USA), pp. 7–16, IEEE Computer Society, 2004.

[30] T. C. Grosser, *Enabling Polyhedral Optimizations in LLVM*. PhD thesis, Unviersity of Passau, Passau, Apr. 2011.

[31] K. Namjoshi and N. Singhania, "Loopy: Programmable and formally verified loop transformations," Tech. Rep. MS-CIS-16-04, University of Pennsylvania, July 2016.

[32] P. Meloni, G. Tuveri, L. Raffo, I. Loi, and F. Conti, "Online process transformation for polyhedral process networks in shared-memory mpsocs," in *2014 3rd Mediterranean Conference on Embedded Computing (MECO)*, pp. 92–97, June 2014.

[33] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson, "The effect of compiler optimizations on high-level synthesis for fpgas," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pp. 89–96, April 2013.

[34] O. S. Dragomir and K. Bertels, "K-loops: Loop skewing for reconfigurable architectures," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 199–206, Dec 2009.

[35] A. Turjan, B. Kienhuis, and E. Deprettere, "A hierarchical classification scheme to derive interprocess communication in process networks," in *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pp. 282–292, Sept 2004.

[36] T. Basten and J. Hoogerbrugge, "Efficient execution of process networks," *Communicating Process Architectures*, no. 24, 2001.

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

[38] S. Meijer, H. Nikolov, and T. Stefanov, "Combining process splitting and merging transformations for polyhedral process networks," in *2010 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, pp. 97–106, Oct 2010.

[39] T. Stefanov, B. Kienhuis, and E. Deprettere, "Algorithmic transformation techniques for efficient exploration of alternative application instances," in *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, (New York, NY, USA), pp. 7–12, ACM, 2002.

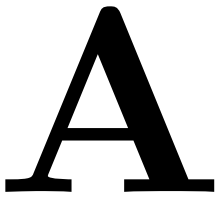
[40] A. Ketterlin and P. Clauss, "Profiling data-dependence to assist parallelization: Framework, scope, and optimization," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 437–448, IEEE Computer Society, 2012.

[41] J. Xue, "Affine-by-statement transformations of imperfectly nested loops," in *Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International*, pp. 34–38, Apr 1996.

[42] G. Jin, J. Mellor-Crummey, and R. Fowler, "Increasing temporal locality with skewing and recursive blocking," in *Supercomputing, ACM/IEEE 2001 Conference*, pp. 57–57, Nov 2001.

[43] T.-C. Huang and C.-M. Yang, "Further results for improving loop interchange in non-adjacent and imperfectly nested loops," in *High-Level Parallel Programming Models and Supportive Environments, 1998. Proceedings. Third International Workshop on*, pp. 93–99, Mar 1998.

Appendix A: Modulo Unfolding and Plane Cutting Case Studies



The following sections highlight the results of the case studies on various benchmarks using Modulo Unfolding and Plane Cutting methods.

A.1 Atax: Cprof and Compaan Results

PC Factor	Cprof	Vivado RTL
2	10028	10044
4	5068	5086
8	2588	2610
16	1689	1719
32	1689	1735

Table A.1: Plane Cutting results for compaan_outlineproc3 around iterator i up to a factor of 32 with Cprof and Compaan comparison.

