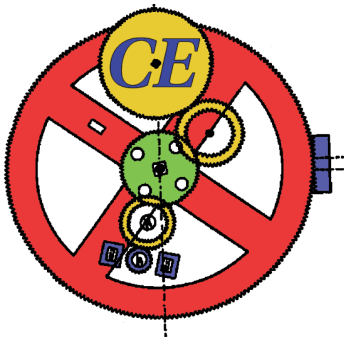


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2018

MSc THESIS

Polyhedral Compilation for Data Flow Computers

Kees van der Bok

CE-MS-2018-14

Abstract

The increasing transistor density of Integrated Circuits (ICs) ever since their introduction, has scaled the computational performance of microprocessors. As a consequence of the gain in transistor density, the power dissipation density has also increased to a degree that has become a limiting factor in further performance scaling. The prevalent control flow computing paradigm is, especially in the context of High Performance Computing (HPC), faced with this power crisis. Data flow computing can achieve a better computations per unit power ratio and might therefore be the solution to achieving exa-scale computing, enabling progress in science and technology. This work contributes in investigating methods for increasing the abstraction level of data flow programming by using a Polyhedral Process Network (PPN) as abstraction to facilitate automated analysis and transformation of data flow applications. A Polyhedral Process Network (PPN) is a process network based on Polyhedral Compilation (PC) a mathematical framework for code analysis and optimisations. A tool has been implemented to translate a PPN to a data flow implementation. The Maxeler Data Flow Engine (DFE) technology, a data flow hardware platform supported by a tool-flow, is used as target technology. For a number of specific challenges involved in the transformation from PPN to DFE-implementation a method has been implemented. This has shown that a PPN as well as the capabilities of PC are useful concepts that are applicable to further increase of the abstraction-level of data flow computing. However the transformation from PPN to DFE-implementation is complex. Therefore this work concludes that the development of a data flow tailored Polyhedral Compilation (PC) abstraction is the most promising future direction.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Polyhedral Compilation for Data Flow Computers

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Kees van der Bok
born in Dirksland, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Polyhedral Compilation for Data Flow Computers

by Kees van der Bok

Abstract

The increasing transistor density of Integrated Circuits (ICs) ever since their introduction, has scaled the computational performance of microprocessors. As a consequence of the gain in transistor density, the power dissipation density has also increased to a degree that has become a limiting factor in further performance scaling. The prevalent control flow computing paradigm is, especially in the context of High Performance Computing (HPC), faced with this power crisis. Data flow computing can achieve a better computations per unit power ratio and might therefore be the solution to achieving exa-scale computing, enabling progress in science and technology. This work contributes in investigating methods for increasing the abstraction level of data flow programming by using a Polyhedral Process Network (PPN) as abstraction to facilitate automated analysis and transformation of data flow applications. A Polyhedral Process Network (PPN) is a process network based on Polyhedral Compilation (PC) a mathematical framework for code analysis and optimisations. A tool has been implemented to translate a PPN to a data flow implementation. The Maxeler Data Flow Engine (DFE) technology, a data flow hardware platform supported by a tool-flow, is used as target technology. For a number of specific challenges involved in the transformation from PPN to DFE-implementation a method has been implemented. This has shown that a PPN as well as the capabilities of PC are useful concepts that are applicable to further increase of the abstraction-level of data flow computing. However the transformation from PPN to DFE-implementation is complex. Therefore this work concludes that the development of a data flow tailored Polyhedral Compilation (PC) abstraction is the most promising future direction.

Laboratory : Computer Engineering
Codenumber : CE-MS-2018-14

Committee Members

Advisor : Dr. Ir. A.J. van Genderen
Department of Computer Engineering, TU Delft

Advisor : Prof. Dr. Ir. G. N. Gaydadjiev
Maxeler IoT Labs Delft
Department of Parallel and Distributed Systems, TU Delft
Department of Computing, Imperial College

Chairperson : Dr. Ir. J.S.S.M. Wong
Department of Computer Engineering, TU Delft

Member : Dr. Ir. C. Strydis
Erasmus MC

Dedicated to my family and friends

Contents

List of Figures **ix**

List of Acronyms **xii**

Acknowledgements **xiii**

1 Introduction **1**
1.1 Observations . . . 1
1.1.1 Challenges of Reconfigurable High Performance Computing . . . 1
1.1.2 Polyhedral Compilation and Polyhedral Process Networks . . . 2
1.2 Thesis Formulation . . . 2
1.2.1 Research Questions . . . 3
1.3 Thesis Organisation . . . 3

2 Polyhedral Compilation and Polyhedral Process Networks **5**
2.1 Polyhedral Compilation . . . 5
2.1.1 The Fundamental Concepts of Polyhedral Compilation . . . 5
2.2 The Polyhedral Model . . . 7
2.2.1 Instance Set . . . 8
2.2.2 Access Relation . . . 10
2.2.3 Dependence Relation . . . 11
2.2.4 Schedule . . . 14
2.3 Polyhedral Scheduling and Transformations . . . 16
2.4 The Integer Set Library . . . 18
2.4.1 Named Integer Tuples . . . 18
2.4.2 Named Integer Tuple Templates . . . 19
2.4.3 Structured Named Integer Tuple Templates . . . 19
2.4.4 Spaces . . . 20
2.4.5 Presburger Formulas . . . 20
2.4.6 Presburger Sets and Presburger Relations . . . 20
2.5 Polyhedral Process Networks . . . 21
2.5.1 ISA library . . . 26
2.5.2 Polyhedral Process Networks Implementation . . . 26

3 Data flow computing **29**
3.1 Data Flow Computing vs Control Flow Computing . . . 29
3.2 Computation on FPGA . . . 30
3.3 Maxeler Data Flow Computing . . . 31
3.3.1 Maxeler Data Flow Engine . . . 31

3.3.2 Maxeler Tool flow . . . 32

4 State of the Art **37**
4.1 Field Programmable Gate Array (FPGA) based Computing Platforms and Tools . . . 37
4.1.1 OpenCL . . . 37
4.1.2 The Intel Acceleration Stack for Xeon CPU with FPGAs . . . 38
4.1.3 Xilinx Vivado High Level Synthesis (HLS) . . . 38
4.1.4 Xilinx SDAccel . . . 38
4.1.5 Chisel . . . 39
4.1.6 Liquid Metal . . . 39
4.1.7 BlueSpec Verilog . . . 39
4.2 Frameworks using Polyhedral Compilation . . . 39
4.2.1 CARP Project . . . 39
4.2.2 PPCG . . . 40
4.2.3 Polly . . . 40
4.2.4 Pluto . . . 40

5 Design and Architecture **41**
5.1 Architecture . . . 41
5.2 A Manual MaxJ Implementation . . . 42
5.3 Generating MaxJ from a Polyhedral Dependence Graph (PDG) . . . 47
5.4 The DfeGraph . . . 47
5.5 Transformations, Parallelism and C-slow retiming . . . 51

6 Implementation **55**
6.1 *DfeGraph* Generation Algorithm . . . 55
6.1.1 Step 1: Counter-chain Generation . . . 55
6.1.2 Data Path Generation . . . 59
6.2 Implementation Details . . . 60
6.2.1 Inputs and Outputs . . . 60
6.2.2 Reuse Buffer Detection and Generation . . . 60
6.2.3 Feedback Path Detection and Generation . . . 66
6.2.4 Multiplexer Detection and Generation . . . 66
6.2.5 Constants . . . 68
6.2.6 Computation . . . 69

7 Conclusions **71**
7.1 Addressing the Research Questions . . . 71
7.2 Future Work . . . 73

Bibliography **76**

A Dependence Analysis in the Integer Set Library **77**
A.1 Access Relations . . . 77
A.2 Dependence Relations . . . 78

A.3 Filling our Toolbox 79
A.3.1 Composition 79
A.3.2 The Inverse of a Presburger Relation 79
A.3.3 The intersection of Presbuger Relations 79
A.3.4 Lexicographical Order 80
A.3.5 The Order Relation 80
A.4 Applying the Tools 81

B Feautrier's Scheduler **83**

C Classic Loop Transformations **87**

List of Figures

2.1 Convexity of Polyhedra 6
2.2 A graphical representation of the instance set of statement T in Listing 2.1 for N = 6, M = 8 9
2.3 The code (left) and the dependence polyhedron (right) modelling the Read-after-write dependence between the statements S2 and S3 the two involved accesses are high-lighted in the code. The code is repeated from Listing 2.2 13
2.4 The schedule-tree corresponding to the code in Listing 2.2 15
2.5 Kernel and Corresponding PPN 23
2.6 A PDG graph 25
2.7 The ISA library tool-flow 26
2.8 Class diagram of the PDG data-structure 28

5.1 Tool Flow 41
5.2 The PPN Corresponding to the Matrix Multiplication Kernel 43
5.3 Part of the PDG related to the data reuse on array A 44
5.4 An example of the intermediate representation (i.e. DfeGraph) 49
5.5 The DfeGraph data path generation from a PDG 50
5.6 C-Slow retiming for C=4 51

6.1 The PDG and the corresponding Abstract Syntax Tree (AST) 57
6.2 The DfeGraph data path generation from a PDG earlier shown as Figure 5.5 61
6.3 Part of the PDG shown in Figure 6.1a 62
6.4 A detailed view of the partial PDG shown in 6.3 62
6.5 A detailed view of the dependences in the partial PDG 64
6.6 The partial PDG relavant for multiplexer detection 67

List of Acronyms

ADG Approximated Dependence Graph

API Application Programmer Interface

ASIC Application Specific Integrated Circuit

AST Abstract Syntax Tree

BSV Bluespec SystemVerilog

CARP Correct and Efficient Accelerator Programming

CPU Central Processing Unit

DFE Data Flow Engine

DMA Direct Memory Access

DSL Domain Specific Language

FPGA Field Programmable Gate Array

GPP General Purpose Processor

GPU Graphics Processing Unit

HDL Hardware Description Language

HLS High Level Synthesis

HPC High Performance Computing

IC Integrated Circuit

IP-CORE Intellectual Property Core

IR Intermediate Representation

ISA Integer Set Analysis

ISL Integer Set Library

KPN Khan Process Network

LLVM-IR LLVM Intermediate Representation

OpenCL Open Computing Language

PC Polyhedral Compilation

PC Polyhedral Compilation

PCIe Peripheral Component Interconnect Express

PDG Polyhedral Dependence Graph

PeNCIL Platform Neutral Compute Intermediate Language

PPCG Polyhedral Parallel Code Generator

PPN Polyhedral Process Network

RTL Register Transfer Level

SCoP Static Control Part

SD-RAM Synchronous Dynamic Random Access Memory

SDK Software Development Kit

SLiC Simple Live CPU

SRAM Static Random Access Memory

YAML YAML Ain't Markup Language[1]

[1] A human readable data serialisation standard for programming languages

Acknowledgements

The advice, help and support I received during this endeavour have been indispensable and I am deeply grateful to all that have provided it. I would like to express my gratitude to my advisor Arjan van Genderen, for the discussions regarding my work, advice and the constructive comments on earlier iterations of this document. It has been of great value to me. My sincere appreciation I would also like to express to Georgi Gaydadjiev my daily advisor, for providing me with advice and constructive comments, both on my work and earlier versions of this document. Furthermore for providing me with a inspiring environment at Maxeler IoT Labs Delft during my thesis project. Also for inviting me to Maxeler Technologies London, were I stayed for a few days. It was a valuable experience. During my stay I have been able to meet and speak to many people, who I am thankful for making me feel welcome. Specifically I want to thank Nils Voss who has helped me to get started with the DFE technology in London and for supporting me via email. I thank him for effectively clarifying things I misunderstood and also providing me with adequate answers to questions regarding the technology. Last but not least I would like to thank my friends and family, especially my parents as well as my brother and sister for there support.

Kees van der Bok
Delft, The Netherlands
July 2, 2018

1 Introduction

This chapter will introduce the thesis as well as the research questions that it poses. First the observations motivating this work are laid out. These observations will also provide a concise description of concepts, research and technology utilised in this work. The observations are meant to pave the way for the thesis formulation and the research questions that logically follow from it. Additionally this chapter sketches the structure of the entire document.

1.1 Observations

This section elaborates on the observations that inspired this thesis. This section will only explore the bare minimum necessary to introduce the thesis and its research questions. Chapter 2 and 3 will describe these ideas and concept in much more detail.

1.1.1 Challenges of Reconfigurable High Performance Computing

Using FPGAs for HPC has been researched for many years. The primary motivation for using FPGAs is in essence their ability to implement custom hardware architectures. Where a General Purpose Processor (GPP) or Graphics Processing Unit (GPU) has a fixed architecture, an FPGA allows the implementation of an architecture specialised for a specific problem. The latter has obviously many benefits, but despite these, there are various challenges that obstruct mass employment of reconfigurable computing. One major problem is the degree of complexity involved in implementing an FPGA design.

Improving programability of FPGAs, has been pursued by many academic and commercial initiatives. One general approach is HLS, a method for translating a general purpose programming language, often C, to Hardware Description Language (HDL). It is, however, challenging achieving good performance using HLS[1].

A concept that increases programmability of FPGAs, without sacrificing heavily on performance is the Data Flow Engine (DFE) technology developed by Maxeler Technologies[2]. This technology is comprised of a hardware platform, and programming tools. The tools provide a Java based programming method for the hardware platform. This doesn't mean that the tools translate general Java code to a DFE implementation. Rather, it provides a library to construct a DFE implementation. This approach lifts the abstraction and therefore increases the productivity. Additionally it opens FPGA based HPC to developers not familiar with HDLs.

From the former observations the following conclusions are drawn: There is a gap between imperative code and an Register Transfer Level (RTL) implementation that is not easily bridged by automated mechanisms. Some human intelligence and intuition is typically required to produce an efficient RTL implementation. The Maxeler DFE technology allows designers to focus on this human specific input by providing a fully integrated platform that raises the abstraction level of application developement significantly. This has proven to be a successful approach in addressing the programability in the context of FPGA based HPC[1], see also Section 4.1.

1.1.2 Polyhedral Compilation and Polyhedral Process Networks

Polyhedral Compilation (PC) and Polyhedral Process Networks (PPNs)[3] might at first seem completely unrelated to what we have discussed so far. However, in oder to motivate its relevance to this thesis it is helpful to first explore these two related concepts on their own.

PC is a mathematical framework for performing both code analysis and code transformations. Imperative code is often used as input source in many PC approaches, however PC is not necessarily restricted to the imperative paradigm. In PC the notion of *statement instances* is central. A statement instance is the dynamic execution of a statement. The significance of this is most prevalent when considering a statement within a loop construct. This statement is, typically, executed multiple times. Polyhedral Compilation (PC) abstracts this concept of iterative execution of a statement and gives it a geometric interpretation. Additionally it allows for code analysis and code transformation on a statement instance granularity.

The concept of PPNs uses PC concepts to model the input code as a data dependence graph. The observation made is that a data dependence graph seems to be a suitable abstraction from which a DFE implementation can be conveniently generated.

1.2 Thesis Formulation

Based on the observations in the previous section the research proposal of this thesis is to investigate the applicability of PC and in particular PPNs in the context of data flow computing. This is done by implementing a tool that translates a PPN to a DFE implementation. In the first place this will reveal whether a PPN is a suitable abstraction of a DFE implementation and secondly, whether this abstraction allows for automatic optimisations, of the generated DFE implementation, by PC techniques.

1.2.1 Research Questions

Based on the thesis, formulated above the following research questions are proposed:

- Is it possible to generate a Data Flow Engine (DFE) implementation from a Polyhedral Process Network (PPN)?
- Can such a mapping be efficient, in terms of the theoretical achievable performance?
- Can the concepts from Polyhedral Compilation (PC) be applied to perform transformations on the Polyhedral Process Network (PPN) abstraction that will improve the efficiency of the generated Data Flow Engine (DFE) implementation?
- What are performance improving transformations when a Data Flow Engine (DFE) implementation is considered as our target?

These research questions will be addressed in the conclusions in Chapter 7.

1.3 Thesis Organisation

The remainder of this thesis report is organised as follows:

Chapter 2 Polyhedral Compilation and Polyhedral Process Networks explores and explains Polyhedral Compilation (PC). The chapter will also cover the general idea and a specific implementation of the Polyhedral Process Network (PPN) concept.

Chapter 3 Data flow Computing covers the concept of data flow computing and compares it to control flow computing. The second part of the chapter will discuss and explain the Maxeler DFE approach to data flow computing.

Chapter 4 State of the Art discusses related research and projects.

Chapter 5 Design and Architecture documents the design process and design choices that have led to the architecture of the developed tool.

Chapter 6 Implementation describes the implementation of the tool in detail. In this chapter specific implementation details are elaborated upon.

Chapter 7 Conclusions discusses the conclusions using the research questions as structure for this discussion.

Polyhedral Compilation and Polyhedral Process Networks

2

This chapter describes and explores the concepts and approaches to Polyhedral Compilation (PC). This chapter functions as general introduction to the subject and covers some of the mathematics involved. Additionally this chapter discusses the concept of a Polyhedral Process Network (PPN) and the specific PPN implementation used in this work, the Polyhedral Dependence Graph (PDG).

2.1 Polyhedral Compilation

We have briefly introduced the term PC in Chapter 1, presently we will discuss PC in detail. Polyhedral Compilation (PC) is a term that refers to a field of research that uses mathematical concepts to model programming code. This model allows for dependence analysis and code transformations. Historically polyhedra were used to abstract various aspects of the input code. Recently, another similar abstraction using sets and relations has been introduced[4, 5]. The later is also regarded as an approach to PC, while technically it does not use polyhedra.

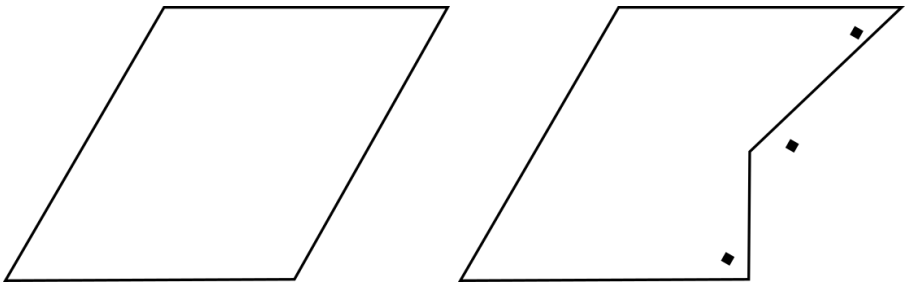
Although the implementation of the abstraction differ among approaches within the Polyhedral Compilation (PC) field there are some fundamental ideas and concepts that they all share. This chapter intends to reveal these ideas and concepts while also discussing their implementation in the specific approaches.

The appeal of PC is that it abstracts the dynamic executions of statement instances. PC allows to reason about different dynamic executions of the same statement, which occur for example in loop constructs. This in contrast to an Abstract Syntax Tree (AST) which is an abstraction of the programs syntax and does not explicitly model the idea of statement instances. The benefit of PC is that it abstracts the actual executed statements and therefore enables both code analysis and code transformation.

2.1.1 The Fundamental Concepts of Polyhedral Compilation

There are some fundamental mathematical concepts that underpin PC. These concepts are key to PC in why this idea works and why there are restrictions to for example the type of allowed transformations.

One of these concepts is that of polyhedra. This concept which has been mentioned multiple times we will now properly address. Polyhedra is the plural of polyhedron. A polyhedron is a space bounded by a finite number of hyperplanes. The hyperplanes



(a) A Convex Polyhedron (b) A Non Convex Polyhedron

Figure 2.1: Convexity of Polyhedra

actually separate the original space in a number of half spaces, the intersection of these half spaces form the polyhedron. In a two-dimensional space hyperplanes are lines and a finite number of lines can be used to bound a the 2d area. The formed area can be infinite if the lines form a cone or bounded if it is restricted in all directions. Such a bounded polyhedron is called a polytope. Later we show how such polytope can be used to abstract for example an *instance set*.

The polyhedra used in PC approaches are in fact convex polyhedra. This brings us to the mathematical concept of convexity in the context of polyhedra, for which there is an informal but rather intuitive definition. This definition states that for any two points that are both members of a convex polyhedron, each point on the straight line between these two points is also a member of that polyhedron. This definition is a also applicable to sets. Polyhedra described using a system of linear inequalities (i.e. an intersection of half spaces) will always be convex. A convex polyhedron is shown in Figure 2.1a, while Figure 2.1b shows a non-convex polyhedron. It is graphically illustrated that one can choose two points in the polyhedron for which there are points on the straight line between them that are not in the polyhedron.

Another mathematical term that one often encounters in PC literature is the term affine. It is for example found in affine transformations. These affine transformations are vector valued functions of the format shown in Equation 2.1. An affine transformation is basically a linear transformation plus a translation[6]. Affine transformations are the most general type of transformations that preserve convexity. This property ensures that convex polyhedra transformed by affine transformations will remain convex polyhedra. Therefore the abstraction remains within the conditions enforced by the PC and can be used for further analysis and code generation. For a more detailed explanation regarding these mathematical concepts see[7, Section 2.4.1].

$$\begin{aligned} f(x_1, \ldots, x_n) &= A_1 x_1 + \ldots + A_n x_n + b \\ f(\bar{x}) &= A\bar{x} + b \end{aligned} \tag{2.1}$$

2.2 The Polyhedral Model

Although there exist multiple approaches to polyhedral compilation, the focus here will be on the approach described in [4, Chapter 5]. This is the approach implemented by the libraries used in this work. This approach is the one using sets and relations instead of polyhedra, and will be referred to as the Integer Set Library (ISL) approach. The classical approach to PC, the one actually using polyhedra, is also discussed and used in examples.

A polyhedral model is an abstraction of a piece of code, and is particularly useful when the code involves iterations of one or more statements. In case the input code is written in an imperative language (e.g., C) statements embedded in loop constructs are an example of iterative statements. Most models put some restrictions on the input format of the code especially on the loop statements. The format that all models, of interest, support is that of the so called Static Control Part (SCoP). A SCoP is an optionally nested loop with static control (i.e. bound and strides of the loops are known at compile time either exactly or as parameters). The code shown in Listing 2.1 is an example of a SCoP.

```
R: some statement
for(i=0; i<N; i++){
  S: some statement
  for(j=0; j<M; j++){
    T: some statement
  }
}
```

Listing 2.1: A SCoP

A polyhedral model is typically composed out of the following components:

Instance Set Is the set of dynamically executed instances. For example, if a statement is encapsulated in a for loop, the instance set abstracts each individual execution of that statement. The instance set can be abstracted as a bounded polyhedron in n-dimensional space. Each integer point in this space represents a specific iteration of the statement. Alternatively the Instance Set could be abstracted as an integer set. Each separate statement has an Instance Set associated with it, instance sets of two statements can overlap or even be equal.

Dependence Relation The dependence relation is a relation that gives information about the dependences between statement instances. Dependences can exist between instances of two distinct statements or instances of one and the same statement. Usually multiple types of dependencies are considered (e.g., *read-after-write*, *write-after-read*) et cetera. Typically the relations reflect that one statement instance should be executed before the other. These relations can be expressed as a polyhedron of pairs of dependant iterations of two instance sets. Alternatively it

can be expressed as an integer relation.

Schedule A schedule is a relation on the instance set that expresses the order in which the statement instances should be executed. A schedule defines a strict partial order and is an irreflexive and transitive relation. Transforming a nested for loop comes down to changing the schedule. However, an alternative schedule needs to respect the dependence relations. Schedules that respect the dependences are called legal schedules. A schedule is generally expressed as a function that assigns a timestamp to each element of the instance set. This timestamp can simply be one-dimensional, but often a multi-dimensional timestamp is used. In a multi-dimensional schedule a lexicographical[1] execution order is assumed. A recent alternative for specifying schedules are schedule-trees discussed later in more detail.

Access Relation The access relation maps elements from the instance set to members of a data set. So it tells which element of the dataset is accessed by a specific access in a statement instance. The datasets are considered to be arrays. Scalars are supported by regarding them as 0-dimensional arrays.

One of the major challenges of PC is finding an alternative schedule. The motivation for such alternative schedule is to perform performance improving code transformations, resulting in an alternative execution order of the statement instances. For example an alternative schedule could optimise the reuse distance (i.e. minimise the number of iterations between two uses of the same data element). Optimisations like this could yield a performance gain because they may improve the utilisation of a cache or scratch path memory. Alternatively, a new schedule can maximise the number of statement instances that can be executed in parallel. Scheduling algorithms typically try to generate a schedule that improves the code for multiple objectives. Besides data reuse distance and parallelism other objectives can be of interest depending for example on the envisioned target hardware.

2.2.1 Instance Set

Consider a simple nested for loop structure as shown in Listing 2.1. It is a rather obvious observation that iterative executions of statements S and T can each be considered a two-dimensional set or polytope in $\mathbb{N}^2$. See Figure 2.2 for a graphical representation of the instance sets (i.e. polytopes) of both statement S and T in Listing 2.1. The loop bounds of the for loop, containing statement T, constrain the values both i and j can take. The loop bounds, an thus the instance set of T, can be be expressed as a system of linear inequalities see Equation 2.2.

$$i \geq 0, i < N, j \geq 0, j < M, \tag{2.2}$$

[1]The lexicographical order is a trivial order on multi-dimensional numerical objects, for a formal definition see Section A.3.4

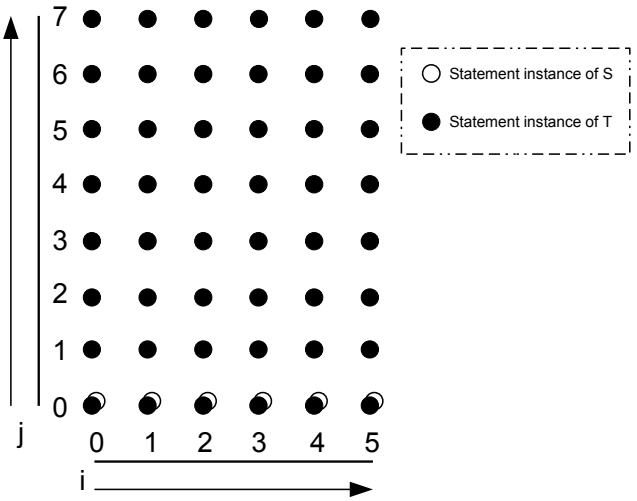


Figure 2.2: A graphical representation of the instance set of statement T in Listing 2.1 for N = 6, M = 8

If we reorganise the inequalities we can form an elegant and uniformly applicable notation using concepts from linear algebra. The latter is actually the accepted form to describe polyhedra, which in this case is actually a polytope because it is bounded.

The system shown in Equation 2.3 has two variables i and j these two variables in some sense form a vector in $\mathbb{N}^2$. Every possible vector that satisfies 2.3 forms an instance of statement T. The area, or more precisely the number of points, formed by the vector is called the instance set or iteration space. The vector is often referred to as the iteration vector. The instance set of statement S is represented by Equation 2.4.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 1 \\ 0 & -1 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ M \\ 1 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{2.3}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} i \\ N \\ 1 \end{pmatrix} \geq \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{2.4}$$

An alternative to this matrix based description is expressing the instance set as shown in Equation 2.5. It does not require a leap of the imagination to see that the two representations actual convey the same information. The set notation has to conform to some constraints, which we will currently ignore, the details are addressed in Section 2.4.5.

$$[N, M] \rightarrow \{S[i] : 0 \leq i < N; T[i, j] : 0 \leq i < N; 0 \leq j < M\} \tag{2.5}$$

2.2.2 Access Relation

The access relations are used to model scalar and array references, in the input code. A statement usually contains multiple array and/or scalar references and is therefore associated with multiple access relations. There is an access relation for each scalar or array reference.

The access relation, also referred to as access function maps a statement instance to a element of an array. Usually this relation maps iteration coordinates (specifying a specific statement instance) to array indices. Scalars are modelled as zero-dimensional arrays. In order to allow for dependence analysis a distinction is made between read and write accesses.

```
int A[N];
int B[N][M];
int c;

S1: c = 2;

for (int i=0; i<N; i++){
  S2: A[i] = 0;

  for (int j=1; j<M; j++){
    S3: A[i] = A[i] + B[i][j] * c;
  }
}
```

Listing 2.2: Simple Kernel

In order to demonstrate how access relations model the data access in the input code we take into consideration the code shown in Listing 2.2. We aim to show as many facets of the access relations as possible while keeping the example code simple. The assumption in this example is that both arrays A and B have been initialised with data outside the shown code. The variable declarations are added to this code listing for clarification.

The depicted code scales the elements of B with the value of c and accumulates these results for each "column" of B and stores these results in array A. The code could be regarded as a scaled histogram, but purely serves an instructional purpose.

In the ISL approach the relations are bundled per type (i.e. read or write) combined in what is formally a union of relations. Each individual relation in this union is separated with a semicolon. Equation 2.6 shows the write access relations of the code in Listing 2.2. For readability each individual relation in this union of relations is printed on a separate line.

$$\begin{aligned}[N, M] \rightarrow \{ & \\ & S1[] \rightarrow c[]; \\ & S3[i, j] \rightarrow A[i] : 0 \leq i < N \wedge 0 < j < M; \\ & S2[i] \rightarrow A[i] : 0 \leq i < N \\ \} &\end{aligned} \tag{2.6}$$

The first relation models the write access of scalar c in statement $S1$. Statement $S1$ is executed only ones (i.e. has only one statement instance). Therefore there are no iterators associated with this statement. This relation also illustrate how scalars are treated as zero-dimensional arrays.

The read access relations can be modelled as a union of relations as follows:

$$\begin{aligned}[N, M] \rightarrow \{ & S3[i, j] \rightarrow B[i, j] : 0 \leq i < N \wedge 0 < j < M; \\ & S3[i, j] \rightarrow A[i] : 0 \leq i < N \wedge 0 < j < M; \\ & S3[i, j] \rightarrow c[] : 0 \leq i < N \wedge 0 < j < M\}\end{aligned} \tag{2.7}$$

Although the notation is rather self-explanatory, it is subject to some technicalities we will later discuss in detail in Section 2.4.5.

2.2.3 Dependence Relation

A dependence relation models the dependence between pairs of statement instance, more particular between accesses in those statement instances. They enforce an order of execution of these two statement instances where the statement instance in the range of the relation can only be executed after the statement instance in the domain of the relation. Such dependence is typically the result of both statement instances accessing the same data element.

The dependences are generally classified based on the type of the two involved accesses (i.e. read or write) causing the dependence. The following three dependence types are usually considered:

Read-after-write The data is written by one statement instantiation and later read by another instantiation.

Write-after-read Data read by one statement instantiation and later written by another.

Write-after-write Data is written by one statement instantiation and thereafter overwritten by another.

One could argue that there is also a *read-after-read* dependence. However, reordering the accesses does not have effect on the final state of the program. A *read-after-read* dependence does therefore not enforce a specific execution order of the involved statement instances. For this reason it is often not considered a dependence.

When code is transformed (i.e. a change of the execution order of statement instances) the dependence relations should be respected in order to preserve the meaning of the original code. If the dependences are not respected the transformed code would not be equal to the original code and most likely produce different results.

In the classical approach to PC a polyhedron is used to model the dependences between a pair of statements. For each pair of statements that have a dependency such a polyhedron is constructed. The general structure of such dependence relation is depicted in Equation 2.8.

$$\left(\begin{array}{cc} F_S & -F_T \\ \hline A_S & 0 \\ 0 & A_T \end{array}\right) \begin{pmatrix} i_S \\ i_T \end{pmatrix} \begin{array}{l} = 0 \\ \hline \geq 0 \end{array} \tag{2.8}$$

In Equation 2.8 F_S and F_T are coefficients of the access functions of statement S and T respectively. F_S and F_T are matrices, while i_S and i_T are vectors. By isolating the part of the system that involves the access functions we gain:

$$\begin{aligned} F_S \cdot i_S - F_T \cdot i_T &= 0 \\ F_S \cdot i_S &= F_T \cdot i_T \end{aligned} \tag{2.9}$$

The statement instance i_S and i_T satisfying the equality are the ones that access the same data element. The sub matrices A_S and A_T specify the statement instances, of the two statements, the dependence applies to.

Considering the code shown in Figure 2.3. There is an *read-after-write* dependence between statement $S2$ and statement $S3$. Statement $S2$ writes to $A[i]$ and after that $A[i]$ is read by $S3$. Equation 2.11 shows the dependence polyhedron modelling this

dependence. Alternatively the dependence can also be expressed as a relation depicted in Equation 2.10.

$$\{S2[i] \rightarrow S3[i' = i, j] : 0 < i < N \wedge 0 < j < M\} \tag{2.10}$$

Each array or scalar reference in a statement is considered an access. In order to identify the distinct accesses we label them per statement using a number. The statements $S2$ and $S3$ in Figure 2.3 have the following accesses:

Accesses of S2: $S2_0 = A[i]$

Accesses of S3: $S3_0 = A[i]$, $S3_1 = A[i]$, and $S3_2 = B[i][j]$, $S3_3 = c$

The two access in the *read-after-write* dependence are $S2_0$ and $S3_1$. Each of these accesses has an access function associated with it. In the example the two access functions are:

$$\begin{aligned} f_{S2,0}(i_{S2}) &= i_{S2} \\ f_{S3,1}(i_{S3}, j_{S3}) &= i_{S3} \end{aligned} \tag{2.12}$$

For all statement instance pairs $< S2(i_{S2}), S3(i_{S3}, j_{S3}) >$ that satisfy the following equation the two statement instance access the same array element.

$$\begin{aligned} f_{S2,0}(i_{S2}) &= f_{S3,1}(i_{S3}, j_{S3}) \\ i_{S2} &= i_{S3} \\ i_{S2} - i_{S3} &= 0 \end{aligned} \tag{2.13}$$

```
int A[N];
int B[N][M];
int c;

S1: c = 2;

for (int i=0; i<N; i++){
  S2: A[i] = 0;

  for (int j=1; j<M; j++){
    S3: A[i] = A[i] + B[i][j] * c;
  }
}
```

$$\left(\begin{array}{cccc} 1 & -1 & 0 & 0 \\ \hline 1 & 0 & 0 & -1 \\ -1 & 0 & 0 & N \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & N \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & M \end{array}\right) \begin{pmatrix} i_{S2} \\ i_{S3} \\ j_{S3} \\ 1 \end{pmatrix} \begin{array}{l} = 0 \\ \hline \geq 0 \end{array} \tag{2.11}$$

Figure 2.3: The code (left) and the dependence polyhedron (right) modelling the Read-after-write dependence between the statements S2 and S3 the two involved accesses are high-lighted in the code. The code is repeated from Listing 2.2

Recalling the dependence polyhedron in Equation 2.11 we see that the last statement in Equation 2.13 corresponds to the equality in Equation 2.11.

2.2.4 Schedule

The purpose of a schedule is to prescribe an execution order to an instance set. The schedule is implemented by a function or relation that maps members of the instance set to a timestamp. This timestamp can simple be a scalar, where the implied execution order is in ascending order of the assigned value. However, in practice a multi-dimensional timestamp is often used. This timestamp is a vector or vector-like object. The distinct dimensions of this multi-dimensional timestamp can be thought of as the seconds, minute and hours we, humans, typically use to notate time. The resolution of consecutive timestamps is a unit-step in the first-dimension of the timestamp. This unit-step models an atomic discrete time-step, but has no specific time in seconds related to it. In a hardware implementation this unit-step often relates to the clock-cycle.

A schedule is by its very nature a function. In the classic approach it is represented as a function while in the ISL approach it is represented as a relation. For efficiency reasons, briefly touched upon in[4], schedule-trees, has been introduced in ISL as an alternative schedule representation. Although these schedule-trees have become the preferred method, using relations to represent schedules is still supported in the ISL.

In the context of this thesis both ISL approaches are relevant. A PDG gives a schedule for each node in the graph in the form of a relation. The ISL implemented scheduler produces a schedule-tree.

The concept of schedule-trees has developed since it has been proposed[8]. A helpful treatise on the subject can be found in[4, Section 5.6]. The latest version of the ISL manual describes the current implementation in ISL.

The concept of a schedule-tree is that it is a structured representation of the schedule. The individual nodes in the graph typically give a partial schedule. The schedule of a specific statement is retrieved by traversing the schedule-tree.

Although the actual implementation is quite detailed, the general concept of a schedule-tree can be understood by considering four different node types.

A Domain node models the instances sets of the statements referenced in the tree. The root-node of a schedule-tree is always a domain node.

A Sequence Node models a compound statement (i.e. a sequence of statements that should be executed in sequential order). It has as children a sequence of nodes that should be executed in the specified order.

A Band Node models a loop-statement. The band node has an associated partial (i.e. one-dimensional) schedule. Mapping the statement instances to one dimension of the timestamp.

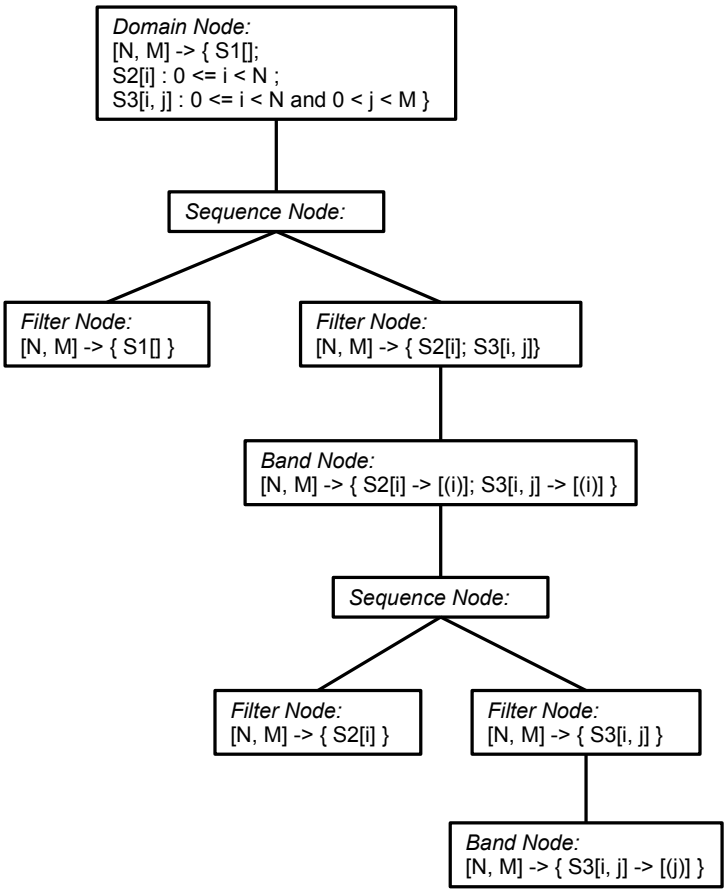


Figure 2.4: The schedule-tree corresponding to the code in Listing 2.2

A Filter Node specifies a sub-set of a particular instance set that applies to the sub-tree of which the filter node is the root. The filter nodes may refer to the whole instance set.

An example of a schedule-tree based on the code in Listing 2.2 is depicted in Figure 2.4. This tree is to be traversed in depth-first order. Children of a node are traversed from left to right.

The members of a band node can be labeled as coincident, if the dependences allow, meaning they can be executed in parallel. The first band node visited, in depth-first traversal of the schedule-tree depicted in Figure 2.4, has two members (i.e. partial schedules):

$$\begin{aligned} S2[i] &\rightarrow [(i)] \\ S3[i,j] &\rightarrow [(i)] \end{aligned} \tag{2.14}$$

Both these partial schedules can be labelled as coincident meaning that this band node can be implemented as a parallel for loop. Such labelling is performed by a scheduler.

The schedule-tree implementation in ISL additionally provides set nodes, which are sequence nodes that don't prescribe an order.

The alternative to a scheduling tree is to provide a schedule, in the form of a relation, for each statement. Considering the code in Listing 2.2, the schedules are modelled as follows:

$$\begin{aligned} S1[] &\rightarrow [0,0,0]; \\ S2[i] &\rightarrow [i,0,1]; \\ S3[i,j] &\rightarrow [i,j,2]; \end{aligned} \tag{2.15}$$

Note that the first dimension only contains constants. This dimension is used to express the sequential order of the statements in the original code text.

2.3 Polyhedral Scheduling and Transformations

There is quite some similarity between the idea of polyhedral transformations and polyhedral schedules. The two terms are often used interchangeable there is however a subtle difference between the two. Where a schedule assigns an timestamp to each instance in an instance set, a transformation specifies an alternative execution order of an instance set.

The classical loop transformations are all expressible as affine transformations. The following examples, based on[9], show how loop transformations can be expressed as affine transformations. Appendix C provides some additional examples.

Loop Interchange

A loop interchange transformation switches the inner and outer loop around.

Original Code:

```
for (int i=0; i<N; i++)
  for (int j=0; j<M; j++)
    S(i,j);
```

Instance Set: $[N] \rightarrow \{S[i,j] : 0 \leq i < N \wedge 0 \leq j < M\}$
Original Schedule: $\{S[i,j] \rightarrow [i,j]\}$
Transformation: $\{[i,j] \rightarrow [j,i]\}$

Transformed Code:

```
for (int j=0; j<M; j++)
  for (int i=0; j<N; i++)
    S(i,j);
```

Loop Tiling

Loop tiling splits the original program in smaller sub problems. In this example the original instance set is split up in tiles of size 4 by 4. Note that these tiles do not have to be square, they even do not have to be rectangular (i.e. the can even be skewed).

A tiling transformation doubles the dimensionality of the instance set. In the example the original instance set is transformed from 2-dimensional to 4-dimensional. The two outer most loops (iterators Ti and Tj) iterate over the tiles, while the two inner most loops provide the iteration order local to the tile. Loop tiling typically requires some transformation of the iterators used by the statement(s) in the transformed loop nest.

Original Code:

```
for (int i=0; i<N; i++)
  for (int j=0; j<N; j++)
    S(i,j);
```

Instance Set: $[N] \rightarrow \{S[i,j] : 0 \leq i < N \wedge 0 \leq j < N\}$
Original Schedule: $\{S[i,j] \rightarrow [i,j]\}$
Transformation: $\{[i] \rightarrow [\lfloor i/4 \rfloor, \lfloor j/4 \rfloor, i \bmod 4, j \bmod 4]\}$

Transformed Code:

```
for (int Ti=0; Ti<N/4; Ti++)
  for (int Tj=0; Tj<N/4; Tj++)
    for (int ti=0; ti<4; ti++)
      for (int tj=0; tj<4; tj++)
        S(4*Ti+ti, 4*Tj+tj);
```

Scheduling

In Section 2.2.4 we have discussed the idea of a polyhedral schedule in this section we will explore how such schedule (i.e. its defining parameters) can be found. In the previous example we have seen how the classical loop transformations are performed within the context of PC. However, the real challenge is to figure out which transformations or combination of transformations are beneficial.

In order to do the latter, scheduling algorithms exist that automatically construct a transformation or schedule. There are two such scheduling algorithms in the polyhedral community that are of particular interest: because (1) they are the most mature approaches and (2) they are both implemented in the ISL.