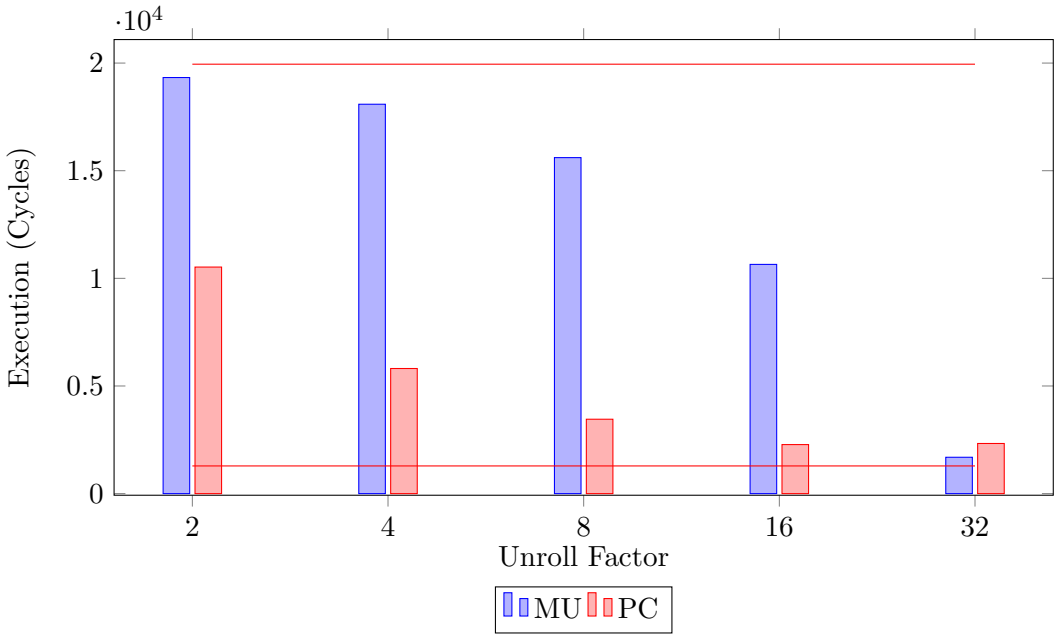


Figure A.1: Comparing Plane Cutting and Modulo Unfolding for compaan_outlinedproc3 around iterator j.

A.2 MM2: Cprof Results

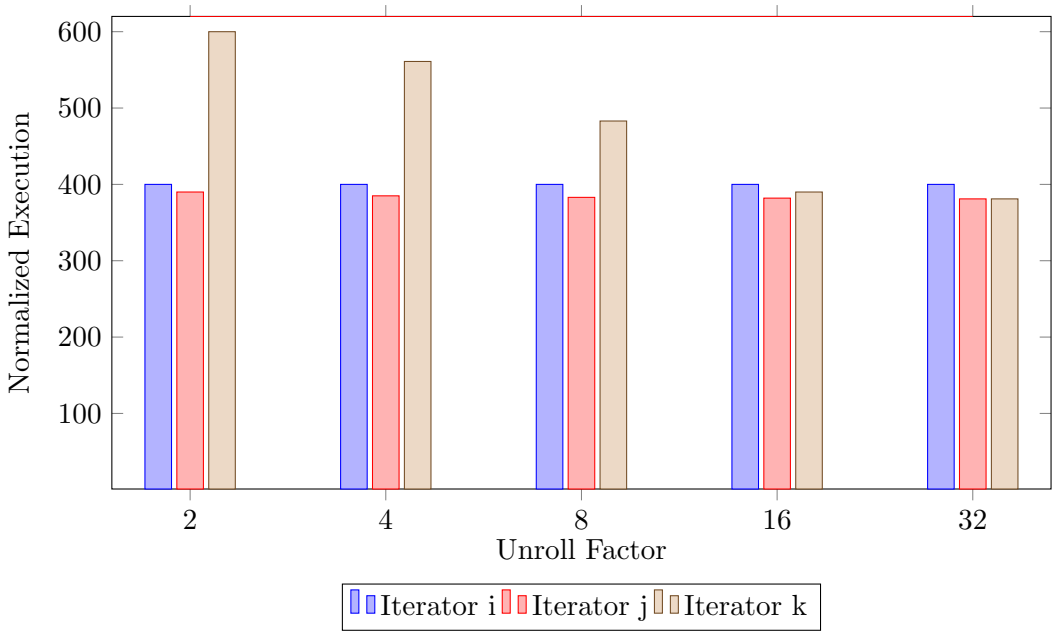


Figure A.2: Modulo Unfolding for compaan_outlinedproc7 around iterators i, j and k with execution values normalized to the unbounded case of 838 cycles.