The two algorithms are the *Feautrier scheduler*[10, 11] and the *Pluto scheduler*[7]. The Feaurtrier scheduler is a greedy iterative algorithm that looks for a schedule with minimal latency. The algorithm tries to resolve, in each step, as much dependences as possible. Generally the schedule maximises fine-grain parallelism.

The Pluto scheduler tries to find a tiling-schedule. And optimises for both parallelism and data-locality. This strategy has proven to be beneficial when targeting a Graphics Processing Unit (GPU)[12]. In the ISL, the Pluto algorithm is used as the primary scheduling algorithm. However it may occur that the Pluto algorithm can not find a schedule, in which case one iteration of the Feautrier scheduler is performed instead that will hopefully satisfy the dependence or dependences that prevent the Pluto scheduler from finding a schedule. After this step and when no total schedule has been found yet, the ISL switches back to the Pluto scheduler. This latter is the standard scheme of the ISL, it is however also possible to only use either scheduler. A detail description of the ISL scheduling strategy can be found in[12]. The latter publication also includes further details regarding the two mentioned scheduling algorithms .

2.4 The Integer Set Library

The Integer Set Library is a library that allows working with integer sets and integer relations. In principle the library is a more or less general library, however it is designed for and used in the field of PC. Two documents have served as inspiration for this section[4]: A tutorial on the underlying concepts of the ISL approach to Polyhedral Compilation and [5]: The official ISL Manual. This section can both be regarded as an introduction as well as a summary of the two mentioned documents. The focus will be on the aspects that are relevant to this thesis.

Some of the concepts from the ISL have informally been introduced previously, deliberately skipping over some formal details, primarily to not confuse the reader by introducing to much detail at once. In this section we will discuss and explore these details. Appendix A describes, including an example, how dependence analysis is performed in the ISL approach to PC.

2.4.1 Named Integer Tuples

In order to understand a set of named integer tuples let us first define what is meant by a named integer tuple. A named integer tuple simply "is an identifier followed by a comma delimited list of integer values enclosed in square brackets"[4]. So $A[2, 8, 1]$ is a named integer tuple, however [] could also be considered a named integer tuple without an identifier and an empty sequence of integers.

Sets of these named integer tuples can be constructed, as shown in Equation 2.16 by simply listing the members of the set. This latter is called an extensional description.

$$\{A[2, 8, 1]; B[4, 5]; []\} \tag{2.16}$$

Relations can for example be described as follows:

$$\{A[2, 8, 1] \to B[5]; A[2, 8, 1] \to B[6]; B[5] \to B[5]\} \tag{2.17}$$

In this extensional description the members of a set or relation must be explicitly enumerated. For this reason their usefulness, in the context of PC is limited, however it could serve as a more or less gentle introduction to the intentional description that follows in the next section.

2.4.2 Named Integer Tuple Templates

The next concept I would like to introduce are *named integer tuple templates* which are used to describe the members of the set or relation intensional. In the extensional description the members of the set are explicitly listed. This in contrast to an intensional description, which uses variables and constraints on these variable to describe the members of the set.

The difference between a named integer tuple and a named integer tuple template is that the latter uses a list of variables instead of integers. This allows for an intensional description of a set or relation. Let us first consider a set both described extensional as well as intensional.

A possible intensional description of a set could be:

$$\{B[i] : 5 \leq i \leq 6; C[]\} \tag{2.18}$$

An extensional description of the same set would be:

$$\{B[5]; B[6]; C[]\} \tag{2.19}$$

Both the intensional and extensional description of the set are in fact equal.

2.4.3 Structured Named Integer Tuple Templates

A structured named Integer Tuple Template is basically a generalisation of named integer tuple templates, such that the definition is also applicable to relations. An relation has both a domain and range integer tuple. As stated in [4] a structured named integer tuple is either:

- A named integer tuple, (i.e., an identifier n along with $d \geq 0$ integers) i_j for $0 \leq j < d$, written $n[i_0, i_1, ..., i_d]$, or,
- A named pair of structured named integer tuples, (i.e. an identifier n along with two structured named integer tuples i and j written $n[i \to j]$.

An example of a set of structured named integer tuples is:

$$\{B[5]; S[B[6] \to A[2, 8, 1]]; Q[B[5] \to S[B[6] \to A[2, 8, 1]]]\} \tag{2.20}$$

The following is example of a binary relation of structured named integer tuples:

$$\{B[5] \to A[2, 8, 1]; S[B[6] \to A[2, 8, 1]] \to B[5]\} \tag{2.21}$$

2.4.4 Spaces

An important concept in ISL is that of spaces. A space gives the identifier and the number of dimension of a named integer tuple. The space of tuple $A[i]$ is $A/1$ while the space of $B[1, i, j]$ is $B/3$.

2.4.5 Presburger Formulas

The sets and relations used in the ISL approach to PC are officially called Presburger sets and Presburger Relations, because the functions that are used in the intensional description of these sets and relations are Presburger formulas.

A Presburger formula is a first order formula in the Presburger language. The Presburger language also referred to as Presburger arithmetic is a first-order theory on the natural numbers that includes addition but excludes multiplication. One specific property of a first-order languages in general is that it allows existential and universal quantification of variables. For a detailed and technical treatise on the Presburger Language see[4, Section 3.2].

The main practical reason for using Presburger formulas in the ISL is that they are decidable[4] (i.e. it is always possible to decide whether a Presburger Formula is satisfied or not).

The practical consequence of the former is that one can not use a multiplication of variables when constructing a function describing the set or relation membership. This seems at first restrictive, however it is not more restrictive than the restrictions enforced by a classical polyhedral description. Therefore Presburger sets and relations can be used to abstract code, and have the same expressive potential as the classical approach.

2.4.6 Presburger Sets and Presburger Relations

Using the concepts covered so far we can dissect the notation of Presburger sets and Presburger Relations. The set and relation used for this example are derived from a polyhedral model, doing so simultaneously links the theoretical concept to a practical implementation.

The following Presburger set abstracts the instance set of the code shown in Listing 2.3. Formally this is considered to be a union of Presburger sets, since it contains two sets with distinct spaces. A set is typically associated with a list of parameters, referred to as the context.

```
S: prod = 0;
for (int i = 0; i < N; ++i)
   T: prod += A[i] * B[i];
```

Listing 2.3: Inner Product Kernel

$$[N] \rightarrow \{S[]; T[i] : 0 \leq i < N\} \tag{2.22}$$

$[N] \rightarrow$ *Represents the context (i.e. list of parameters) of the set, In this example there is only one parameter (i.e. N).*
$S[]$ *The first set of the union, it has only one member.*
$T[i]$ *A template for members of the second set in the union set.*
$0 \leq i < N$ *The Presburger formula defining membership of the second set.*

Similarly a Presbuger relation can be described as follows:

$$[N] \rightarrow \{S[] \rightarrow prod[]; T[i] \rightarrow prod[] : 0 \leq i < N\} \tag{2.23}$$

2.5 Polyhedral Process Networks

A Polyhedral Process Network (PPN) is as the name suggests a process network that uses Polyhedral Compilation (PC) concepts. The structure of a PPN is a graph. In this graph the nodes or vertices represent the statements or processes, while the edges represent the communication channels between processes. Polyhedral Compilation techniques are used to describe the processes, but also the communication channels (i.e. the edges).

Research into process networks has been done for decades. The famous Khan Process Networks (KPNs) are the most well-know result of this research. Although originally developed for distributed computing KPNs, have found application in many other fields. A KPN or in fact any process network can be considered a model of computation.

In the KPN approach the channels between the processes in the network are modelled as unbound FIFO's. PPNs are a variation of KPNs which use PC techniques to model the process and the communication channels between the processes. In contrast to KPNs, PPNs model the communication channels exactly. Using the PPN methodology it is possible to determine the type and the volume of the communication channels.

Let us consider an instructive example to explore the concept of a PPN. We choose some relevant kernel, say matrix multiplication and work out how it is converted to a PPN. Listing 2.5a shows the C code for this kernel while Figure 2.5b shows its corresponding PPN.

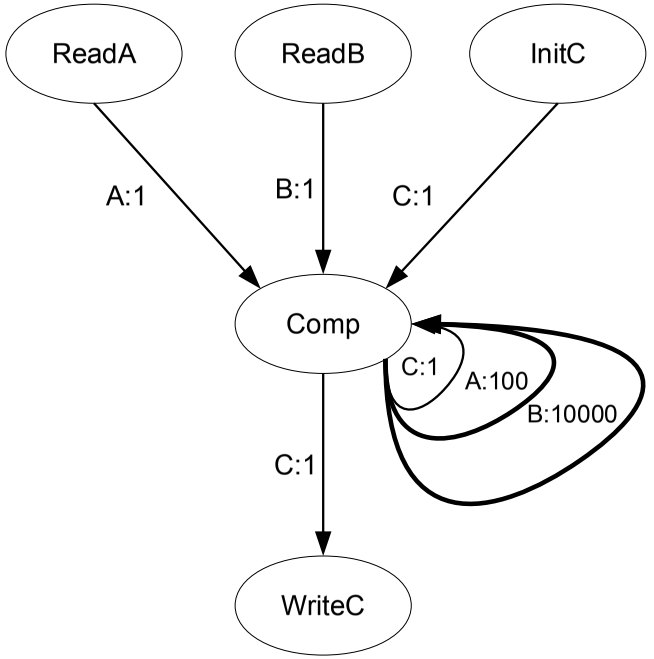
Each node in the PPN is labeled with the statement label from the original code. The edges are labeled with the corresponding array, and the size required when implementing the dependence as communication channel. Beside this, the PPN also specifies the type of a communication channel. The choices are FIFO, shift-register and reordering buffer.

The code in Listing 2.5a has besides the basic kernel some functions added that simulate the data-transfer from a host system to for example a Maxeler Data Flow Engine (DFE). This approach is also taken by[3] and is instructive and useful for this thesis, because

it allows the data transfers between host and DFE to be modelled. In a more mature, implementation of the ideas in this thesis, the necessity of these data transfer functions to be explicitly modelled in the input code could be dropped.

```
int ReadMatrix();
void WriteMatrix(int output);

int main(void){

  int A[100][100];
  int B[100][100];
  int C[100][100];

  for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
       ReadA: A[i][j] = ReadMatrix();
       ReadB: B[j][i] = ReadMatrix();
       InitC: C[i][j] = 0;
    }
  }

  for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
      for (int k = 0; k < 100; k++) {
         Comp: C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
    }
  }

  for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
       WriteC: WriteMatrix(C[i][j]);
    }
  }

  return 0;
}
```

(a) Matrix Multiplication Kernel



(b) The PPN Corresponding to the Matrix Multiplication Kernel

Figure 2.5: Kernel and Corresponding PPN

Considering that the data of matrix B is transferred in transposed order, this allows the associated data channel to be constructed as a shift-register. An automatic reordering of input data could be done using PC concepts. The dimensions of the arrays and loop bounds in this example are hard-coded. This allows for the communication channels to be specified exactly, however the PPN implementation used in this work allows the dimensions and loop bounds to be specified as parameters as well.

A dependence can be implemented as a FIFO if the order in which elements are written is equal to the order in which they are read. A shift-register is a special case of the FIFO type. In addition to the order requirement the distance in the iteration space between a write and its corresponding read must be constant. If none of the mentioned conditions hold the dependence can only be implemented as a reordering-buffer (i.e. a random access memory). In figure 2.5b the thick edges signify that these edges can be implemented as shift-registers. All other edges are FIFOs of size 1, which are trivially implemented as registers.

Figure 2.6 shows a more detailed representation of a PDG. The focus in this diagram is on the PC concepts used in the PDG. For each node the statements are shown in bold. Above each statements its schedule is printed in a box. This depiction clearly shows that a dependence, represented by an arrow, is actually between accesses. The dependence relation of each dependence is annotated with its associated dependence relation (in the dashed box). The instance sets of the statements have been omitted from this representation to maintain readability. These instance sets are represented as Presburger sets and shown next.

The instance sets for the nodes of the PDG as shown in Figure 2.6:

ReadA: $\{ReadA[i,j] : 0 \leq i \leq 99 \wedge 0 \leq j \leq 99\}$

ReadB: $\{ReadB[i,j] : 0 \leq i \leq 99 \wedge 0 \leq j \leq 99\}$

InitC: $\{InitC[i,j] : 0 \leq i \leq 99 \wedge 0 \leq j \leq 99\}$

Comp: $\{Comp[i,j,k] : 0 \leq i \leq 99 \wedge 0 \leq j \leq 99 \wedge 0 \leq k \leq 99\}$

WriteC $:\{WriteC[i,j] : 0 \leq i \leq 99 \wedge 0 \leq j \leq 99\}$

In the top-right of Figure 2.6 a node with a incoming edge (i.e. dependence) is dissected. Most of it is considered self-explanatory, however some details might need some further clarification. Like for example the distinction between read and write access. Furthermore the right-hand-side of the statement is implemented as an AST. The operators (*, +) are considered function calls, width arguments either accesses or other function calls.

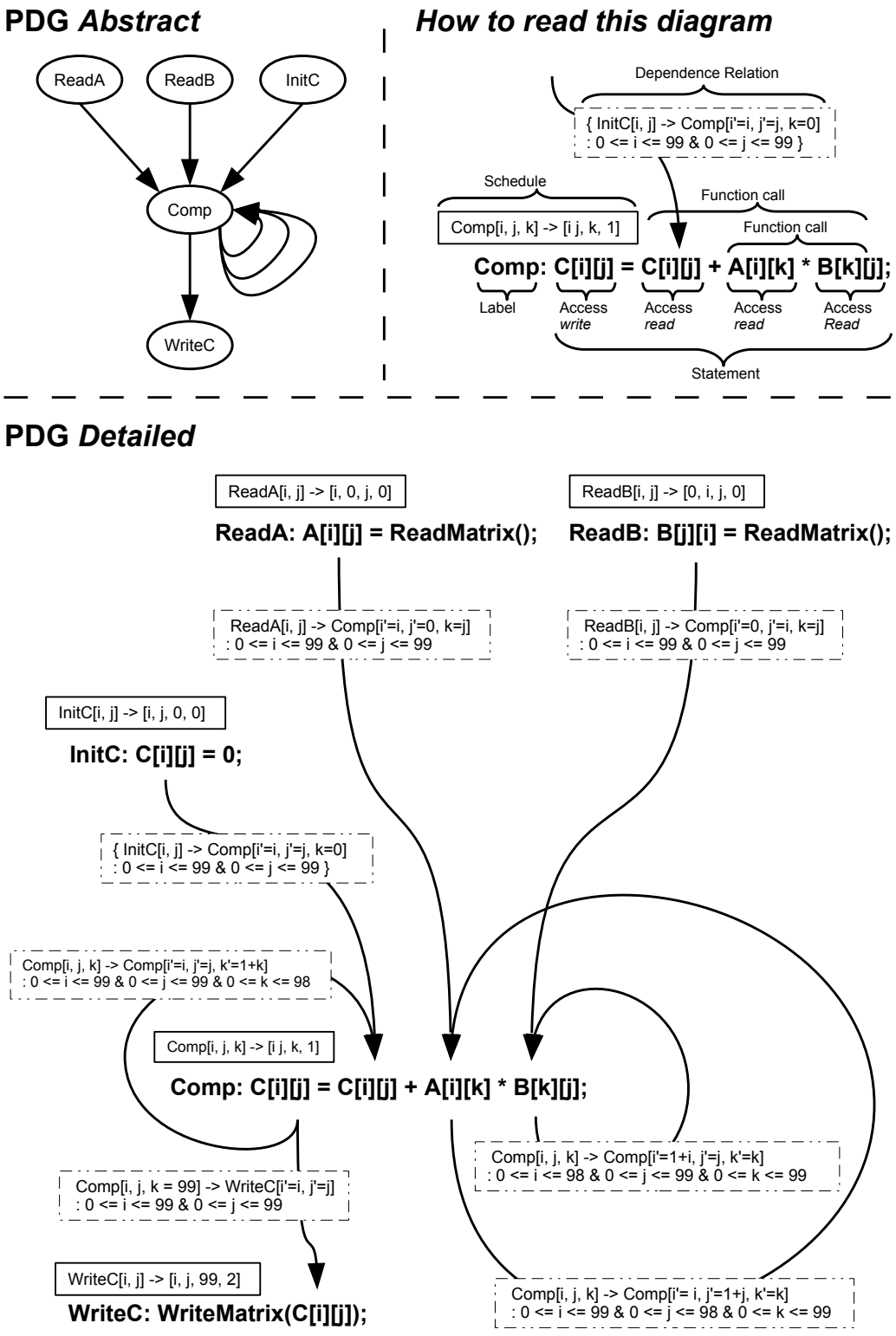


Figure 2.6: A PDG graph

2.5.1 ISA library

The tools for generating PPNs are implemented in the Integer Set Analysis (ISA) library. Figure 2.7 shows the tool-flow for generating a PPN from input C code. In this library the PPN implementation is called a PDG. Such PDG can be constructed from a C file using the *c2pdg* command-line tool. This tool will produce a YAML Ain't Markup Language (YAML)[2] file (i.e. a serialisation of an instantiation of the PDG data-structure). At this point the PDG only contains nodes, the dependences (i.e. edges) can be added to it by passing the PDG to the *pn* tool. This tool produces again a PDG including the dependences. This last PDG is an actual PPN implementation.

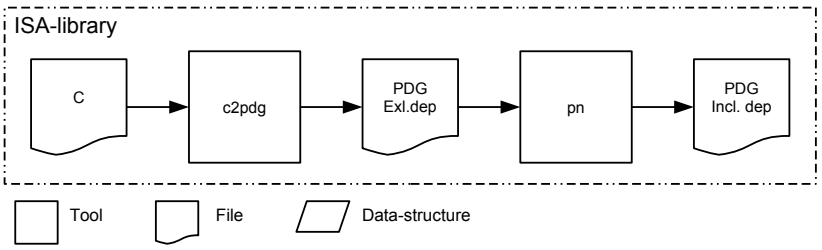


Figure 2.7: The ISA library tool-flow

2.5.2 Polyhedral Process Networks Implementation

Having discussed the abstract idea of a PPN, let us consider details of its implementation in the ISA library as a PDG. This PDG is implemented as a C++ class. Figure 2.8 shows a simplified class diagram of the PDG data-structure. Some details that are currently irrelevant and could only add confusion are deliberately left out.

The *PDG* class implements a graph data-structure by providing a set of nodes; and a set of edges, that link two nodes. The latter are implemented as a vector of type *node* (i.e. the nodes of the graph) and a vector of type *dependence* (i.e. the edges of the graph).

The nodes themselves are implemented by the *node* class. A node object (i.e. an instantiation of a *node*) has a number (nr) giving a node a unique identifier. It also has a name, which is either the label of the statement in the input code, or an automatically generated name of the form *S_ <nr>*, in case no labels are provided by the input code.

The *source* member represents the iteration domain of the node as an *isl_set* (i.e. a Presburger set), while *schedule* represents the schedule of the node, it is an *isl_map* (i.e. a Presburger relation). It is noteworthy to mention that this schedule is a global schedule, in context of the complete network. The statement represents the actual statement implemented by this node. The *pdg:statement* class details are discussed later.

[2]The acronym officially stands for: YAML Ain't Markup Language. "YAML is a human friendly data serialization standard for all programming languages." According to yaml.org

Recall that a node represents a statement in the original code. This statement is represented by the member *statement*. This member is of type *statement*. An object of type *statement* provides a list of input accesses of the statement as well as a list of outputs access. The members of both these lists are of type *access*. The *seq* type, used for implementing the lists, is a wrapper type, under the hood it is actually a *std::vector*.

The actual data manipulation done by the statement is abstracted using a object of type *function_ call*. Operators like addition (+) and multiplication (*) are also regarded as functions. A *function_ call* object has a name, which is a C-string containing the name of the function in the original C code. In case of an operator, the name is the operator symbol as used in the C code. The object provides a vector of arguments. These arguments are of type *call_ or_ access*, which is a type that behaves either as a call or a access. An argument it self could actually be the result of a function call or operation. A *fuction_ call* object is very similar to an Abstract Syntax Tree (AST) of the original C statement.

The dependences in the graph are modelled as edges between two nodes, actually between two accesses. Therefore the *dependence* specifies the two nodes involved in the dependence (the members *from* and *to*), but also the two accesses (*from_ access* and *to_ access*).

A dependence is associated with a specific array, the member *array* is a pointer to an object representing the associated array. Furthermore the class holds some tags that give information regarding the nature of the dependence: *reordering* tells whether the dependence reorders data accessed, while *multiplicity* tells whether elements written to the channel are read multiple times at the other end. The latter indicates reuse and is usually transferred into a self dependence on the node by the *pn* tool.

The *value_ size* member gives the size of a buffer. In general this member is a function that expresses the size in term of the parameters. If the parameters are known at compile time an actual number can be deduced from this function.

The class *access* abstracts an access. The *map* represents the access function discussed earlier in the theoretical discussion. A pointer to the associated array object is kept in *array*. An integer number is used to give the access an identifier, this identifier is not globally unique (i.e. in the context of the PDG), but only locally (i.e. per node) unique. The type of the access, *read* and *write*, is given by *type*.

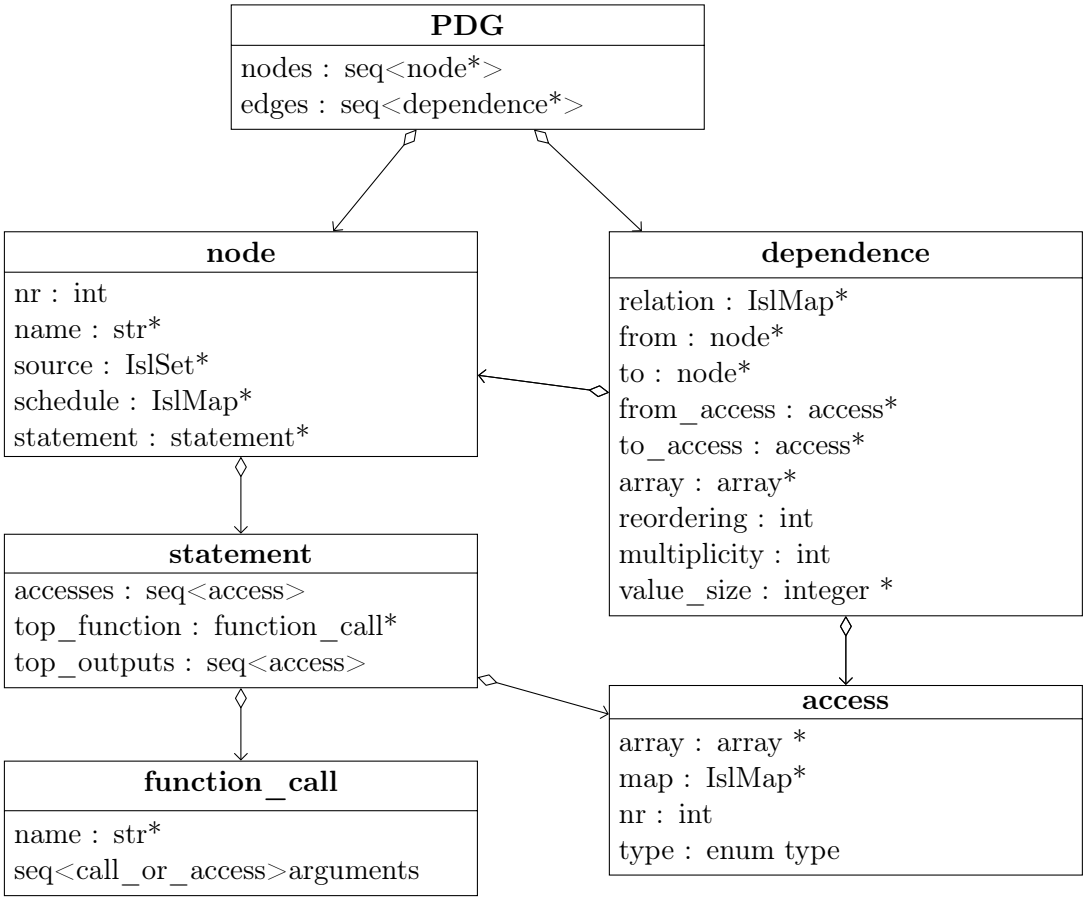


Figure 2.8: Class diagram of the PDG data-structure

3 Data flow computing

In this chapter the general concept of data flow computing is covered as well as the specific approach, Maxeler Data Flow Engines (DFEs), used in this work.

3.1 Data Flow Computing vs Control Flow Computing

There are two computing paradigms that are relevant in the scope of this thesis. The most widespread paradigm now-a-days is the control flow paradigm. All mainstream computing is in fact control flow computing. The basic idea of the control flow paradigm is a fixed processor architecture that is applicable to all Turing-computable problems. In control flow computing the control is programmable using binary instructions.

The data flow paradigm is different in that it envisions a specific data path to be constructed for each particular problem. The data is streamed through this data path. The control, if any, is simple and there is no need for binary instructions. The logical conclusion is that the data flow paradigm requires a computational vehicle that allows a custom data path to be constructed. One device that can emulate this is an Field Programmable Gate Array (FPGA).

Data flow computing is usually harder to implement than control flow. Platforms enabling data flow computing, oftem involving FPGAs, usually run at a lower clock-frequency, compared to a General Purpose Processor (GPP). However the achievable throughput on these platforms is higher when compared to control flow computing platforms. One advantage to data flow computing that is in particular relevant today, is the greatly reduced power consumption. Given the increasing power consumption of computing today, and the demand to reduce carbon-dioxide emission, data flow computing might be a significant contributor to sustainable future.

In the field of High Performance Computing (HPC) data flow based computing might be the solution to the power crisis control flow based computing is currently facing[13]. The later is especially significant in the pursued for exa-scale computing within a affordable energy budget.

3.2 Computation on FPGA

In section 1.1.1 we have briefly touched upon the challenges of reconfigurable HPC. In this section we will elaborate on this subject in the specific case of using FPGAs as computational vehicle.

An Field Programmable Gate Array (FPGA) is a logic device that can be reconfigured in order to implement a digital logic system. Reconfiguring an FPGA can be done post production, and not just ones but as often as required by the application. The basic form of an FPGA is an array of configurable look-up tables connected via a configurable network of wires. This allows for the desired flexibility at the expense of logic density and speed compared to a fully hard-wired digital system. Besides look-up tables, modern FPGA's also are equipped with configurable on-chip memory and arithmetic units.

FPGA's where first used as prototyping platform and for custom digital systems for low-volume markets. When used for general purpose or scientific computation, the reconfigurability of an FPGA allows to build specific computational units for a given algorithm. From the conceptual point of view this has obvious advantages. However, turning these ideas in reality can be challenging.

First, we should consider the features of an FPGA in order to understand its strengths and weaknesses when applied as computational device. One of the major advantages of doing computations on FPGA is that one can construct a complex custom data path for a statement. This statement can be an elaborated statement with multiple inputs and many operations. This statement is typically deeply pipelined. Furthermore, the data path can be replicated multiple times, on the surface of the chip, in order to execute parallel instances of the statement.

The achievable degree of parallelism and the degree of pipelining is much higher than what can be achieved with conventional systems. However, this may require high I/O bandwidth, in order to provide enough data so that the data path is fully utilised. The latter is a challenge due to the limited capacity of on-chip memory (bounded by silicon area) and the limited bandwidth to external memory (bounded by the number of pins of the package).

The major challenge seems to be the efficient usage of on-chip memory and external I/O bandwidth. It is not far fetched to assume that data reuse optimisation is a promising technique in this context. The on-chip memory is highly configurable and can be organised with an arbitrary width (number of bits per addressable data-word) and depth (the capacity in data-words). The latter comes with a speed penalty, as expected.

With regard to external memory, an FPGA usually provides multiple independent interfaces. The latter allows for parallelism. However the achievable parallelism is dependent on how the input data is distributed over the individual channels. Preferably one wants to distribute the input data over the available channels in an optimal way.

Probably one of the most impactful features of an FPGA is that it allows, within certain bounds, to construct custom arithmetic operations. An FPGA allows for the construction

of for example a floating-point adder of custom precision. But it could also be used to implement a specific complex function like a square root.

Both GPPs and Graphics Processing Units (GPUs) are equipped with a number of basic fixed precision arithmetic operators. A specific processor could for example be equipped with a 32-bit fixed point adder. When doing 16-bit fixed-point arithmetic this provided precisions is over-kill. However, when we want to do 64-bit fixed-point arithmetic multiple steps and additional control is required. On an FPGA one could implement the fixed-point adder with the required precision. According to [14] this is actually the most important feature of FPGAs that makes them compete with GPPs and GPUs in the HPC domain. The Flopoco project [15] is based on these ideas.

3.3 Maxeler Data Flow Computing

In the previous section we were introduced to the data flow computing paradigm. In this section we will discuss a particular approach to data flow computing. This particular approach is used in here as the target for the proposed tool-flow and methodology.

Maxeler is a company specialised in multi-scale data flow computing. In order to enable efficient data flow computing Maxeler has developed a complete solution composed of hardware, programming tools, a run-time environment and a dedicated tool-flow. The tool-flow allows for applications to be developed that run partially on a host Central Processing Unit (CPU) and partially on the Maxeler DFE.

3.3.1 Maxeler Data Flow Engine

A Maxeler Data Flow Engine (DFE) is an FPGA based computational architecture optimised for data flow computing. There are multiple DFE versions. The versions differ mainly in capacity while they have a general architectural idea in common. A DFE includes one or more FPGAs; is equipped with Synchronous Dynamic Random Access Memory (SD-RAM); and provides a high bandwidth bi-directional communication channel with a host system. The latter is usually realised via the, industry standard, Peripheral Component Interconnect Express (PCIe) bus.

The most common form of a DFE is a PCIe card that is additionally equipped with SD-RAM. Multiple such cards can be connected so that the DFEs can communicated directly. A box may contain more that one DFE, which are interconnected by a network. The network topology is a ring (i.e. each DFE is connected to two neighbouring DFEs).

3.3.2 Maxeler Tool flow

A Maxeler application consists of three parts:

Host Code The CPU part of the application, which can interact with the DFE implemented part through the Simple Live CPU (SLiC)-interface. This SLiC interface provides an Application Programmer Interface (API) to interact with the DFE implemented part of the application.

Kernel Code Implements the custom computing sub-system on a DFE. Running an actual computation, called an *action*, on a specific DFE implementation is initiated from the host code using the SLiC-interface.

Manager Code Implements the I/O between kernels of which multiple can be simultaneous active on one DFE. It also provides the interface to the on-card SD-RAM and the host, via PCIe, as well as possibly streaming links to other DFEs.

The mentioned SLiC-interface is a C API providing functions to interact with a DFE implementation. SLiC Skins provide native calls to the SLiC API in languages other than C. Such Skins for Python, Matlab and R among others are currently available.

Coding the DFE part, both the kernel and the manager, is done using Java. The DFE philosophy is not to translate general Java code to a DFE implementation. Rather, it provides the application developer with a library of Java classes to construct a data path with optionally some tightly-coupled control implemented using counters and multiplexers. Such code is referred to as MaxJ code.

By running the MaxJ code, a data flow graph is generated that in turn is translated to a DFE implementation. This process will also generate the SLiC-interface, which is specific for each DFE implementation. The tools also enable to simulate the DFE implementation.

3.3.2.1 Kernel

The example, shown in Listing 3.1 implements a kernel in MaxJ that performs a moving average calculation using a 3 value wide window on a stream of single-precision floating-point values. The object *DFEVar*, actually implements a stream. A stream is a channel through which data sequentially flows driven by a systolic clock.

A DFE kernel operates conceptually on the basis of a discrete unit of time. This latter is called a *tick* in Maxeler jargon. The *tick* can be thought of as a clock-cycle, however there is a bit more to it than that. In the actual hardware there might be some latencies because of PCIe transfers or due to data stream synchronisation (on the level of the kernel all data streams are synchronised). This however is hidden from the perspective of the application developer. So the *tick* is the unit of time in which the state of the kernel changes or, alternatively in which the data actually advances.

```
package movingaveragesimple;

import com.maxeler.maxcompiler.v2.kernelcompiler.kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.kernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class MovingAverageSimpleKernel extends Kernel {

  MovingAverageSimpleKernel(KernelParameters parameters) {
    super(parameters)

    DFEVar x = io.input("x", dfeFloat(8, 24));

    DFEVar prev = stream.offset(x, -1);
    DFEVar next = stream.offset(x, 1);
    DFEVar sum = prev + x + next;
    DFEVar result = sum / 3;

    io.output("y" result, dfeFloat(8, 24));
  }
}
```

Listing 3.1: A DFE kernel implementing the moving average computation

MaxJ code allows to read a value from a specific stream relative to the current *tick*. Therefore *stream.offset(x, -1)* gives the value of stream x at the previous *tick* while *stream.offset(x, 1)* gives the value of stream x at the next *tick*[1]. These concepts of *ticks* and *streams* provide an elegant mental framework that allows programmers to implement a computation on a DFE. It relieves a developer from much headache, compared to implementing such data flow kernel on an FPGA using a Hardware Description Language (HDL). In this case developers have to introduce FIFO's and shift-registers, to buffer and delay data and signals manually in order to synchronise data streams.

Another concept of MaxJ code demonstrated in the example code is the possibility of multi-precision arithmetic. MaxJ allows for customisable precision for both integer and floating-point arithmetic. This example shows how this works for floating-point arithmetic. The statement *dfeFloat(8, 24)* defines a floating-point type with 8 exponent bits and 24 mantissa bits. The latter is in fact the format of single precision floating point. However, as the reader can see any number of bits for both mantissa and exponent can be conveniently used.

The *io* object is used to define the inputs and outputs of the kernel. The *io.input* method returns a *stream* while the method *io.output* takes a *stream* as input. The string passed to both methods, functions as identifier used to address the kernel input/outputs at the DFE manager level.

Because of the streaming performed by an DFE there is some notion of an implicit outer loop. When the reference C implementation of a kernel has only one loop, the kernel

[1]The *tick* can be regarded as the clock-cycle of the kernel level clock-domain.

can be implemented without any notion of the loop. The loop is actually implemented in space because the data is *streamed* through the kernel. This is the very idea of the implicit outer loop.

When the original C implementation contains a more complicated loop nest, the kernel implementation usually requires some control to be implemented. This control is usually implemented using chained counters. I will discuss such an example in Section 5.2.

For loop constructs in MaxJ code implement parallel replication of the computation described in their body. This means that a for loop creates multiple parallel instantiations of the computation on the surface of the chip.

```
package movingaveragesimple;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.IOType;


class MovingAverageSimpleManager {
    public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
        Manager manager = new Manager(params);

        // Instantiate the kernel
        Kernel kernel = new MovingAverageSimpleKernel(
              manager.makeKernelParameters());

        manager.setKernel(kernel);
        // Connect all kernel ports to the CPU
        manager.setIO(IOType.ALL_CPU);
        manager.createSLiCinterface();
        manager.build();
    }
}
```

Listing 3.2: DFE manager implementing of a moving average

3.3.2.2 Manager

The manager can be regarded as the interface between one or more kernels and the outside world. Within the manager code the kernel's inputs and outputs are connected to either *streams* from and to the host or from and to the on-card memory. In multi-DFE configurations, the manager code is used also to connect *streams* from and to neighbouring DFEs.

The manager code for the *moving average example* is shown in Listing 3.2. Most of

this code is rather obvious. A *manager* object is instantiated, then the earlier discussed *kernel* is instantiated and the *kernel* is "added" to the *manager*. The method call *manager.setIO(IOType.ALL_CPU)* connects all the inputs and outputs of the kernel to the PCIe between the DFE and the CPU.

The method call *manager.createSLiCinterface()* generates an API that can be used to interact with the DFE from the host code.

3.3.2.3 Host code

The third part of a Maxeler DFE implementation is the host code. The host code initiates DFE execution, sends data to the DFE and receives data back from the DFE.

During DFE development one will usually include a reference implementation in the host code. and compare the results of the DFE and the reference code, in order to validate the former.

Listing 3.3 shows the host code of the example we have been discussing. In this example running the DFE is as simple as a single function call. The function call *MovingAverageSimple(8, dataIn, dataOut)* is used to run the DFE. The input and output array are provided as well as the size of the input data. The inputs are arrays in the host memory which are transferred to the DFE using Direct Memory Access (DMA) transfers over PCIe. The output data is stored in the *dataOut* array. The function call used here is blocking (i.e. will return only after the output data has been produced).

More advanced interactions, like for example non-blocking calls, are provided. But are deliberately excluded from the current discussion. It is also possible to communicate scalars to the DFE, the SLiC-interface will accommodate this using function arguments of the function that is meant to initiate DFE execution.

```
#include "Maxfiles.h"              // Includes .max files
#include <MaxSLiCInterface.h>     // Simple Live CPU interface

float dataIn[8] = { 1, 0, 2, 0, 4, 1, 8, 3 };
float dataOut[8];

int main()
{
    printf("Running DFE\n");
    MovingAverageSimple(8, dataIn, dataOut);

    for (int i = 1; i < 7; i++) // Ignore edge values
        printf("dataOut[%d] = %f\n", i, dataOut[i]);

    return 0;
}
```

Listing 3.3: Host code for the moving average DFE implementation

4 State of the Art

This chapter describes related research both from academia and industry. The chapter is divided in two major parts: Section 4.1 discussing approaches that similar to Maxeler's Data Flow Engine (DFE) technology target Field Programmable Gate Array (FPGA) based computing; and Section 4.2 discussing projects and frameworks using Polyhedral Compilation (PC).

4.1 FPGA based Computing Platforms and Tools

This section discusses platforms and tools that target FPGA based computing. This section is restricted to approaches that are similar to Maxeler's DFE approach. A detailed analysis of these alternatives is not within the scope of this thesis. However, to give some idea regarding the performance of Maxeler DFE Technology in comparison with a number of alternatives, the reader is referred to[1]. This paper compares an implementation of gzip on DFE with two other FPGA based gzip implementations. One is a Open Computing Language (OpenCL) implementation and the other a Verilog implementation.

It is shown in[1] that the DFE implementation out performs the two other implementations. It also makes a compelling case for the higher productivity in DFE implementation as compared to OpenCL and Verilog.

4.1.1 OpenCL

The OpenCL is a framework for writing applications for heterogenous systems. OpenCL is an open standard originally developed by Apple Inc. but currently maintained by a non-profit consortium called Khronos Group. Both proprietary and open-source implementations of the standard exist for various platforms. Most implementations make use of C or C++ syntax width specific extensions. An OpenCL implementation is usually composed of some host code that runs on a General Purpose Processor (GPP) and one or more compute kernels that can be run on an accelerator. One of the major appeals of OpenCL is that a kernel's functionality is portable between accelerators, however performance is not portable. The fact that the performance is not portable exposes a major issue facing OpenCL. A consequence of the lack of performance portability necessitates a OpenCL kernel to be optimised for each specific computational device, which challenges the claimed portability of OpenCL.

OpenCL can be used to code FPGA implementations. Software Development Kits (SDKs) are provided by both Xilinx and Altera/Intel to do the latter, enabling FPGA based accelerator cards to be programmed using OpenCL. There are various FPGA acceleration cards available on the market. They usually interface to a host system using the Peripheral Component Interconnect Express (PCIe) bus and have Synchronous Dynamic Random Access Memory (SD-RAM) memory available on the card. Like in the mainstream computing market the choice for SD-RAM is motivated by the capacity and reduced cost compared to Static Random Access Memory (SRAM). Besides external memory these cards usually provide other I/O interfaces such as TCP/IP networking etc. One specific accelerator card is discussed in more detail in the next section.

4.1.2 The Intel Acceleration Stack for Xeon CPU with FPGAs

Intel provides a stack for FPGA accelerated computation. This stack provides an Application Programmer Interface (API) to the application programmer through which the FPGA can be controlled and reconfigured. It provides also functions to write to and read from registers in the FPGA; as well as functions for accessing data using a shared memory. There is an Intellectual Property Core (IP-CORE) library available to create accelerators. The actual communication exposed by the API is implemented by an FPGA Driver. The FPGA is configured with a static part that implements the communication between the host and the user defined accelerator. Applications are developed using OpenCL

4.1.3 Xilinx Vivado High Level Synthesis (HLS)

Vivado is Xilinx's HLS solution. It allows for hardware creation from C, C++ and System C specifications. It is meant to easy the creation of custom hardware for Xilinx programmable devices. Its primary aim is to boost productivity compared to using Hardware Description Languages (HDLs). Vivado HLS does not provide a solution for data flow computing.

4.1.4 Xilinx SDAccel

SDAccel is a famework for FPGA acceleration developed by Xilinx. It allows to develop applications that partially run on a Central Processing Unit (CPU) and partially on an FPGA. SDAccel is aimed at datacenter employment of FPGA acceleration. The CPU part of the application is developed in C/C++ using OpenCL API calls, while the hardware part of the application is developed in C/C++, OpenCL or Register Transfer Level (RTL).

4.1.5 Chisel

Chisel[16] is a hardware construction language. Like MaxJ it uses an existing language to embed hardware generation constructs. Chisel uses Scala a Java like language that is object oriented and also supports functional programming. Chisel is developed as an open-source language at UC Berkeley. The language is primary aimed at generating hardware modules at a higher level of abstraction compared to HDLs and is not specifically aimed at data flow computing. It does not provide the notion of *streams* which are prevalent in MaxJ code. The Chisel language translates to low-level Verilog to be passed to FPGA or Application Specific Integrated Circuit (ASIC) tools. It does not provide a complete solution that both integrates the tools and a hardware platform.

4.1.6 Liquid Metal

The Liquid Metal project is an approach in some sense similar to the goals of OpenCL, in that it envisions one language to be used to program both Graphics Processing Unit (GPU) based and FPGA based accelerators. The language, developed as part of the project, is called Lime and is described as "a Java-compatible synthesisable language for heterogenous architectures"[17]. The Lime language is based on Java and supports data flow computing.

4.1.7 BlueSpec Verilog

Bluespec SystemVerilog (BSV)[18] is a high-level HDL, developed by Bluespec. A compiler is provided that compiles BSV into synthesisable Verilog RTL or SystemC. The approach is comparable to Chisel. The language syntax has the taste of a higher level HDL and not a data flow language, it lacks the higher abstraction concepts MaxJ and Lime provide.

4.2 Frameworks using Polyhedral Compilation

The are a number of mostly academic projects that have resulted in compiler and code transformation frameworks using Polyhedral Compilation (PC). The most relevant approaches are discussed in this section. This discussion is focused primarily on approaches that target High Performance Computing (HPC) and accelerators or are for other reasons relevant to this thesis. None of the approaches target FPGAs specifically.

4.2.1 CARP Project

A research project related to this work is the Correct and Efficient Accelerator Programming (CARP) project. The goal of the CARP project was to develop techniques and tools to ease the programming of accelerators. The overall idea of the project is to enable

accelerator programming using Domain Specific Languages (DSLs). The DSLs are compiled into Platform Neutral Compute Intermediate Language (PeNCIL) code. PeNCIL in essence is a subset of C-99 with additional pragmas an other means of annotating the code. PeNCIL is, because it is a C-99 subset, human readable. Coding directly in PeNCIL is therefore also possible. Using PC the PeNCIL code is compiled into OpenCL code. Target hardware in the CARP project is GPUs, to the authors knowledge FPGAs have not been targeted in this research.