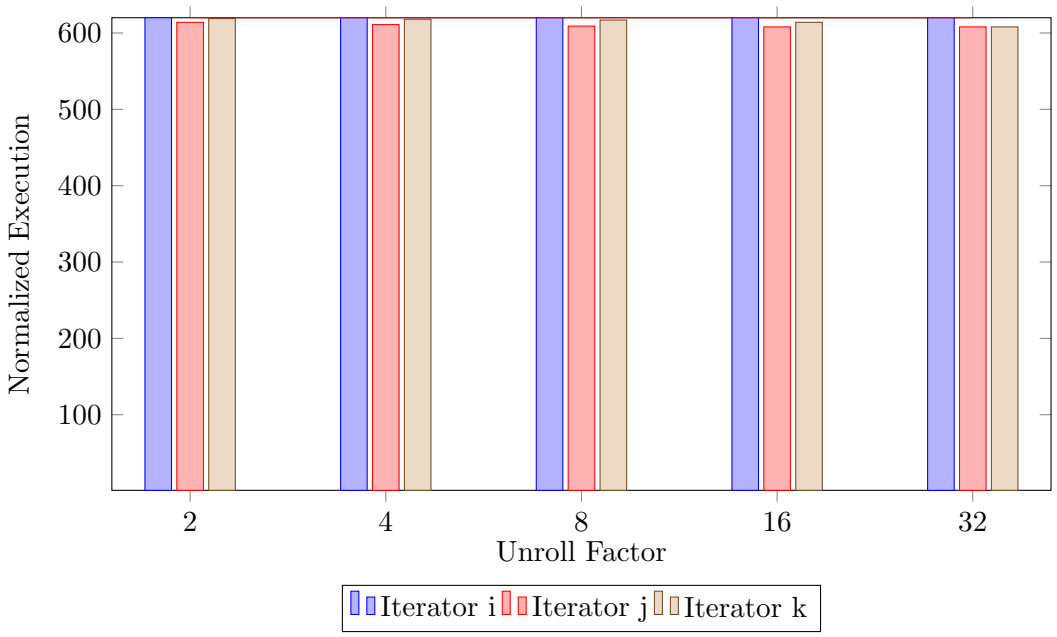


Figure A.3: Modulo Unfolding for compaan_outlinedproc9 around iterators i, j and k with executions values normalized to the unbounded case of 838 cycles.

A.3 MM2: Function Latency Experiments

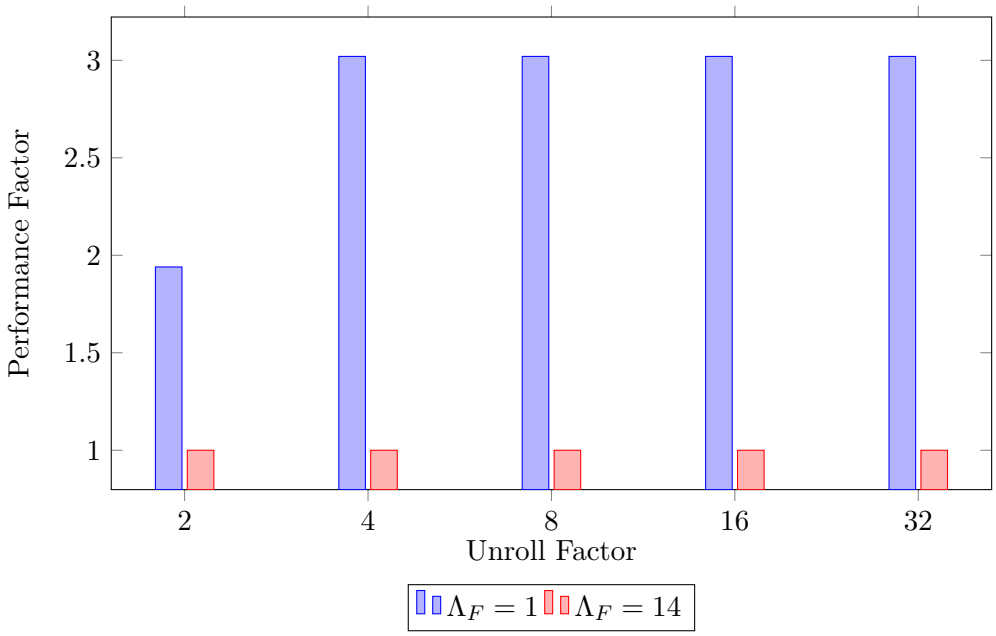


Figure A.4: Modulo Unfolding for compaan_outlinedproc9 around iterators i with varying function latency of Proc7.

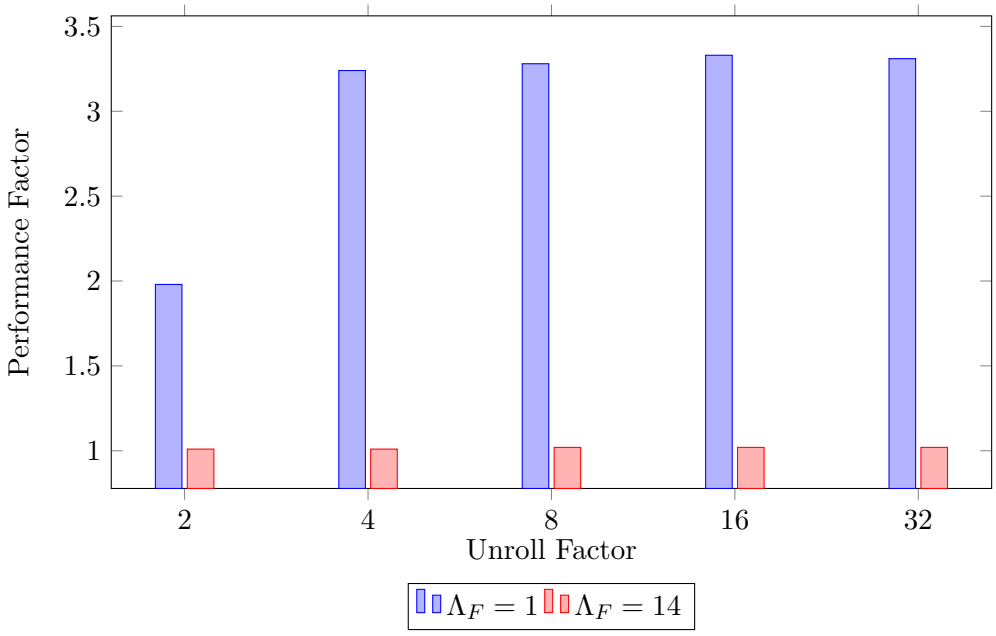


Figure A.5: Modulo Unfolding for compaan_outlinedproc9 around iterators j with varying function latency of Proc7.

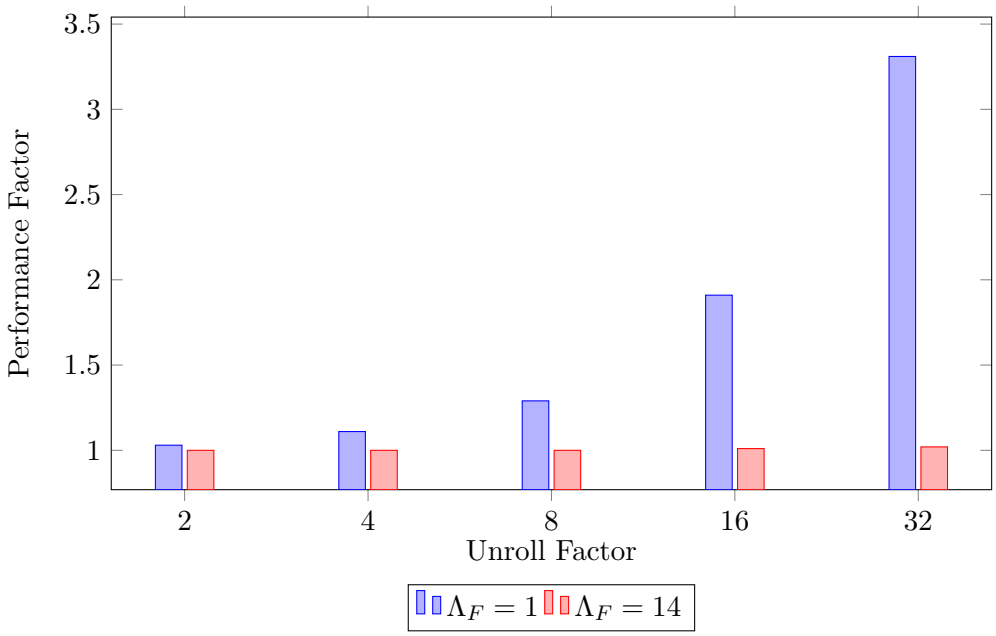


Figure A.6: Modulo Unfolding for compaan_outlinedproc9 around iterators k with varying function latency of Proc7.