4.2.2 PPCG

The Polyhedral Parallel Code Generator (PPCG)[19] is a compiler that generates parallel code from a sequential C code input. The current focus of the project is on code generation for GPUs. The PPCG project has its origins in the CARP project.

4.2.3 Polly

Polly[20] is implemented as a LLVM[1] compiler pass. Polly uses PC to perform optimisations. Polly works on the LLVM Intermediate Representation (LLVM-IR) which is the specific Intermediate Representation (IR) LLVM uses. Like most compiler IRs, it is implemented as an assembly-like language.

4.2.4 Pluto

Pluto is a tool focussed on tiling transformations of input code. It is the result of the Phd. research of Uday Bondhugula[7]. A tiling scheduling algorithm is at the heart of the Pluto project. An implementation of this algorithm is used in the Integer Set Library (ISL).

[1]The original acronym LLVM is inherited from the founding project, and is kept, but is currently regarded the name of the project not an acronym.

5 Design and Architecture

This chapter describes the architecture of the tool developed during this thesis. The envisioned function of this tool is to generate a Data Flow Engine (DFE) implementation based on a Polyhedral Process Network (PPN) produced by the tools from the Integer Set Analysis (ISA) library. These tools produce a Polyhedral Dependence Graph (PDG) which is a specific implementation of the PPN concept. In this chapter the architecture of this tool is specified, additionally some design choices during its development are discussed. The specific implementation of the tool will be discussed in the next chapter.

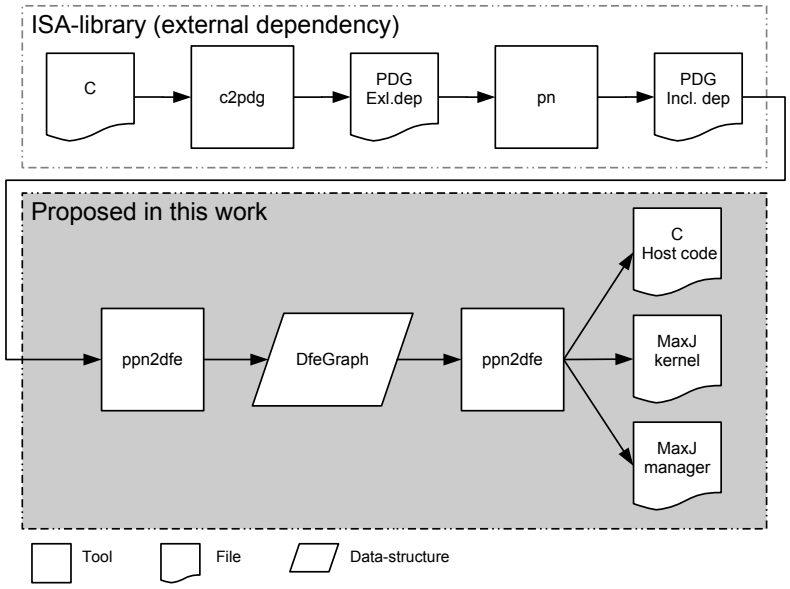


Figure 5.1: Tool Flow

5.1 Architecture

Figure 5.1 shows the envisioned tool-flow of the tool developed in this work, as well as the tools used from the ISA library. A PDG file forms the input for the *ppn2dfe* tool developed as part of this thesis. This input PDG file must be the output of the *pn*

tool. This tool performs the dependence analysis and adds the dependences to the PDG (i.e. the PDG produced by *c2pdg* does not include the dependences). The *pn* tool also annotates the dependences in the PDG with size and type information. This information specifies the volume and type of a communication channel required to implement this dependence.

The ISA library provides a tool that can produce an alternative representation of a PPN. The latter is an Approximated Dependence Graph (ADG) which can be generated from a PDG. The ADG is more specific and is tailored to a particular implementation strategy not necessarily compatible with the implementation target envisioned in this work.

A PDG is stored in a YAML[1] file, this file contains a serialisation of a C++ data structure. The ISA library provides a function to load this YAML file returning a C++ data structure. This structure is the PDG described in Section 2.5.2.

The envisioned output of the tool is a DFE implementation consisting of: host code, kernel code and manager code. Currently the kernel code generation is implemented in the tool, provisions have been made to generate the other two files. Because the kernel code generation is the most challenging part of the tool flow the focus has been on this part during this thesis. The implementation of the other two parts is therefore left to future work.

In the tool-flow the data-structure *DfeGraph* is used as an intermediate abstraction, while translating a PDG to MaxJ code. This *DfeGraph* is implemented as part of this thesis.

5.2 A Manual MaxJ Implementation

In order to attain insight in what is exactly involved in generating a MaxJ kernel from a PDG, a manual implementation based on a PDG was first implemented. The PDG used for this experiment is generated from a matrix multiplication kernel written in C. It is in fact the example we have used throughout this thesis.

In this section we will go through this manual implementation and consider how it corresponds to the related PDG. The PDG under consideration is shown in Figure 5.2, while the kernel in MaxJ is depicted in Listing 5.2. Figure 5.4 can be consulted for a graphical depiction of the implementation.

This MaxJ code performs a naive sequential implementation, which can actually be improved significantly. However in order to get some idea of how the correct MaxJ code can be automatically generated from the information provided by the PDG this is actually very helpful.

[1] "YAML is a human friendly data serialization standard for all programming languages." yaml.org

```
int ReadMatrix();
void WriteMatrix(int output);

int main(void){

  int A[100][100];
  int B[100][100];
  int C[100][100];

  for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
       ReadA: A[i][j] = ReadMatrix();
       ReadB: B[j][i] = ReadMatrix();
       InitC: C[i][j] = 0;
    }
  }

  for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
      for (int k = 0; k < 100; k++) {
         Comp: C[i][j] = C[i][j] + A[i][k] * B[k][j];
      }
    }
  }

  for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 100; j++) {
       WriteC: WriteMatrix(C[i][j]);
    }
  }

  return 0;
}
```

Listing 5.1: The input code used by the ISA tool-flow

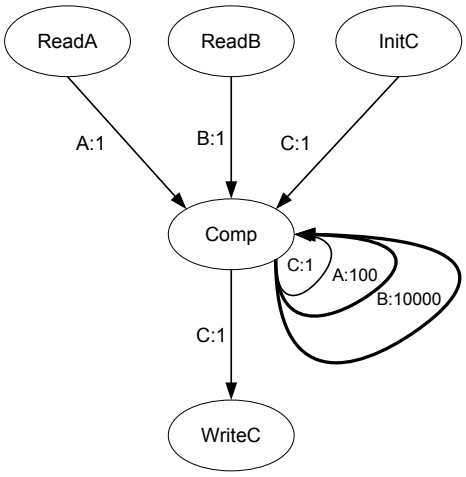


Figure 5.2: The PPN Corresponding to the Matrix Multiplication Kernel

The general idea of this implementation is that it stores a single row of matrix A and the complete matrix B in two separate memories. Then sequentially the inner products of the row of A and the columns of B are computed. After that the next row of A is fetched and inner product computation is repeated in order to compute the next row of the output matrix C.

One of the consequences of the latter is that the data of matrix A can not be streamed continuously. Instead this data is transferred in bursts (i.e. per matrix row). This is accomplished by using a counter-chain. A counter-chain is simply a number of counters which wrap-signal (or overflow) is connected to the enable-signal of the next counter in the chain. The behaviour of such counter-chain is similar to how a nested for loop iterates through an iteration space.

The MaxJ code implements the three iterators of the original code i, j and k using a counter-chain, in fact the counter-chain has one extra member we will discuss later.

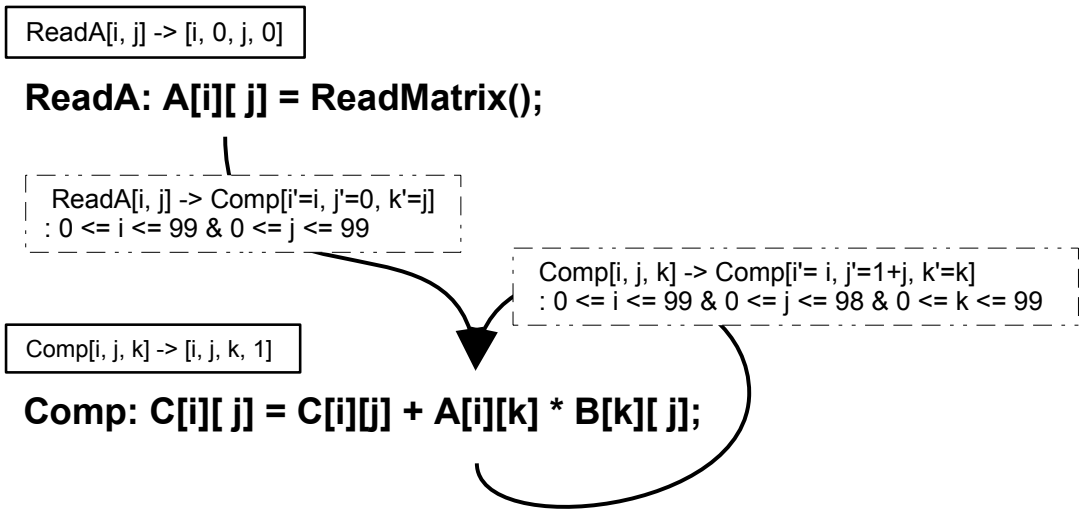


Figure 5.3: Part of the PDG related to the data reuse on array A

Using the counter values the data transfers can be controlled. To do this an *io.input* is used with an additional enable signal. This signal controls whether data is read from this input. From the original code, but also from the PDG it can be deduced that the new data must be read when $j == 0$. Figure 5.3 shows the part of the PDG relevant to the data reuse on array A. In order to deduce that array A data is read from the host when $j == 0$, we consider the the node labelled *ReadA* in the partial PDG shown in Figure 5.3.

Note that this node models, in the PDG interpretation used in this thesis, the data communication between the host and the DFE. What it in fact does is describe the order in which the data is streamed to the DFE. How this is represented in the PDG is best understood when considering the schedule of node *ReadA* as described by the following Presburger relation.

```
1  package sequential_mat_mult;
2
3  class sequentialMatMultDFEKernel extends Kernel {
4    sequentialMatMultDFEKernel(final KernelParameters parameters) {
5      super(parameters);
6      final DFEType scalarType = dfeFloat(8, 24);
7      // Retrieve the feedback loop length in number of ticks
8      OffsetExpr loopLength = stream.makeOffsetAutoLoop("loopLength");
9      DFEVar loopLengthVal = loopLength.getDFEVar(this, dfeUInt(4));
10     // Counter chain
11     // Created a counter to deal with the feedback
12     Count.Params paramsLoopLengthCounter = control.count.makeParams(4)
13         .withMax(12);
14     Counter loopLengthCounter = control.count.makeCounter(
15        paramsLoopLengthCounter);
16     // Create a counter for iterator k and add it to the counter-chain
17     Count.Params paramsCountK = control.count.makeParams(4)
18         .withEnable(loopLengthCounter.getWrap())
19         .withMax(16);
20     Counter CountK = control.count.makeCounter(paramsCountK);
21     // Create a counter for iterator j and add it to the counter-chain
22     Count.Params paramsCountJ = control.count.makeParams(4)
23         .withEnable(CountK.getWrap())
24         .withMax(16);
25     Counter CountJ = control.count.makeCounter(paramsCountJ);
26     // Create a counter for iterator i and add it to the counter-chain
27     Count.Params paramsCountI = control.count.makeParams(5)
28         .withEnable(CountJ.getWrap())
29         .withMax(16)
30         .withWrapMode(WrapMode.STOP_AT_MAX);
31     Counter CountI = control.count.makeCounter(paramsCountI);
32
33     DFEVar i = CountI.getCount();
34     DFEVar j = CountJ.getCount();
35     DFEVar k = CountK.getCount();
36     DFEVar l = loopLengthCounter.getCount();
37     // Inputs
38     final DFEVar a_in = io.
39       input("a_in", dfeFloat(8, 24), j === 0 & i < 16 & l === 0);
40     final DFEVar b_in = io.input("b_in", dfeFloat(8, 24), i === 0 & l === 0);
41     // Memory alloc. array A stores one row of A
42     Memory<DFEVar> rowOfA = mem.alloc(dfeFloat(8,24), 16);
43     DFEVar a_int = rowOfA.port(
44         k, a_in, j===0 & i<16 & l===0, RamWriteMode.WRITE_FIRST);
45     // Memory alloc. array B stores all of array B
46     Memory<DFEVar> matrixB = mem.alloc(dfeFloat(8,24), 16*16);
47     DFEVar b_addr = j.slice(0, 4) # k.slice(0, 4);
48     DFEVar b_int = matrixB.port(
49        b_addr, b_in, i===0 & l===0, RamWriteMode.WRITE_FIRST);
50     // The multiplication
51     DFEVar mult = a_int * b_int;
52     // The loop body itself
53     // At the head of the loop, we select whether to take the initial value,
54     // or the value that is being carried around the loop cycle
55     DFEVar carriedSum = scalarType.newInstance(this); // sourceless stream
56     DFEVar sum = k === 0 ? 0.0 : carriedSum;
57     DFEVar newSum = mult + sum;
58     DFEVar newSumOffset = stream.offset(newSum, -loopLength);
59     // At the foot of the loop, we add the backward edge
60     carriedSum <== newSumOffset;
61     // We have a controlled output to deliver the sum at the end of each row
62     DFEVar outputValid = k === 15 & l === (loopLengthVal-1);
63     // Output
64     io.output("c_out" , newSum, scalarType, outputValid);
65   }
66 }
```

Listing 5.2: Matrix multiplication DFE implementation

$$\{ReadA[i,j] \rightarrow [i,0,j,0]\} \tag{5.1}$$

This relation (i.e. a schedule) maps each statement instance of statement *ReadA* to a global timestamp. This global timestamp is a state of the counter-chain. The timestamp is 4-dimensional, however the first (i.e. rightmost) dimension is constant. This dimension is used to specify the textual order in the original code. This notion is irrelevant in the context of a DFE implementation and is therefore ignored. The actual schedule we consider is:

$$\{ReadA[i,j] \rightarrow [i,0,j]\} \tag{5.2}$$

What this schedule dictates is the following: *The statement instance at the coordinate [i,j] in the instance set, is executed at timestamp [i, 0, j] (e.g. ReadA[1,2] is executed at timestamp [1, 0, 2]).* These timestamps are "generated" by the counter-chain. We can express this timestamp in the counters (i, j and k) as follows: [i, j, k]. From this we can conclude that the statement *ReadA* is only executed when counter j is zero.

In the MaxJ code there are actual some additional constraints. One of them is $i < 16$ which stops the reading when the whole matrix has been fetched. The other condition are explained later.

The counters are also used for the addressing of the memories. In order to address the memory that stores a row of A, iterator k can be used. Because each dimension of this particular hard-coded problem is a power of two, a simple concatenation of iterators i and j can be used without wasting memory. The latter trick could be used in general but would, when the iterator range is not a power of 2, lead to a some memory slack. In such cases an arithmetic expression can be used to generate the address. This latter approach requires a multiplier and an adder.

This brings us to the computational part of the algorithm. Form the very nature of matrix multiplication we know that the computational part is a multiply-accumulate. It is a multiplication of two scalars, which results are accumulated by addition per inner-product. The multiplication is rather straight forward to implement in MaxJ. The accumulation, however requires some extra attention. The accumulation results in a feedback path and because the floating-point adder generated is pipelined, the result is only available after a number of clock-cycles, twelve in this particular case. Therefore in this naive implementation the streams need to be stalled in order to wait for the result of the addition to become available. Next we will see how this is implemented in MaxJ.

The accumulation is implemented starting at line 55 of Listing 5.2. It starts with instantiating a so called source-less stream, which source we will connect later. The next line implements a multiplexer that is used to initialise the sum to "0" at the start of each inner-product computation. The line after that implements the addition. The next line is a *stream-offset* operation we have discussed earlier. This time it is used to read a previous value from the stream. This is used to synchronise the result being feedback with the

result of the multiplication. The method $stream.makeOffsetAutoLoop$ can automatically determine the maximal feedback loop within the kernel in terms of *ticks*. Finally the offset *stream* is connected as source to the earlier instantiated source-less *stream*.

The feedback path requires to delay the result. This is actually what the $LoopLengthCounter$ is used for. This counter actually implements an additional inner-loop. This counter is used as an additional condition to the input enable signal, for this very reason. It is also used in determining the output enable that signals when there is valid data available on the output. The *stream* or $DFEVar$ called $OutputValid$ defines the valid condition.

5.3 Generating MaxJ from a PDG

In this section we explore the input PDG and how it can be used to generate the required output. Comparing the PDG in Figure 5.2 with the MaxJ code in Listing 5.2, it is clear that there is no trivial one to one correspondence between the two.

There are multiple constructs in the MaxJ code that have no direct relation to the PDG. The MaxJ code includes the counter-chain that is not present in the PDG. Other constructs missing form the PDG are the memories that are used to buffer A and B and the multiplexer. Some further analysis reveals that the information to generate the MaxJ constructs is present in the PDG. However, extracting it is not always trivial, moreover the code constructs can not always be generated in one go. It is often necessary to add some information to earlier detected constructs. When directly generation the MaxJ syntax, the latter procedure will be complex if not impossible to implemented. Therefore the choice was made to develop an intermediate abstraction to aid the MaxJ generation from a PDG.

In the MaxJ code one can only refer to a $DFEVar$ that has been previously defined, as is common in programming languages. Therefore the manual implemented MaxJ code in Listing 5.2 gives the order in which the output code should be generated. From this code, we see that the counter-chains must be generated first. There is no direct notion of these counter-chains present the PDG shown in Figure 5.2. However, the information to generate the counter-chain is present in the PDG. The bounds for the counter-chain (i.e. the global iteration domain of the PDG in Polyhedral Compilation (PC) terminology) can be extracted from the union of the node schedules. In the next chapter we will discuss the exact implementation.

5.4 The DfeGraph

The intermediate abstraction developed is essentially an abstract representation of the target MaxJ code. The nodes represent structures like memories, multiplexers, and computational operations. Nodes are connected using streams, which more or less represent the DFEVars in a MaxJ implementation.

Let us examine the intermediate representation by example. For this example we use the earlier introduced matrix multiplication code. For convenience we repeat the code in Listing 5.1. Figure 5.4 depicts a graphical representation of the intermediate abstraction from now on referred to as a *DfeGraph*.

A *DfeGraph* implements a representation that can easily be mapped to MaxJ code. This is obvious when comparing the particular instantiation shown in Figure 5.4 to the MaxJ code shown in Listing 5.2.

Figure 5.4 shows that there are two distinguishable parts to this graph: namely the counters implementing control and the data path implementing the computation. In Figure 5.4 some details have been deliberately left out to prevent confusion.

The *DfeGraph* is generated while traversing the input PDG. The basic idea is that for each node it is decided based on some characteristics of that node what should be added to the *DfeGraph*. Figure 5.5 depicts the transformation of a PDG to a *DfeGraph*. Much detail has been left out of this picture, however it gives a high-level overview of the process, and how parts of the two graphs relate.

How each particular node of the *DfeGraph* is generated is discussed in the next chapter.