Appendix B: Optimization Techniques Numerical Results

B

Benchmark	CS(cycles)	CS Nodes	Naive(cycles)	Nodes	Absolute(cycles)	Abs Nodes	UB(cycles)	UB Nodes
adi	3,076	21	5,620	20	5,596	12	1,094	960
atax	10,030	9	1,689	69	19,948	7	1,286	3,200
bicg	9,977	11	1,638	71	19,897	9	646	3,264
cholesky	17,903	97	4,166	71	91,500	9	4,166	36,928
correlation	17,431	141	310,288	297	338,498	19	2,121	41,121
covariance	13,700	75	3,224	133	341,992	9	1,085	38,976
doitgen	1,212	37	1,210	14	181,028	5	209	14,000
durbin	10,755	48	10,743	104	10,803	11	10,712	2,179
dynprog	37,750	40	161,375	25	242,398	7	289	316,522
fdtd_2d	2,119	124	2,649	11	2,187	8	150	10,304
fdtd_apml	4,363	180	4,363	65	4,508	37	725	9,331
floyd_warshall	638	8	412	10	2,000	3	292	640
gemm	2,115	41	99,110	39	99,172	8	46	37,890
gemver	27,256	20	1,993	140	55,034	16	1,838	4,418
gramschmidt	2,275	45	3,344	30	4,746	9	3,335	1,360
jacobi_1d_imper	1,029	22	1,031	5	1,031	5	65	3,491
jacobi_2d_imper	1,873	37	1,899	5	1,899	5	19	6,483
lu	2,857	36	2,804	66	31,364	4	1,866	35,840
mvt	9,781	15	1,318	71	10,711	9	326	3,264
reg_detect	2,759	51	3,687	25	3,649	7	143	664
seidel_2d	$1.38 \cdot 10^5$	14	181,228	3	181,228	3	9,574	2,822
symm	3,001	88	3,001	103	281,351	10	1,510	69,634
syrk	4,486	39	4,486	69	508,958	7	527	36,864
syr2k	$4.93 \cdot 10^5$	42	4,998	102	1,016,878	9	1,039	70,656
trisolv	5,854	52	1,909	68	9,497	6	1,909	2,144
trmm	$3.12 \cdot 10^5$	6	17,118	36	418,625	5	1,742	35,841
mm2	2,421	74	2,421	104	521,204	11	838	7,168
mm3	3,159	104	3,159	104	631,713	11	646	106,496
ludcmp	28,873	168	15,427	172	123,680	18	13,905	75,010

Table B.1: Numerical Results of the Optimization Techniques with Estimated Execution Time of the implemented PPNs in cycles. In addition, to amount of node(processes) allocated for the network

In the above table, the results of the two optimization techniques are given and compared with each other. Note the abbreviation CS stands for the Channel Size technique. While Absolute and UB represent the Absolute and Unbounded throughput results of the Cprof PPN execution estimation on each benchmark. The amount of processes or nodes needed for each network are also given in the adjacent column of each execution estimation. The number of processes were also estimated from Cprof.

Appendix C: Polybench Suite

C

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	Multiresultion analysis kernel (MADNESS)
durbin	Toeplit System Solver
dynprog	Dynamic Programming (2D)
fdtd-2d	2-D Finite Different Time Domain Kernel
fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer
gemm	Matrix-multiply C=alpha*A*B+beta*C
gemver	Vector Multiplication and Matrix Addition
gramschmidt	Gram-Schmidt Decomposition
jacobi-1d	1-D Jacobi Stencil Computation
jacobi-2d	2-D Jacobi Stencil Compuation
lu	LU Decomposition
ludcmp	LU Decomposition
mm2	2 Matrix Multiplication (D=A*B; E=C*D;)
mm3	3 Matrix Multiplication (E=A*B; F=C*D; G=E*F;)
mvt	Matrix Vector Product and Transpose
reg-detect	2-D Image Processing
seidel	2-D Seidel Stencil Computation
symm	Symmetric Matrix Multiply
syrk	Symmetric rank-k Operations
syr2k	Symmetric rank-2k Operations
trisolv	Triangular Solver
trmm	Traingular Matrix Multiply

Table C.1: Polybench Benchmark Suite

Appendix D: Loop Transformation Techniques

D

Source-to-source transformations are nothing new for loops. Loop distribution, loop fusion, loop skewing and loop interchange allow for a better traversal of a loop iteration domain to exploit performance gains. The difficulty is applying these transformations in a systematic and general manner. One method of applying the transformations generally is by using unimodular transformations on loops. Unimodular transformations allows for simple manipulation of loops, but data dependencies between statements can mean unimodular transformations cannot be used[41].

D.1 Loop Skewing

Loop skewing is a transformation technique that modifies the schedule of a process by shifting the iterations in order to explicitly exploit the parallelism available. Figure D.1 demonstrates the transformation visually on a 2-dimensional iteration domain with dependencies shown with arrows. By skewing the iteration domain in the j direction, the throughput of the system is increased.

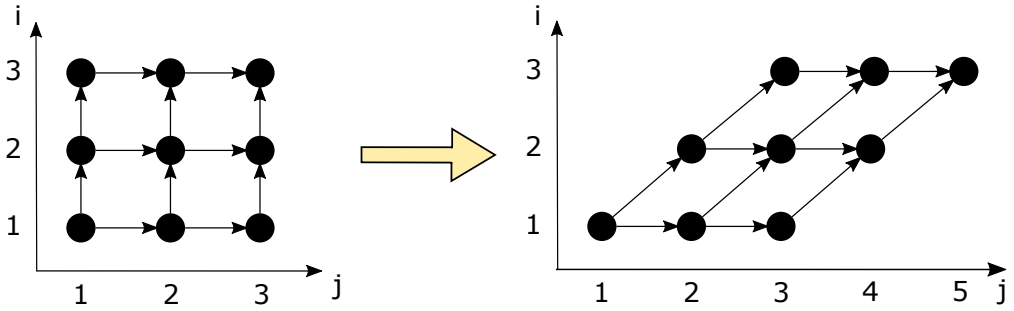


Figure D.1: Loop Skew applied to an Iteration Domain

Take for example iterations $(1, 2)$ and $(2, 1)$, both are independent as no dependencies exist between them. Therefore both can be executed in sequence, rather than utilizing a lexicographical schedule where $(1, 2)$ is executed followed by $(1, 3)$. This leads to a stall in the pipeline as $(1, 3)$ will have to wait on the output of $(1, 2)$. Thus, iteration domain can be traversed in a more optimal fashion.

Listing D.1 demonstrates the modified C code in this example which helps to explicitly state the desired traversal order. The work of Stefanov[39] presents methods for transforming a loop statement by skewing using a mathematical description of the statement. With this transformation Compaan is then able to process the modified C code and implement the modified process[39].

Loop skewing is an important technique to increase the hardware utilization of a

Listing D.1: Loop Skewed C Code from example in Listing 2.1

```
1 for(i=2; i < N+3; i++){
2     for(j=max(1,i-3); j < min(i,4); j++){
3         A[i-j][j] = foo(A[i-j][j-1], A[i-j-1][j]);
4     }
5 }
```

process. The interest to improve the efficiency of loops is an important factor for scientific programs and image processing application[42]. The data reuse is an important factor when utilizing all the performance possible in a loop. The work present by Jin et al. demonstrate a method for General Purpose Processors that improve the temporal locality of data in a loop for scientific applications[42]. Reducing cache misses by 27%.

The difficulty to find an appropriate skew factor is also challenge as seen in the work of Loopy[31]. Where the authors manually found the optimal skew factor for the *seidel-2d* benchmark of Polybench. However, it must be noted their research dealt only with GPPs architectures and the findings may not be applicable to a PPN performance.

D.2 Loop Interchange

Loop interchange is another code transformation that can lead to an improved schedule for the process. Figure D.2 demonstrates a loop interchange on a perfectly nested loop. A perfectly nested loop is one where all statements are contained within the inner most loop.

Listing (D.2) Loop Order i,j

```
1 for(i=0; i < N; i++){
2     for(j=0; j< N; j++){
3         A[i][j] = A[i+1][j-1]+5;
4     }
5 }
```

Listing (D.3) Loop Order j,i

```
1 for(j=0; j < N; j++){
2     for(i=0; i< N; i++){
3         A[i][j] = A[i+1][j-1]+5;
4     }
5 }
```

Figure D.2: Example of Loop Interchange

Perfectly nested loops can be transformed using a unimodular matrix transformation. A unimodular matrix is one where the determinant is +1 or -1 one. The mathematical description of the unimodular transformation for loop interchange of a 2-dimensional loop is given as follows.

$$TI = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i' \\ j' \end{bmatrix} = I' \tag{D.1}$$

Where i' and j' are the new iteration vectors of the loop and T the transformation matrix. This transformation must be done on all iterations points of the loop. In order for this transformation to be legal, it must pass a dependency test. The test checks

Listing D.4: Example of a cycle in a PPN

```
for(i=0; i < N; i++){
    proc1(&v, v);
    proc2(&v, v);
    proc3(&v, v);
}
```

for whether the dependence vectors for each iteration point is lexicographically positive after the transformation. A dependence vector is defined as follows.