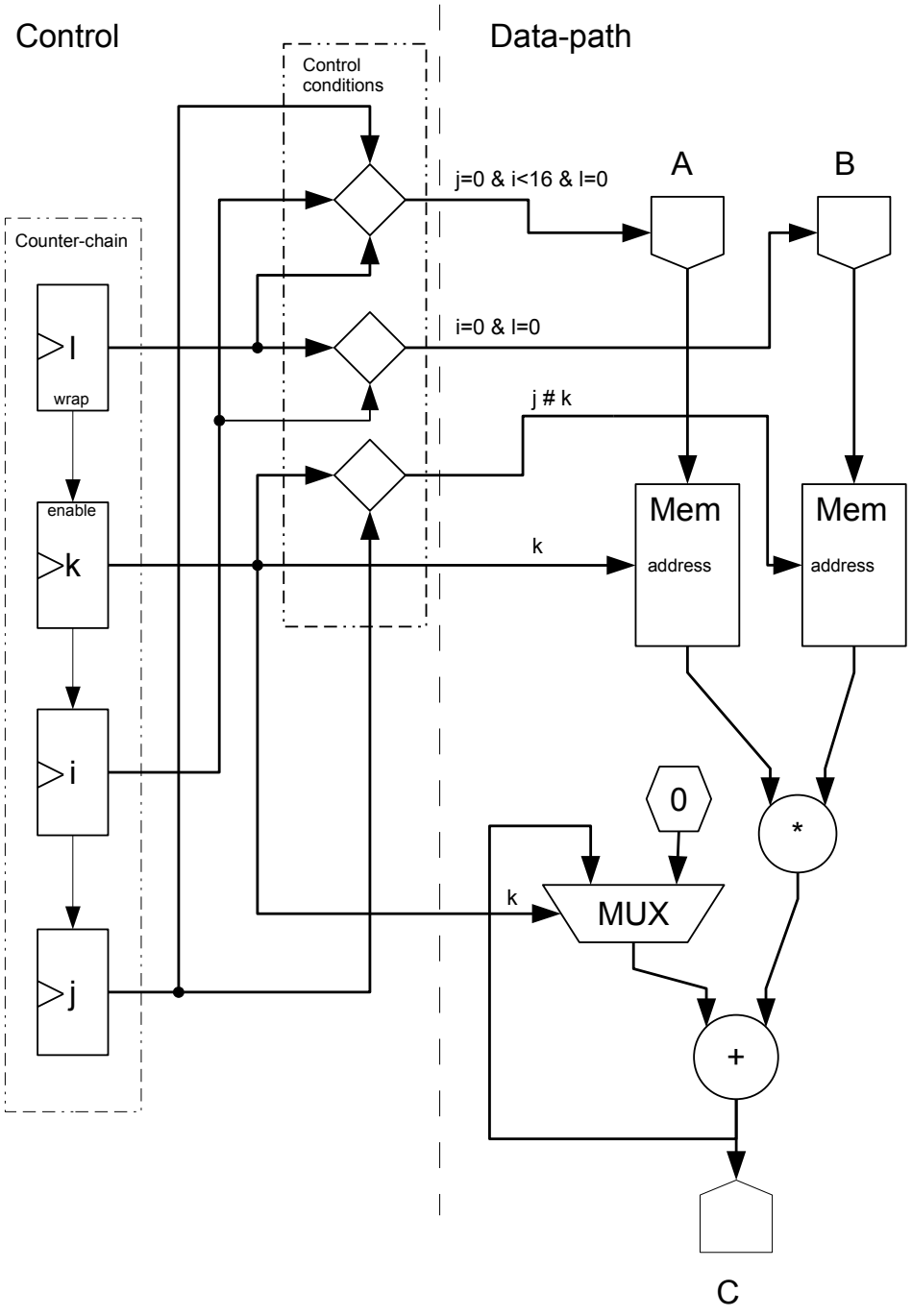


Figure 5.4: An example of the intermediate representation (i.e. DfeGraph)

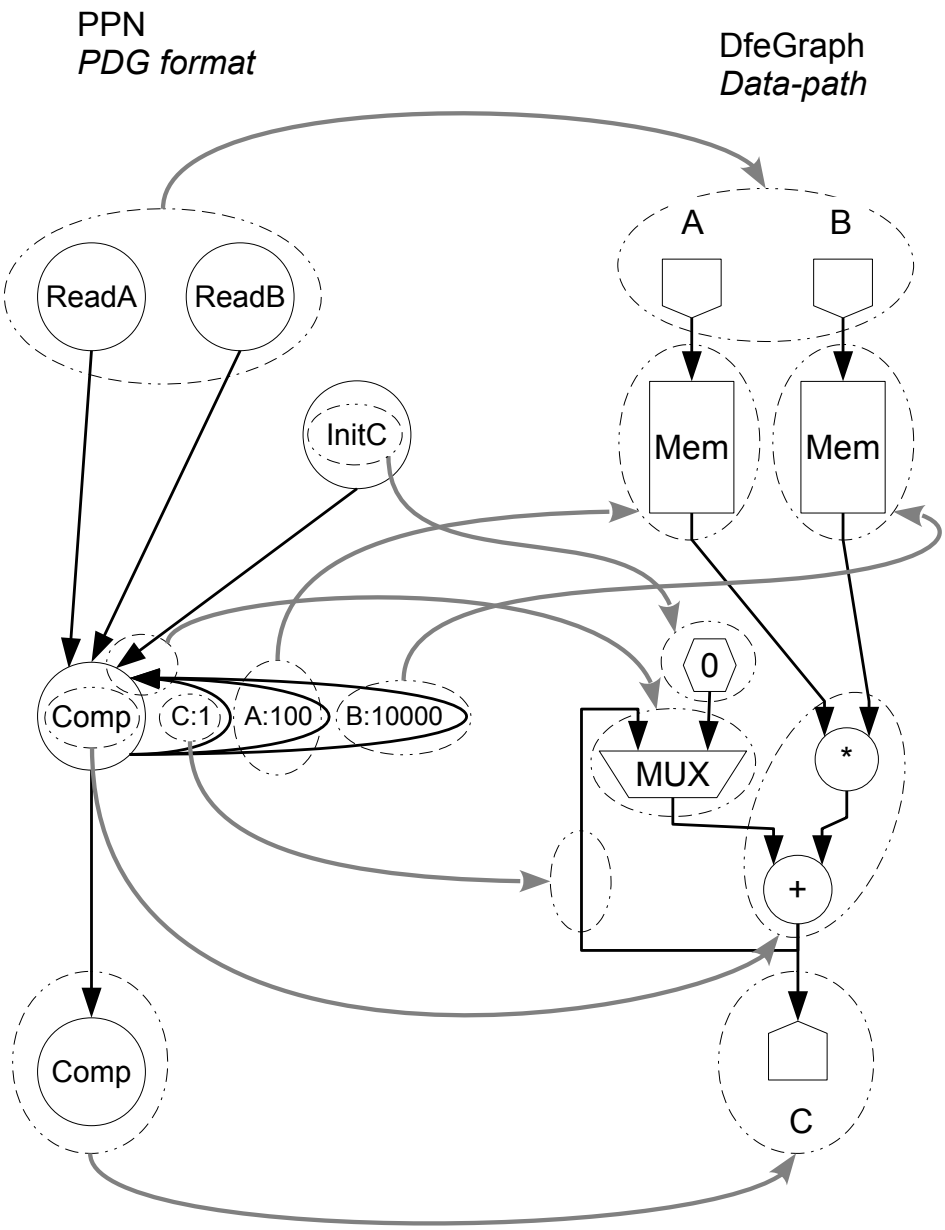


Figure 5.5: The DfeGraph data path generation from a PDG

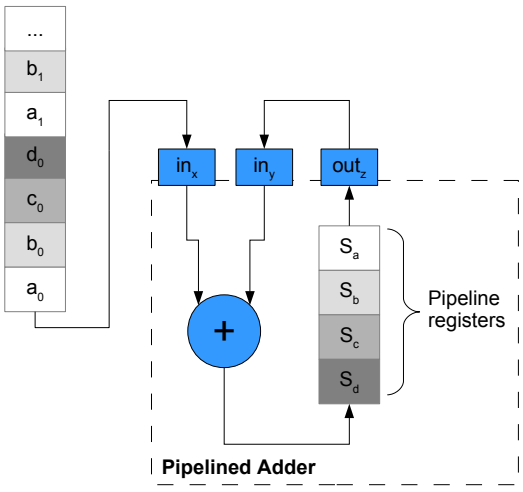


Figure 5.6: C-Slow retiming for C=4

5.5 Transformations, Parallelism and C-slow retiming

Although it has turned out to be infeasible within the duration of this thesis to implement transformations and parallelisation some design considerations are discussed next.

A data parallel view on PPNs is proposed in[21], these ideas are not implemented in the current PPN implementation in the ISA library. Therefore to enable parallelism these ideas need to be implemented in the ISA library.

As discussed in Section 5.2, feedback paths on pipelined operators are a performance issue, because they stall the data path and require significant on-chip memory for buffering data. A solution to this problem is the technique called C-slow retiming. The primary observation is that a pipelined accumulator (i.e. an operator whose result is feedback to one of its inputs). Can actually compute C independent accumulations in parallel. In order to exploit this the order of the operations has to be changed, such that C independent accumulations are executed in interleaved order.

The transformation that is of particular interest to this work is *tiling*. A tiling transformation splits a nested for loop in partitions that only need to communicate before and after the execution of the tile. The amount of data required by a tile and the data volume it has to communicate can be controlled by the tile-sizes, which can be arbitrarily chosen.

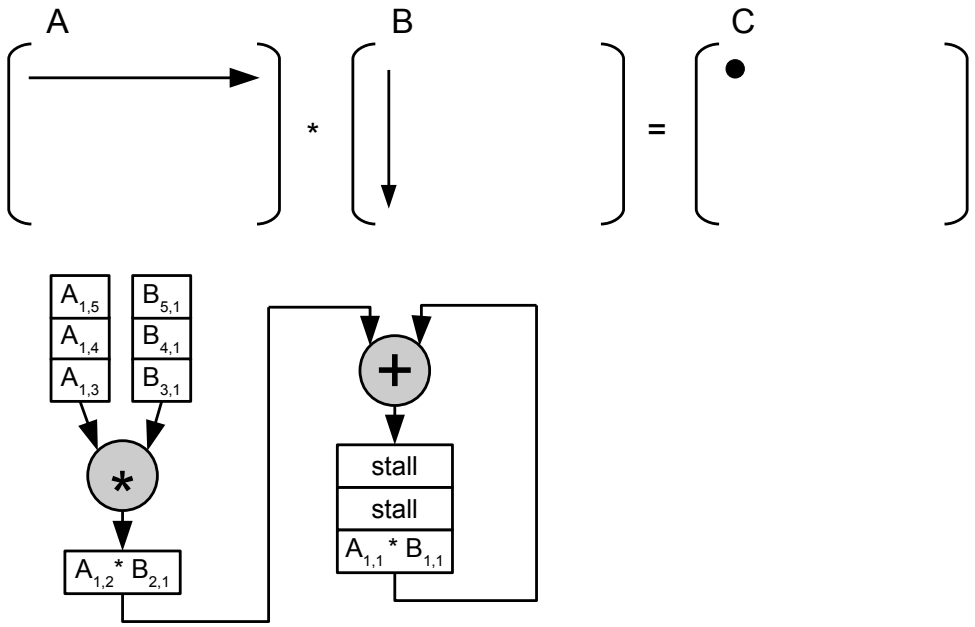
Tiling can be used to split a problem into parts for which the data can be kept in the DFE memory. Furthermore a tiling transformation can be used to exploit C-slow retiming as proposed in [22]

In Figure 5.7a it is shown how the trivial execution order in a matrix multiplication kernel stalls the pipeline. Figure 5.7b displays how stalling is prevented by using C-

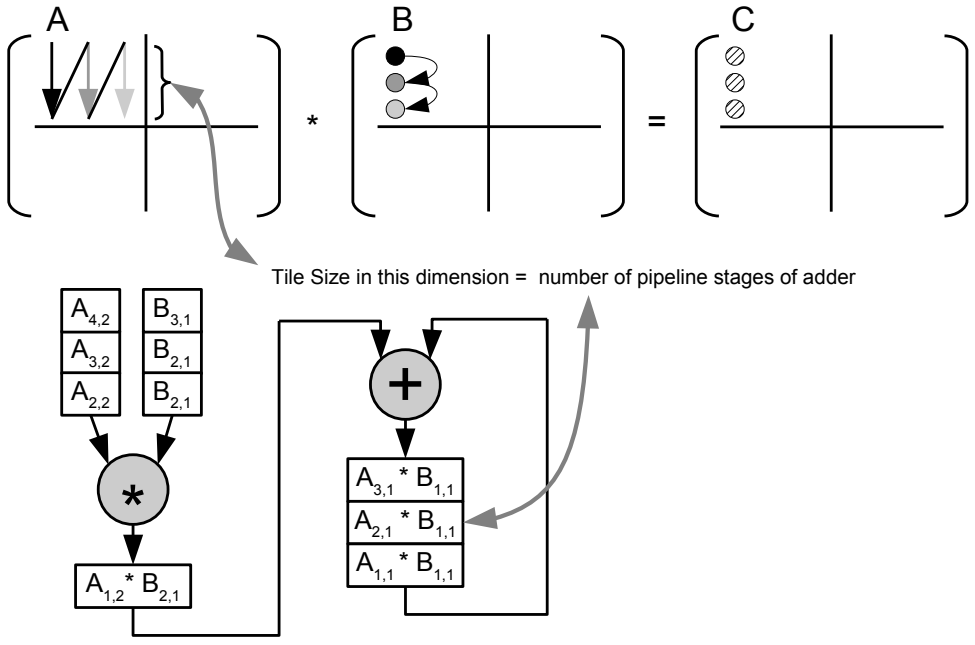
Slow retiming. In this example 3 inner-products are, pipeline-parallel, computed. The pipeline of the adder is used to store the 3 partial inner-products. For this to work the 3 streams of data, one for each inner-product need to be interleaved. This can be done using tiling.

Tiling divides the computation in uniform partitions, each operating on segments of the original dataset(s). There is both an internal execution order on the tile-level and an execution order on the tiles. These orders must, both, respect the data dependences.

Figure 5.7b shows the effect of tiling from a data flow perspective. The data partition of A is streamed in column-major order. This allows to interleave the data for independent inner-products in one stream. The number of pipeline stages of the adder determine the number of independent inner products that can be processed using pipelined parallelism. This can be controlled by the tile-size in the column major direction of matrix A. This order should be chosen equal to the pipeline depth of the adder. The tile-size in the other direction can be arbitrarily chosen.



(a) The Original Matrix Multiplication Stalls the Pipeline



(b) Utilising The Pipeline 100% by C-slow Retiming (Enabled by Tiling)

6 Implementation

The previous chapter introduced the *DfeGraph*. This chapter goes in more detail especially regarding how this *DfeGraph* is generated from the Polyhedral Dependence Graph (PDG). It discusses the node types in more detail and how they are generated. Consequently this chapter will elaborate on how, while traversing a PDG, it is decided which node type to generate and how to extract the information from the PDG and insert the required information into the *DfeGraph*.

6.1 *DfeGraph* Generation Algorithm

There are two major parts of the algorithm for generating a *DfeGraph* given a PDG. The first step generates the control part of the *DfeGraph*, while the second step generates the data path part of the *DfeGraph*. Some information specifying the connections between the control and data path needs to be communicated between step 1 and step 2. This is done by means of a table that links nodes in the PDG to specific control signals. The control is typically based on a number of counters that form a counter-chain. The control signals are some logical expression in terms of the outputs of these counters.

6.1.1 Step 1: Counter-chain Generation

The algorithm to add the control part to the *DfeGraph* is depicted in Algorithm 6.1. First an Abstract Syntax Tree (AST) is created from the PDG using the Integer Set Library (ISL). Algorithm 6.2 is called by the Algorithm 6.1 to extract the control, both the counter-chain and additional control signals, from this AST. Note that Algorithm 6.2 is a recursive function.

By recursively traversing the AST, the for loop nodes are visited from the inner-most loop outwards. From each for loop a counter is generated the order in which they are generated makes it easy to chain them in a counter-chain.

For each if-statement an entry, consisting of the condition and the PDG node it relates to, is added to a table. For the construction of the table the traversal order is not of importance. The related PDG is the PDG that is mentioned in the if statements then clause. The table used to store the (condition, PDG node) pairs is called *control table* in Algorithm 6.2.

Figure 6.1b shows a C-style printout of the AST corresponding to the PDG shown in Figure 6.1a. Because of the recursive traversal of the AST the first counter generated

will be the one that implements iterator $c2$, then the one implementing $c1$ and last the one implementing $c0$. Note that the iterators in the AST have been renamed: they correspond to the iterators in the original code as follows: $c0 = i, c1 = j, c2 = k$.

The three counters are organised into a counter-chain by connecting the *wrap*-signal of a counter to the *enable*-signal of the next counter in the chain. The counter-chain generated from the AST in Figure 6.1b operates as follows: the $c2$ counter will increase with every *tick*; $c1$ will increase on every wrap-around of $c2$ and $c0$ will increase on every wrap-around of $c1$. The behaviour of the counter-chain will be equal to the nested for loop represented by the AST in Figure 6.1b.

Algorithm 6.1: Counter-chain generation

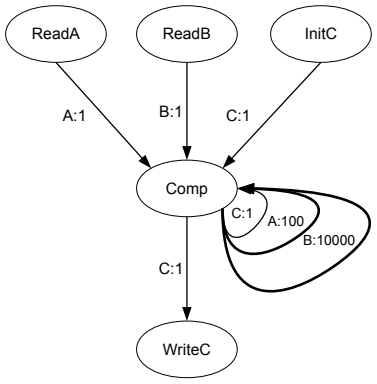
Input: a PDG
Output: a partial DfeGraph containing;
- a counter-chain;
- a table of <PDG node, condition> pairs referred to as **control table**;

forall *nodes in PDG* **do**
    add schedule of PDG node to a union of schedules;
Generate AST from the union of schedules;
extractControl(root node of AST);

Algorithm 6.2: extractControl(AST-node)

forall *child* $\in$ *childeren of AST-node* **do**
    extractControl(child)
if *AST node is for-loop* **then**
    generate counter based on loop bounds and stride;
    if *#counters in counter-chain > 0* **then**
        Connect enable of previous counter to wrap of this counter;
    return;
else if *AST node is if-statement* **then**
    Store a pair of <pdg node, condition> in **control table**;
    return;
else
    return;

The if-statements in the AST put restrictions on the executions of statements in their then clause. For example $ReadB(c1, c2)$ is only executed if $(c0 == 0)$. This *ReadB* statement will become an input in the Data Flow Engine (DFE) implementation. The $c0 == 0$ condition will be the enable signal for this input. So we need this condition to be added to the input node in the *DfeGraph*, when generating the data path, later. In order to save this information the condition and the node corresponding to the statement, is added to a the *control table*. For the statement *ReadB* the following entry will be added to the *control table*[1]:<*ReadB, c0==0*>



(a) The PDG Corresponding to the Matrix Multiplication Kernel

```
for (int c0 = 0; c0 <= 99; c0 += 1)
  for (int c1 = 0; c1 <= 99; c1 += 1) {
    InitC(c0, c1);
    for (int c2 = 0; c2 <= 99; c2 += 1) {
      if (c0 == 0)
        ReadB(c1, c2);
      if (c1 == 0)
        ReadA(c0, c2);
      Comp(c0, c1, c2);
    }
    WriteC(c0, c1);
  }
}
```

(b) AST corresponding to the PPN (printed as C code)

Figure 6.1: The PDG and the corresponding AST

[1]See Algorithm 6.2

Algorithm 6.3: Data path generation

Input: a PDG
Output: a*DfeGraph* Data path
forall *nodes in PDG traverse in ascending order of their ids* **do**
  switch *classify PDG node* **do**
    case *input node* **do**
      add input node to DfeGraph;
      if *PDG is in the map* ***control table*** **then**
        add condition from map **control table** to the enable of this DfeNode;
    case *output node* **do**
      add output node to DfeGraph;
      find input DfeStreams for this node and add as input;
      if *PDG is in the map* ***control table*** **then**
        add condition from map **control table** to the enable of this DfeNode;
    case *constant node* **do**
      add constant node to DfeGraph
    case *compute node* **do**
      forall *incoming dependences* **do**
        if *DfeStream corresponding to this dependences* $\notin$ *DfeGraph* **then**
          switch *Classify Input dependence* **do**
            case *dependence represents a memory* **do**
              add memory node in DfeGraph;
              inputStreams += output stream of memory
            case *dependence represents feedback path* **do**
              add Feedback stream to DfeGraph;
              inputStreams += feedback stream
      forall $i \in$*InputStreams* **do**
        forall $j \in$ *InputStreams* $> i$ **do**
          if *InputStreams share destinatio* **then**
            add Multiplexer node to DfeGraph;
            inputStreams += output stream of multiplexer;
      generate statement of compute node;

6.1.2 Data Path Generation

Algorithm 6.3 shows the procedure for generating the data path of the *DfeGraph*. The algorithm traverses the nodes of the graph and translates them in one or more nodes in the *DfeGraph*, see Figure 6.2. Dependences of size 1 are translated into *DfeStreams*. A *DfeNode* of type memory is generated if the size is larger than 1.

Each node in the PDG is labelled with a numeric id. These ids, in ascending order, provide a traversal order that respects the dependences in the PDG, except for feedback and reuse dependences. This traversal order makes sure that the input *DfeStreams* from earlier *DfeNodes* are available when generating the current *DfeNode*.

Considering the PDG shown in Figure 6.1a the traversal order will be:

1. ReadA
2. ReadB
3. InitC
4. Comp
5. WriteC

The PDG nodes *ReadA* and *ReadB* are translated into input nodes in the *DfeGraph*. The node $InitC$ represents the initialisation of variable C to "0" in the original code. This node is translated into a *DfeNode* of type constant. When adding a node to the *DfeGraph* the outgoing dependences of the corresponding PDG node are also generated and added to the *DfeGraph* as *DfeStreams*.

PDG node *Comp* represents the actual computation of the matrix multiplication kernel. This node is classified as a computational node by Algorithm 6.3. These computational nodes can have some challenging dependences associated with them, these dependence types can not occur in association with input, output or constant nodes. Before generating computational nodes the incoming dependences of that node need to be dealt with first.

Some of these dependences might not yet have a *DfeStream* associated with them in the *DfeGraph*. The dependences can either represent a feed-back path or a reuse-buffer. In case of a feed-back path a feedback stream is added as input, which source will be later connected. In case of reuse buffers the dependence is translated into a node of type memory in the *DfeGraph*. How the two cases are detected and additional details regarding their implementation we will be discussed in Section 6.2.2 and Section 6.2.3.

Once all the incoming dependences are dealt with, an additional step has to be taken. It may occur that multiple of the incoming dependences have the same destination (i.e. both dependences represent a write to one and the same memory element). Both writes will not occur simultanious therefore we can use a multiplexer to select between the two data streams. An extra node, of type multiplexer, is added to the *DfeGraph*. Additionally the appropriate control signal for the multiplexer needs to be derived.

The final step in translating a compute node in the PDG is to generate the actual computational statement represented by the node. In case of our example this would be the statement:

$C[i][j] = C[i][j] + A[i][k] * B[k][j]$

This statement is translated to a partial *DfeGraph* in which each operator will be a node. Streams are added to this partial graph to communicate partial results.

6.2 Implementation Details

This section gives some additional details regarding the generation of the *DfeGraph*. The translation of specific node types in the PDG to constructions in the *DfeGraph* is covered. Additionally we will explore how to detect feed-back paths and reuse buffers in the PDG and how to add them to the *DfeGraph*

6.2.1 Inputs and Outputs

Input and output nodes are to be implemented as inputs and outputs in the DFE kernel code. These are the streams that either receive data from or send data to the host. The current selection criterion for determining whether a node is an input is to check if it has no incoming dependences and its associate statement is a function call without arguments. For an input node, a node of type input is added to the *DfeGraph*. For each outgoing dependence of the node a *DfeStream* is generated and added to the *DfeGraph*. A node in the input PDG is classified as an output if it has no out-going dependences. For both input and output nodes it has to be checked whether they have an enable signal.

6.2.2 Reuse Buffer Detection and Generation

Earlier, when discussing the Polyhedral Process Network (PPN) concept, we have seen that the dependences actually model communication channels. The dependences specify, both the type and size of the channel. Some channels can, for example, be implemented as a FIFO while others would require a random access memory. How a FIFO, shift-register or memory is actually implemented may depend on the target. For example a channel of size 1 can simply be implemented as a register or a *DFEVar* since we are considering MaxJ code.

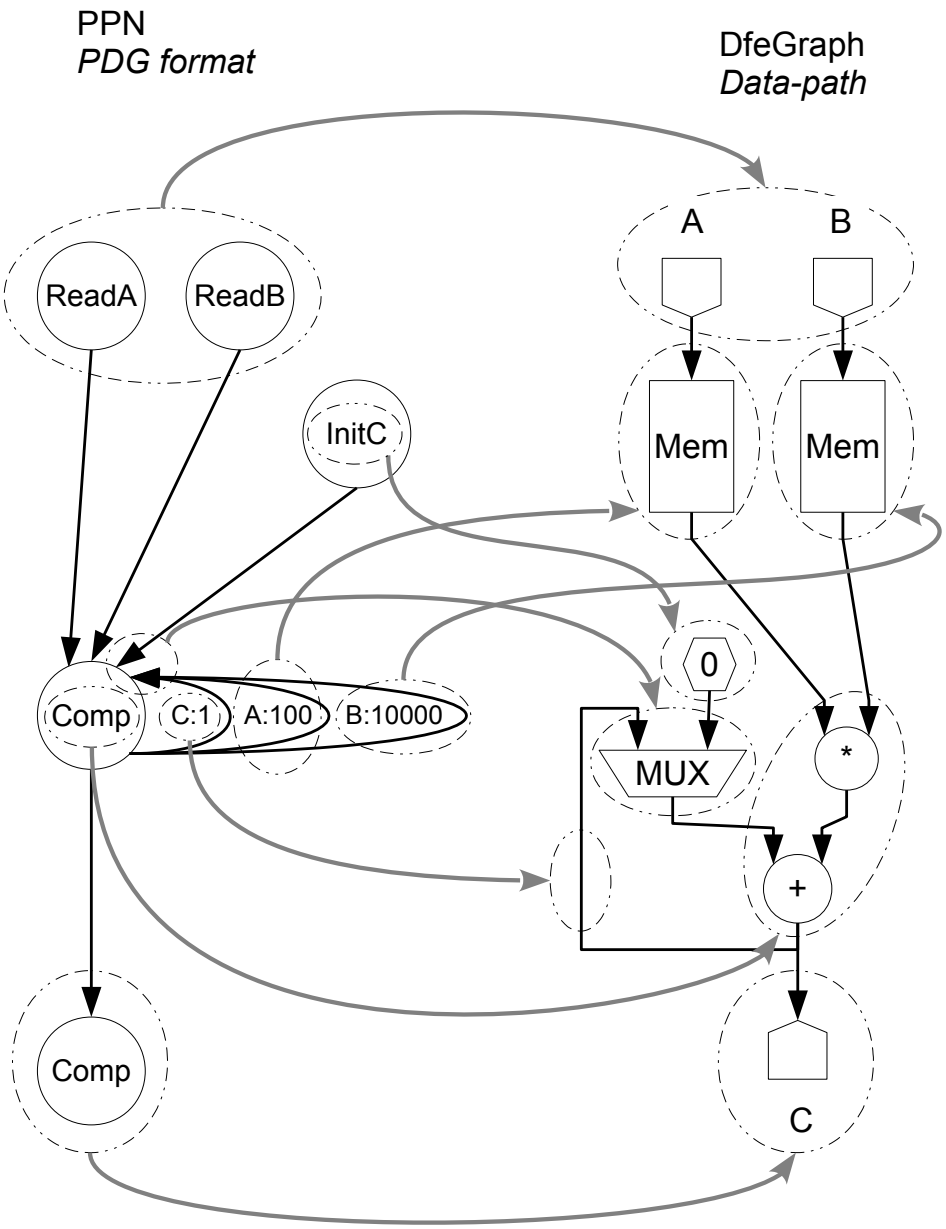


Figure 6.2: The DfeGraph data path generation from a PDG earlier shown as Figure 5.5

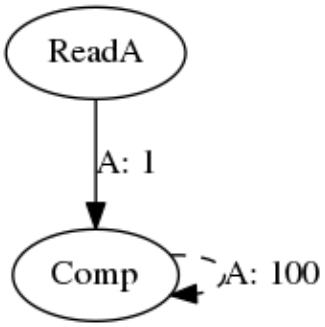


Figure 6.3: Part of the PDG shown in Figure 6.1a

One important issue to consider is how to detect the size and type of the memory to be generated. Let us consider the matrix multiplication example and in particular its PDG representation in Figure 6.1a. When visiting node *Comp* in the PDG we detect a dependence, which is not yet represented in the *DfeGraph*. Figure 6.3 shows the part of the PDG relevant for the current discussion. The dashed edge shows the dependence that lacks representation in the *DfeGraph*.

The dependence, represented by the dashed edge in Figure 6.3, is associated with the array A, has a size of 100 elements and as type a shift-register. This information, available in the PDG, defines the type and size of the memory that has to be implemented.

The next thing to consider is what functions as data input and data output to this shift-register. To get an answer to these questions a systematic method is implemented in the tool.

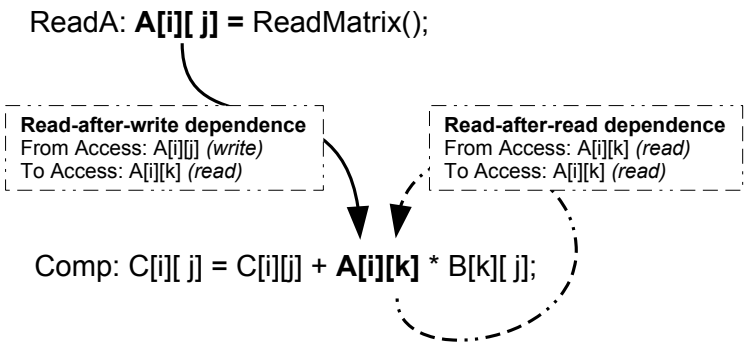


Figure 6.4: A detailed view of the partial PDG shown in 6.3

In order to explain this method, let us have a look at the properties of this dependence. Both the source and the destination of the dashed dependence in Figure 6.3 are one and the same node. Additionally both the *from access* and *to access* turnout to refer to the same access and the type of this access is a *read*, see Figure 6.4. So what the dependence actually models is a *Read-after-read* dependence.

The dependence models the data-reuse on matrix A. Considering the typical implementation of matrix multiplication if we buffer a single row of matrix A we can reuse it N (i.e. number of columns of matrix B) times. In this specific example we need to buffer one row of 100 elements. The implicit idea of the PDG (N.B. this is not how our tool implements this channel), is that the data element is used as input to the computation and simultaneously pushed into a shift-register to re-emerge after 100 clock-cycles at the other end exactly at the moment it is again required.

Because the dependence under consideration only models the *read-after-read* dependence it does not tell us where the data is originated. There must be a *read-after-write* relation with as *to access* the access of the *read-after-read* dependence. There is in fact such dependence in the PDG. According to the PDG the source of the data is the PDG-node *ReadA*. Figure 6.4 shows this clearly.

Note that the *to access* of the dependence between *ReadA* and *Comp* (the non-dashed) dependence in Figure 6.4 refers to the same access as the one on either side of the (dashed) *read-after-read* dependence. The dependence (i.e. communication channel) that functions as source of the data to our memory is the non-dashed edge.

Now that we have found the source of the data, there still remain some issues to consider before we could add the memory node to the *DfeGraph*. The first matter of concern comes from the observations that some times data should be read from the stream from *ReadA* while in other instances the data comes from the reuse buffer. In the target DFE implementation we should implement this behaviour.

The memory, to be implemented, should function as a reuse-buffer. A burst of data is written to the memory, the data is reused multiple times and then overwritten by a new burst, which will in turn be reused. Considering the running example we see that in this particular example one row is streamed into de buffer, the data is read multiple times and when no longer required replaced by the next row of the matrix. This is exactly how it works in the manual MaxJ code discussed earlier.

One alternative implementation considered is using an multiplexer and a *stream offset*. A *stream offset* is implemented as FIFO. Using the multiplexer we either provide the source stream to the FIFO or we provide the output of the FIFO to its own input. This however has one major and significant draw-back, since it forces the data to "move" on each *tick*. In other words, the stream can not be halted as in case of a feedback-path is sometimes necessary.

Another more flexible implementation is to use a memory. And in particular one with a so called read/write port. This memory enables to read and conditional write, simultaneously. One advantage is that we would not need a multiplexer. Because the memory type is a shift-register there is a rather obvious way to implement this in MaxJ. We can simply use the inner-most loop-counter to provide the address. The address will sequentially run trough the address space and then wrap-around and start reading again from the start. A new row should be read, when iterator j is 0. When j is 0 we enable the write signal and the data in the memory is overwritten with new data. Note that the enable (i.e. write) signal in implements the data multiplexing. The latter solution, which has been

implemented, is more elegant easier to generate and possibly cheaper in resources than the idea we have previously considered.

In the *DfeGraph* we add the memory as a node, this node is written to by a stream from node *ReadA* and read from one of the nodes the *comp* node in the PDG will be translated to. We place this node in between these two nodes.

Because the reuse buffer size is smaller than the original array, a new address mapping has to be generated. This new address mapping needs to be valid (i.e. the addresses should stay within the address range of the buffer and simultaneous live data should not be mapped to the same address). There is some work done to determine such mappings in the context of Polyhedral Compilation (PC) see[23]. This work explores valid modular mappings, in such mappings a modulo operator is used to keep the address within range of the reuse buffer. In our running example and in many other cases one of the iterators could actually function as a legal addressing function. From our manual MaxJ implementation we know that we can simple use iterator k as address for the reuse buffer on matrix A.

In the context of this thesis a simple algorithm that can find whether a simple expression in the iterators is a legal address function, is implemented. This algorithm is explained by the following example based on the matrix multiplication example.

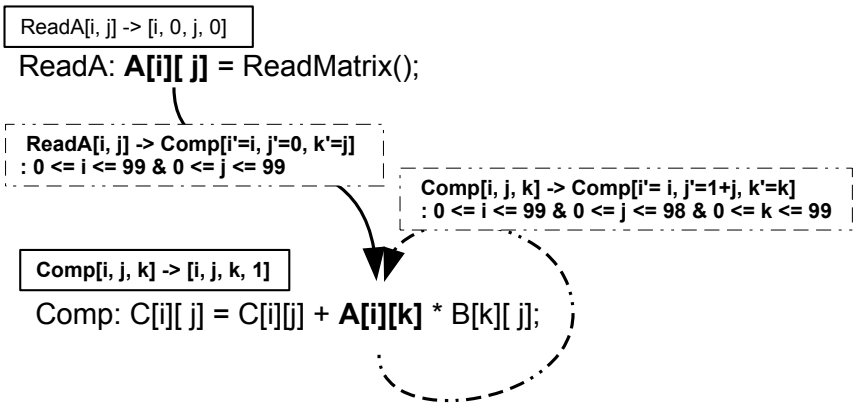


Figure 6.5: A detailed view of the dependences in the partial PDG

We again consider the reuse buffer on array A we have earlier discussed in this section. The method requires two inputs:

- The dependence relation of the dashed dependence in Figure 6.5.
- The schedule of node *Comp* also in Figure 6.5

Both the dependence relation and the schedule are Presburger relations.The dependence relation of the dashed dependence in Figure 6.5:

$$\begin{aligned} R = \{&Comp[i,j,k] \rightarrow Comp[i' = i, j' = 1 + j, k' = k] \\ &: 0 \leq i \leq 99 \wedge 0 \leq j \leq 98 \wedge 0 \leq k \leq 99\} \end{aligned} \tag{6.1}$$

The schedule of node *Comp* also in Figure 6.5:

$$S = \{Comp[i,j,k] \rightarrow [i,j,k,1]\} \tag{6.2}$$

By performing the following operations a Presburger relation can be attained that links the timestamps of dependent statement instance executions:

$$\begin{aligned} &D = S \circ (R \circ S^{-1}) \\ &D = \{[i,j,k,1] \rightarrow [i,j+1,k,1] : 0 \leq i \leq 99 \wedge 0 \leq j \leq 98 \wedge 0 \leq k \leq 99\} \end{aligned} \tag{6.3}$$

Next the *deltas* function is applied. This *deltas* functions produces a Presburger set containing the element-wise difference between the domain and the range of all elements in the relation D:

$$\begin{aligned} &\Delta = deltas(D) \\ &\Delta = [0,1,0,0] \end{aligned} \tag{6.4}$$

From a more applied perspective we have calculated the *reuse distance* for elements of array A. The reuse distance in this case is a single vector, this was anticipated, because a dependence labelled as *shift-register* by definition has a constant reuse distance.

The dimensions of the vector relate to the iterators in the original code, but also to the counters in the counter-chain that implement these iterators in the DFE implementation. The first dimension is actual a constant dimension we can ignore in our current discussion. The dimensions of the distance vectors relate to the counters, and constant as shown next:

$$\begin{aligned} &[c0, c1, c2, const] \\ &[0,1,0,0] \end{aligned} \tag{6.5}$$

The constant dimension is dropped in the remainder of the discussion.

The dependence distance vector models that an element of array A is read again after [0,1,0] *ticks* of the counter-chain. So between these dependence the counter $c0$ will wrap-around once. Therefore the counter $c0$ can be used as address for the reuse buffer containing A. This procedure works because the dependence is labeled a shift-register and because the vector has a "1" in one and only one dimension.

Repeating this procedure for the reuse buffer containing B results in the following reuse distance vector:

$$\begin{matrix}[c0, c1, c2] \\ [1, 0, 0]\end{matrix} \tag{6.6}$$

Note that the constant dimension has been dropped. The reuse distance in this instance is larger in lexicographical sense. To address the reuse buffer a combination of $c0$ and $c1$ is required. There are two options to generate the address:

1. a concatenation of $c1$ and $c0$ (i.e. $c1\#c0$)
2. a linearisation of $c1$ and $c0$ (i.e $c1 * max(c0) + c0$)

Option 1 is efficient in the sense that it does not require additional computation resources. However, if $max(c1)$ and $max(c0)$ are not both a power of two, one needs to allocate more memory than strictly necessary.

Option 2 does not require more memory than strictly necessary, but does require additional compute resources to compute the address.

6.2.3 Feedback Path Detection and Generation

The occurrence of feedback paths in the source PDG calls for special attention, because it requires a specific construct in the MaxJ code. As we have seen when discussing the manual MaxJ implementation in Section 5.2. In this example we have found that in MaxJ a feedback path is constructed by first declaring a *source-less stream*, and later, in the program text, connect the source. Therefore we need to find a mechanism to detect a feedback path in the source PDG. And a procedure to generate the appropriate MaxJ code.

In the original C code there is an obvious feedback path on each element of the C matrix. What we need to figure out now is how we can detect the feedback-path from the PDG. The feedback path is expressed by the dependence on array C that has both its source and destination at PDG node *Comp*. Inspecting this particular dependence we see that its *from access* is a write and its *to access* is a read. Using the fact that the PDG nodes are generated in the order of the data flow, we can detect when data flows "against the stream". A dependence with a *to node* with an *id* that is either lower or equal to the dependence's *from node* where the *from access* is a write and the *to access* is a read, is a feedback path.

6.2.4 Multiplexer Detection and Generation

When generating MaxJ code from a source PDG multiplexers have to be introduced, in some specific cases. Let us consider again the running example. considering the MaxJ

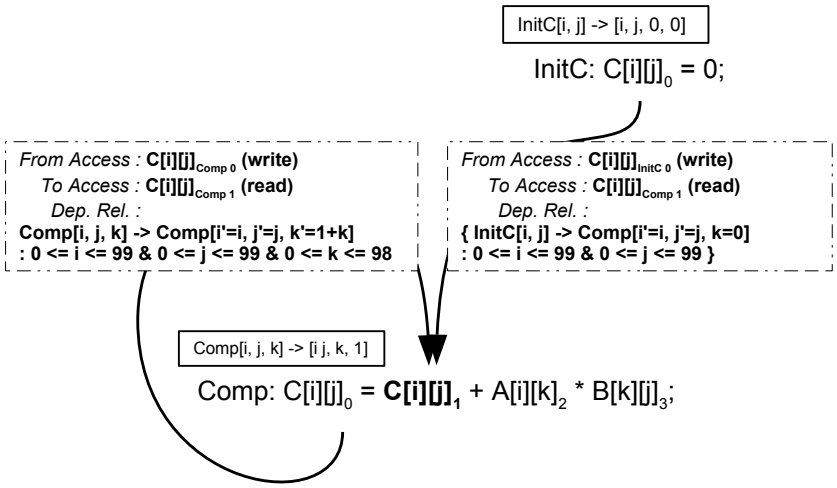


Figure 6.6: The partial PDG relavant for multiplexer detection

code we can see that there is a de-facto multiplexer implemented by the conditional statement on line 56 of Listing 5.2, this particular statement is repeated next:

```
DFEVar sum = k === 0 ? 0.0 : carriedSum;
```

Based on the value of iterator k either the constant *0.0* is assign to the stream *sum* or the value of *carriedSum*.

Let us see how we can deduce this multiplexer from the PDG. A multiplexer might be required when multiple dependences have the same *to access*. It is not generally the case that this should be a multiplexer because the dependences involved in the reuse buffer, we have discussed earlier, fulfil the exact same condition. So the additional condition should be that the *from access* of both dependences should be a write. Figure 6.6 shows the dependences and statements involved. Note that in this picture the accesses are numbered per statement, this is to distinguish them.

When generating the *DfeGraph* a new node is added for each multiplexer encountered. Adding the "conflicting" streams as its inputs and generating a new output stream. This is the trivial part. The next part, which is more interesting, is figuring out the selection signal of the multiplexer.

Given two conflicting dependences we can device a selection criterion that leads to a disjunction, in time, of the inputs streams. This selection criterion needs to decide when, which of the *input streams* should be forwarded to the output. The primary observation made here is that for a set of timestamps one input should be selected and for another set of timestamps the other input should be selected. We could use the constraints of one of the timestamp sets as selection criterion, because the two sets are disjunct by definition. In many cases this leads to a needlessly complicated selection criterium. Therefore an

algorithm that is able to find simpler selection criteria, saving on hardware and reducing complexity, is implemented in the tool developed as part of this thesis.

The algorithm is based on the observation that if the two sets of timestamps are disjunct in one of the dimensions, this can be used to find a simple selection criterium for this multiplexer. The Integer Set Library (ISL) can be used to test whether the two sets are disjunct in one dimension. The concept is rather trivial, we simply project both sets to one of the dimensions of the timestamp-space and check whether these projections are disjunct. If they are in fact disjunct we can uses the constraints on this specific dimension to determine a selection criterium.

In the matrix multiplication example we have been using in this document, there is one occurrence of a multiplexer as earlier mentioned. The dependences involved are:

$$\begin{aligned} D_1 &= Init[i,j] \rightarrow comp[i' = i, j' = j, k = 0] \\ D_2 &= Comp[i,j,k] \rightarrow Comp[i' = i, j' = j, k' = k+1] \end{aligned} \tag{6.7}$$

Additionally we will need the schedules of both source nodes. We use these schedules to map the dependences to a timestamp. These tell us the source of data at each timestamp.

$$\begin{aligned} S_{InitC} &= InitC[i,j] \rightarrow [i,j,0] \\ S_{Comp} &= Comp[i,j,k] \rightarrow [i,j,k] \end{aligned} \tag{6.8}$$

$$\begin{aligned} S_{InitC} \circ D_1 &= InitC[i,j] \rightarrow [i,j,0] \\ S_{Comp} \circ D_2 &= Comp[i,j,k] \rightarrow [i,j,k+1] \end{aligned} \tag{6.9}$$

The range of the two Presburger relations in Equation 6.9 are disjunct. From this we can conclude that $InitC$ is the source when $k = 0$ and $Comp$ is the source when $k > 0$. Both these conditions can be used as selection signal for the multiplexer.

6.2.5 Constants

In the running example the *InitC* node's sole purpose is to assign the constant *0.0* to the accumulation on *C[i][j]*. In the hand coded MaxJ implementation Listing 5.2 this constant is explicitly coded in the multiplexer, see line 56 or Listing 5.2. Because it is slightly easier to generate, I decided to simply generate a stream that is assigned the constant value.

In the PDG a node representing a constant has no incoming dependences and the *statement* represents an assignment of a constant value to a variable. The conditions are relatively simple to check.

6.2.6 Computation

A computational node is a node that performs actual data manipulation on an input stream, in the current version of the tool I only consider addition and multiplication, but this can be extended to support a wide variety of operators and mathematical functions. In case of a computational node we generate the actual statement of the node. This statement could exist of a tree of function calls, if there are nested function calls or multiple operations involved in the statement. Although not particularly complicated, the computational statement in the example actually is a tree of function calls. This tree is implemented in space on the surface of the chip, as part of the data path.

7 Conclusions

This chapter contains the conclusions drawn from this thesis work. The chapter starts by addressing the research questions this thesis initially set out from. Each research question is addressed individually. Additionally some insights regarding the proposed tool-flow and its future development directions are explored.