$$\theta_k = \begin{cases} 1 & \text{if} \quad u_k > 0 \\ 0 & \text{if} \quad u_k = 0 \\ -1 & \text{if} \quad u_k < 0 \end{cases} \tag{D.2}$$

Where $u = (i_p, j_p) - (i_c, j_c)$ is the distance vector between the producer and consumer and $k \in \{i, j\}$. The dependence vector for iteration $(1, 2)$ of variable A can be seen as a distance vector from $(1, 2)$ to $(2, 1)$. Creating a dependence vector of $(1, -1)$.Applying transformation T to the dependence vector yields a new dependence vector $(-1, 1)$. The resulting dependence vector is $(1, -1) \succ (-1, 1)$ and so it is lexicographically negative. Applying loop interchange on the loop of Figure D.2 is then deemed illegal.

The example given demonstrates the need for transformation legality as the loop interchange affects the scanning direction of an iteration domain and so care must be taken to ensure the dependencies are not affected.

The transformation can be expanded to non-adjacent perfectly nested loops of dimension greater than 3 and imperfectly nested loops. In the paper of T. Huang et al.[43] techniques were given on how to apply the transformation to adjacent and imperfectly nested loops.

D.3 Stream Multiplexing

The method of PPN optimization in Section 6.2 explained in that in the PhD thesis of Zissulescu[9] a process in a PPN with a self loop of size less than the functional latency must be modified to increase the amount of independent iterations. This can be accomplished using skewing or loop interchange or by adding independent streams of the same problem.

In the case of a PPN that contains a cycle, overlapping execution may be prevented due to the cyclic nature of the network. To reduce the idle time of process, independent data streams can be added, this is known as stream multiplexing.

In the PhD Thesis of Haastregt[1], this transformation was proposed to increase the throughput of multiple executions of a PPN. By allowing multiple independent data sets to execute in a pipelined fashion.

The technique is similar to software pipelining where iterations of subsequent loops are executed in an overlapping pattern. However, in this case the pipelining is occurring at a task level.

Listing D.5: Streaming Multiplexing C code Example from Listing D.4

```
for(i=0; i < N; i++){
    for(j=0; j<P; j++){
        proc1(&v[j], v[j]);
    }
    for(j=0; j<P; j++){
        proc2(&v[j], v[j]);
    }
    for(j=0; j<P; j++){
        proc3(&v[j], v[j]);
    }
}
```

The example in Listing D.4 depicts a PPN of three processes each dependent on the output of the other. The input of `proc2` is dependent on the output of `proc1` and `proc3` on `proc2`. This causes the utilization of the processes pipeline to be low.

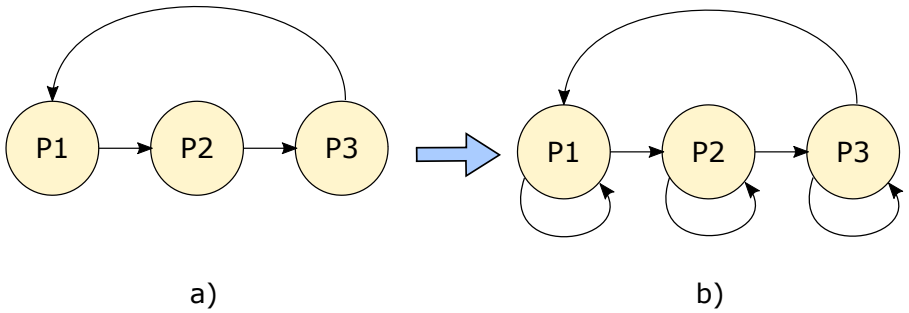


Figure D.3: (a) PPN of example code in Listing D.4 and (b) stream multiplexed version of that same code

To increase the utilization, other datasets of v can be added. The throughput of a single execution of the PPN is not improved, but by considering multiple executions the throughput has increased.

In the example of Listing D.5, each function statement is enclosed by a loop with bound P. Where P is the streaming factor. Now variable v is expanded to an array where each index contains an independent stream of v with P streams.

D.4 Summary

Presented are three loop transformation techniques that may be applied to some process of the network. Determining when to apply these transformations is a crucial part as loop interchange and loop skewing is only necessary when accessing the data in the loop is not optimal due to data dependences. While the transformation known as stream multiplexing focuses on the execution of a network and helps to increase the throughput when considering multiple executions.