7.1 Addressing the Research Questions

The research questions formulated in Section 1.2.1 were:

- Is it possible to generate a Data Flow Engine (DFE) implementation from a Polyhedral Process Network (PPN)?
- Can such a mapping be efficient, in terms of the theoretical achievable performance?
- Can the concepts from Polyhedral Compilation (PC) be applied to perform transformations on the Polyhedral Process Network (PPN) abstraction that will improve the efficiency of the generated Data Flow Engine (DFE) implementation?
- What are performance improving transformations when a Data Flow Engine (DFE) implementation is considered as our target?

Is it possible to generate a Data Flow Engine (DFE) implementation from a Polyhedral Process Network (PPN)?

In this thesis a tool has been developed that is able to partially generate the MaxJ kernel code, from a PPN[1] for some simple kernels. The matrix multiplication kernel has been primarily used to develop and test the tool. The tool is by no means complete, however solutions are proposed and implemented to solve a number of challenges involved in generating a DFE implementation from a PDG.

Generating a DFE implementation from a PDG has proven to be harder than initially anticipated. The complexity of the translation is mainly because a PDG is an abstract representation while the MaxJ code defines a specific implementation of the computation. Additionally there is a significant amount of information that is not directly available from the PDG, but has to be deduced from it. Finally, generating the MaxJ code can not be done by a trivial sequential traversal of the PDG.

[1] The actual PPN implementation used as input for the tools is a Polyhedral Dependence Graph (PDG)

As a solution for the challenges formulated an intermediate abstraction were proposed. This abstraction is tailored to the structure of the target MaxJ code. The abstraction enables relatively easy generation of MaxJ code. Additionally this representation can be constructed iteratively. The representation is the *DfeGraph* discussed in Section 5.4. This abstraction was successful in solving the challenges.

Despite this intermediate abstraction the translation form PDG to MaxJ code has proven, by experience, to be challenging. The conclusion drawn here is that the PDG is not an entirely suitable abstraction to easily generate a DFE implementation from. A possible solution would be to develop a PC based abstraction that by design corresponds naturally to a DFE implementations. This latter although an interesting endeavour was unfortunately not possible in the span of this thesis.

In conclusion, PC has potential in improving programmability of a DFEs. However, it requires the development of a polyhedral representation that is more specifically tuned for data flow computing in general and DFE implementation in particular. The concept of a PPN is in theory a helpful idea but needs to be extended.

Can such a mapping be efficient, in terms of the theoretical achievable performance?

The developed tool, is able to generate sequential code for the matrix multiplication kernel that is similar to the hand-coded version. However, this hand-coded version is not optimal. In order to enable a more optimal implementation the tool should be equipped with the notion of parallelism, in the form of parallel data path replication. The PDG does not provide such notion and models a more or less sequential execution. The form of parallelism available in the PDG is that of pipelined-parallelism of the processes (i.e. the processes form a pipeline). This parallelism is exploited by the tool.

Can the concepts from Polyhedral Compilation (PC) be applied to perform transformations on the Polyhedral Process Network (PPN) abstraction that will improve the efficiency of the generated Data Flow Engine (DFE) implementation?

The PDG abstraction does provide information regarding the volume of communication between processes. This can be very helpful in analysing the theoretical performance of an implementation and the effects on the latter when applying transformations.

Because of the complexity involved in the tool the question regarding the actual effect of transformations can not be backed by an actual implementation. However it is theoretically possible to perform transformations and re-calculate the communication volumes and edge types. This would necessitate some extensions of the PDG and Integer Set Analysis (ISA) library implementation, which was not feasible in the span of this thesis.

We have however discussed some useful transformations, that are able to perform a number of optimisations that a DFE application developer currently has to do by hand. An example is to rearrange the order of computations to prevent a pipelined operator, involving feedback, to stall the data-path. This so called C-Slow retiming can be performed using a tiling transformation. Another example is to minimise buffer sizes between pro-

cesses in the PDG by reordering the production and consumption of the data elements to be buffered. If the time between production and consumption is reduced one can suffice with smaller buffers.

What are performance improving transformations when a Data Flow Engine (DFE) implementation is considered as our target?

Considering the envisioned target, a DFE implementation, we have concluded that the most promising transformation is tiling. Tiling has multiple performance enhancing properties. As discussed, when addressing the previous research question, a tiling transformation can be used to apply C-Slow retiming. Tiling can also be used to reduce the pipeline buffers. By means of tilling the computation can be split up in smaller parts that allow the data to be kept and reused on chip. In conclusion, we state that a tiling transformation is the most relevant in the context of data flow computing.

Another useful transformation is that of data-reordering in the host code. In the example, matrix multiplication, we have considered through out this thesis that, it is beneficial to stream matrix B in column major order. In order to do this the matrix needs, to be stored in the host memory in this order. Transformations can be used to automatically introduce data-reordering, which is beneficial in case of the mentioned example and in many other applications.

7.2 Future Work

By analysing the development process of the tool some conclusions are drawn regarding the applicability of PC in general and PPNs in particular to semi-automated generation of a DFE implementation.

Based on the discussion of the research questions in the previous section the conclusion is drawn that the application of PC in the context of data flow programming has potential, but is also confronted with some serious challenges. This thesis has proven that although not trivial it is possible to generate a DFE implementation. However in order to develop a production level tool or tool-flow a number of issues have to be resolved. The development of a PC abstraction tailored to data flow computing.

The tool-flow presented in this work has an experimental nature. It is an initial implementation of the ideas formulated in this thesis, definitely not a complete and optimal solution. Implementing this tool-flow was a practical feasibility study and has been helpful to gain valuable insights. Based on these insights I have drawn some additional conclusions, specific to the tool-flow developed and possible future directions.

From the development of the tool and my experience with data flow computing according to the Maxeler DFE philosophy, I conclude that a tool that fully-automatically generates a DFE implementation from imperative code is complex. Implementing such tool is not the eventual aim of this work. In the initial exploration of the problem, partially documented in Section 1.1, I already concluded that there exist a fundamental gap between control flow, often implemented by imperative code, and data flow computing, that is not easily bridged.

The attempt of simplifying data flow computing by automatic translation of an imperative algorithm has one, at least in my conclusion, significant problem. It hides from the developer what is actually going on, so they are left without control. This is something, we might not want to do. It is no harm exposing to the developer to the data flow paradigm and it will give the developer control over the implementation.

What is of interest, is raising the level of abstraction of data flow computing, hiding implementation details, without taking away control from the developer. An interesting future direction would be to integrate PC concepts in the MaxJ code so the developer has, for example, less to worry about generating efficiently orchestrated address streams for memories. Additionally PC and in particular PPNs, can be used as analysis tool to explore the affect of specific transformations. Code transformation can also be used to automatically introduce data-reordering in the host code.

In conclusion, the most promising future direction of this work is not to pursuit a method for translating imperative code to a DFE implementation. Instead an approach that utilises the concepts of Polyhedral Compilation (PC) to automate specific parts of DFE application development seems to be much more fruitful.

Bibliography

[1] N. Voss, T. Becker, O. Mencer, and G. Gaydadjiev, "Rapid development of gzip with maxj," in *International Symposium on Applied Reconfigurable Computing*. Springer, 2017, pp. 60–71.

[2] Maxeler, "Maxeler." [Online]. Available: http://maxeler.com

[3] S. Verdoolaege, "Polyhedral process networks," 2010. [Online]. Available: https://www.researchgate.net/publication/226060522_Polyhedral_Process_Networks

[4] ——, "Presburger formulas and polyhedral compilation," KU Leuven, Tech. Rep., 2016.

[5] ——, "Integer set library: Manual," INRIA, Tech. Rep., 2018.

[6] A. Gallini, ""affine function." from mathworld–a wolfram web resource, created by eric w. weisstein." [Online]. Available: http://mathworld.wolfram.com/AffineFunction.html

[7] U. Bondhugula, "Effective Automatic Parallelization ande Locality Optimization using the Polyhedral Model," Ph.D. dissertation, The Ohio State University, August 2008.

[8] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, "Schedule trees," in *IMPACT-4th Workshop on Polyhedral Compilation Techniques, associated with HiPEAC*. ACM, 2014.

[9] "Polly Labs, "Polly labs tutorials." [Online]. Available: http://playground.pollylabs.org

[10] P. Feautrier, "Some efficient solutions to the affine scheduling problem. i. one-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–347, Oct 1992. [Online]. Available: https://doi.org/10.1007/BF01407835

[11] ——, "Some efficient solutions to the affine scheduling problem. part ii. multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, Dec 1992. [Online]. Available: https://doi.org/10.1007/BF01379404

[12] S. Verdoolaege and G. Janssens, "Scheduling for PPCG," 2017. [Online]. Available: https://lirias.kuleuven.be/retrieve/458776

[13] P. Evripidou and C. Kyriacou, "Data-flow vs control-flow for extreme level computing," in *2013 Data-Flow Execution Models for Extreme Scale Computing*, Sept 2013, pp. 9–13.

[14] F. de Dinechin and B. Pasca, *High-Performance Computing using FPGAs*. Springer, 2013, ch. Reconfigurable Arithmetic for High Performance Computing, pp. 631–664.

[15] ——, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[16] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 2012, pp. 1212–1221.

[17] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 89–108. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869469

[18] R. S. Nikhil and Arvind, "What is bluespec?" *SIGDA Newsl.*, vol. 39, no. 1, pp. 1–1, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1862876.1862877

[19] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[20] T. Grosser, H. Zheng, R. A, A. Simbürger, A. Grösslinger, and L.-N. Pouchet, "Polly - polyhedral optimization in llvm," in *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011.

[21] A. Balevic and B. Kienhuis, "A data parallel view on polyhedral process networks," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2011, pp. 38–47.

[22] C. Alias, B. Pasca, and A. Plesco, "Automatic generation of fpga-specific pipelined accelerators," in *Reconfigurable Computing: Architectures, Tools and Applications*, A. Koch, R. Krishnamurthy, J. McAllister, R. Woods, and T. El-Ghazawi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 53–66.

[23] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1242–1257, 2005.

[24] A. Darte, Y. Robert, and F. Vivien, *Scheduling and Automatic Parallelization*, 1st ed. Secaucus, NJ, USA: Birkhauser Boston, Inc., 2000.

[25] A. Schrijver, *Theory of Linear and Integer Programming*. New York, NY, USA: John Wiley & Sons, Inc., 1986.

Dependence Analysis in the Integer Set Library

A

This chapter gives an example of how polyhedral compilation works in the Integer Set Library (ISL)-approach. Let us consider a trivial but practical kernel that computes the inner product of two vectors, a C-code example is given in Listing A.1. The instance set of both statements S and T can be expressed as a union of Presburger sets see Equation A.1.

```
S: prod = 0;
for (int i = 0; i < N; ++i)
  T: prod += A[i] * B[i];
```

Listing A.1: Inner Product Kernel

$$[N] \rightarrow \{S[]; T[i] : 0 \leq i < N\} \tag{A.1}$$

A.1 Access Relations

In the ISL-approach to Polyhedral Compilation (PC) Presburger relations are used to express access relations. In many cases the Presburger relations can model the access relations exactly, however in order to improve the applicability of PC to less regular code the ISL allows the access relations to be approximated, if they can not be expressed by Presburger formulas. A trivial example of an access function that can not be expressed as a Presburger relation is one that involves multiplication of variables. In the ISL three access relations are considered, hence that the may-write and must-write overlap.

May-read Access Relation gives an over-approximation of the read accesses.

May-write Access Relation gives an over-approximation of the write accesses.

Must-write Access Relation gives the maximal sub-set of the may-write relation that can be defined as a Presburger relation.

One may notice that there is no must-read access relation. This is because it is not require for performing the analyses steps.

In many case the access relations can be defined exactly, in this situation the relations are exact as well. In this example and in fact in this thesis as a whole we focus on such code, therefore we can suffice with one read access relation and one write access relation.

considering the code in Listing A.1 the following access relations are deduced:

Write Access Relation:

$$[N] \rightarrow \{S[] \rightarrow prod[]; T[i] \rightarrow prod[] : 0 \leq i < N\} \tag{A.2}$$

Read Access Relation:

$$[N] \rightarrow \{T[i] \rightarrow B[i] : 0 \leq i < N; T[i] \rightarrow A[i] : 0 \leq i < N; T[i] \rightarrow prod[] : 0 \leq i < N\} \tag{A.3}$$

A.2 Dependence Relations

The access relations, discussed in the previous section, are a relations between the statements instances and array elements accessed by that statement instance. Using these access relations we can determine the dependence relations. These dependence relations express which statement instances depend on each other. Three types of dependences can be distinguished, technically a fourth although one can argue whether this should strictly be considered a dependence.

Recall that dependence relations are distinguished based on there type:

Read after Write maps a statement instance i to a statement instance j if j is executed after i and if it reads from a data element that is written by i.

Write after Read maps a statement instance i to a statement instance j if j is executed after i and if it writes to a data element that is read by i.

Write after Write maps a statement instance i to a statement instance j if j is executed after i and if it writes to a data element that is written by i.

Read after Read maps a statement instance i to a statement j if j is executed after i and if it reads a data element that is (also) read by i.
This is not actually a dependence because it implies no specific order of execution of i and j.

For the example the dependence relations are as follows:

Read-after-Write dependence relation:

$$[N] -> \{S[] -> T[i] : 0 <= i < N; T[i] -> T[i'] : i >= 0 \wedge i < i' < N\} \tag{A.4}$$

Write-after-Read dependence relation:

$$[N] -> \{T[i] -> T[i'] : i >= 0 \wedge i < i' < N\} \tag{A.5}$$

Write-after-Write dependence relation:

$$[N] -> \{S[] -> T[i] : 0 <= i < N; T[i] -> T[i'] : i >= 0 \wedge i < i' < N\} \tag{A.6}$$

A.3 Filling our Toolbox

A number of tools are to be understood to perform dependence analysis using the ISL. This section is devoted to giving a concise explanation of the require tools.

The dependence relations are derived using the schedule and the access relations. It is instructive to see how this actually can be done. In order to do so we have to consider some additional concepts.

A.3.1 Composition

The first tool we need is an operation on Presburger relations called composition. The operation is in my opinion best explained by giving its definition and an example from [4].

Given two Presburger relations A and B the composition is defined as follows.

$$B \circ A = \{i \to j : \exists k : i \to k \in A \wedge k \to j \in B\} \tag{A.7}$$

Example Consider the relations:

$$A = \{B[6] \to A[2, 8, 1]; B[6] \to B[5]\} \tag{A.8}$$

$$B = \{A[2, 8, 1] \to B[5]; A[2, 8, 1] \to B[6]; B[5] \to B[5]\} \tag{A.9}$$

Their composition is:

$$B \circ A = \{B[6] \to B[6]; B[6] \to B[5]\} \tag{A.10}$$

A.3.2 The Inverse of a Presburger Relation

Secondly we need to know what the inverse of a Presburger relation is, this is even more trivial than a composition and rather self-explanatory. I think the definition should suffice:

$$R^{-1} = j \to i : i \to j \in R \tag{A.11}$$

A.3.3 The intersection of Presbuger Relations

Another rather obvious concept we need to understand is what a intersection of two Presburger relations means:

The intersection $A \cap B$ of two binary relations A and B contains the pairs of elements that are contained in both A and B[4].

A.3.4 Lexicographical Order

The lexicographical order is more or less a natural order on instantiations of multi-dimensional objects. One example is the method to decide order in a number system. Formally in a number system the most-significant digit in which two numbers differ, while less-significant digits are equal, determines the order. The order implied order on time notated as *hours:minutes:seconds* is also an example of the lexicographical order. This concept can clearly be extended to to vectors or integer tuples. The formal definition of the lexicographical smaller than relations is as follows:

"Given two vectors a and b of equal length, a is said to be lexicographically smaller than b if it is not equal to b and if it is smaller in the first position in which it differs from b. If the shared length of the two vectors is n, then this condition can be expressed as the Presburger formula"[4]

$$\bigvee_{i:1\leq i<n} \Big(\big(\bigwedge_{j:1\leq j<i} a_j = b_j \big) \wedge a_i < b_i \Big) \tag{A.12}$$

A.3.5 The Order Relation

The last thing we need to perform data-analysis is the execution order relation. This relation is the lexicographically smaller relation between the schedule and itself. The lexicographical order is a concept to compare vectors. The normal comparison operators are usually considered to mean a component-wise comparison. However, in many cases one wants to compare on a vector-basis (e.g. decide which one of the vectors is the greater one). It is quite straightforward that such ordering can also be applied to sets of named integer tuples and named integer tuple templates. Applying this ordering on the schedule and itself gives us the ordering relation. This relation contains all the pairs of statements for which the timestamp assigned by the schedule is lexicographically smaller than the time stamp of the second one. One obvious observation is that this relation is quite convoluted, this is one of the reasons for developing new schedule abstractions.

$$\begin{aligned}
[N, M] \rightarrow \{ & T[i,j] \rightarrow S[i'] : i' > i; \\
& S[i] \rightarrow S[i'] : i' > i; \\
& S[i] \rightarrow T[i', j] : i' > i; \\
& S[i] \rightarrow T[i' = i, j]; \\
& T[i,j] \rightarrow T[i', j'] : i' > i; \\
& T[i,j] \rightarrow T[i' = i, j'] : j' > j \}
\end{aligned} \tag{A.13}$$

$$<_S = S \prec S \tag{A.14}$$

A.4 Applying the Tools

Now we have discussed all the tools to perform dependence analysis. Constructing the dependence relation can be done using some simple though elegant statement. The read-after-write dependence relation is given by:

$$(R^{-1} \circ W) \cap \ <_S \tag{A.15}$$

Dissecting this statement we see that $R^{-1} \circ W$ results in a relation that contains all pairs of writes and reads that address the same data element. To better clarify the latter we recall the implementation and function of the access relations.

An access relations in essence relate statement instances to array indices (i.e. a particular data element). The statement instances form the domain of this relation while the array indices form the range of this relation.

The composition operator constructs a new relation from members of the domain of the relation to the right of the operator to members of the range of the relation left of the operator.

The inverse operation simply swaps the domain and the range of a relation. Therefore R^{-1} is a relation from (i.e. the domain) array indices to (i.e. the range) statement instances. Therefore $R^{-1} \circ W$ produces in a new relation from statement instances from the domain W to statement instances from the range of R^{-1} for which the range of W matches the domain of R^{-1}.

This relation contains all pairs of writes and read statement instance that address the same address regardless of the order in which they occur. This ordering is introduced by intersecting the relation with the order relation $<_S$. The order relation contains all pairs of statement instance where the first is executed before the second. Therefore the result of the intersection is a relation that contains all pairs of statement instance where the first is a write, the second is a read, and the first is executed before the second according to the order set (or in fact the schedule). The former is the very definition of the Read-after-write dependence relation.

The other types of dependence relations can be constructed similarly. Using the following statements.

Write-after-read dependence relation:

$$(W^{-1} \circ R) \cap \ <_S \tag{A.16}$$

Write-after-write dependence relation:

$$(W^{-1} \circ W) \cap \ <_S \tag{A.17}$$

Feautrier's Scheduler

B

Freautrier's algorithm or simply, the Feautrier algorithm, was firsts published in two publications[10, 11]. It is also covered by[12]. An example driven concise introduction is given by [24, Section 5.6], the remainder of this section borrows extensively from the latter.

First we have to consider some notation we have not cover before. It is an alternative way to describe the dependence between statement instances. The notation is shown in Equation B.2. In order to clarify the notation we first have to considerer an alternative notation for dependences relations. In Equation 2.9 we have seen how a polyhedron can be used to specify a dependence between statement instances. There is however an alternative notation, That gives a minimal form of the dependence relation.

As shown earlier in Equation 2.9 a dependence relation can be specifies using a polyhedron that maps the source statement instance to a destination statement instance, if and only if there is a dependence between the two statement instances. The alternative notation uses a polyhedron $\mathcal{P}_e$ that describes the destination statement instances and an affine function h_e that maps these destination statement instances to source statement instances. This is probably best clarified by mathematical notation shown in Equation B.1. In this equation $D_{\delta(e)}$ is the instance set of the destination of dependence e and $D_{\sigma(e)}$

$$\begin{aligned} y \in \mathcal{P}_e \implies y \in \mathcal{D}_{\delta(e)} \wedge h_e(y) \in \mathcal{D}_{\sigma(e)} \\ < x, y > \in \mathcal{R}_e \equiv (x = h_e(y) \wedge y \in \mathcal{P}_e) \end{aligned} \tag{B.1}$$

Bearing in mind the previous, the dependence between a statement S statement T can be described as shown in Equation B.2. The extra argument N in $h_e(j, N)$ describes the structural parameters of the input code.

$$j \in \mathcal{D}_e \implies S(h_e(j, N)) \to T(j) \tag{B.2}$$

The Feautrier scheduler uses a known lemma applying to polyhedra. This lemma will be introduced here without proof, the proof for the lemma can be found in[25].

Lemma 1. *Let $\mathcal{D}$ be an nonempty polyhedron defined by p inequalities: $a_k x + b_k \geq 0$, for any $k \in \{1, \ldots, p\}$. An affine from Φ is nonnegative over $\mathcal{D}$ if and only if it is a nonnegative affine combination of the affine forms used to define $\mathcal{D}$:*

$$\Phi(x) \equiv \lambda_0 + \sum_{k=1}^{p} \lambda_k (a_k x + b_k), \forall \in [0, p] \lambda_k \geq 0$$

A schedule is a $\Theta(S, j, NN$ is a nonnegative affine form defined on a polyhedron $\mathcal{D}_S$. This instance set or statement domain can be described as follows:

$$D_S = \{x | \forall i \in [1, p_S], A_{S,i} x + B_{S,i} N + c_{S,i}\} \tag{B.3}$$

The matrix $B_{S,i}$ and the vector N are used to incorporate the structural parameters into this description of the instance set. Although the notation is slightly different it is just another way of specifying a instance set. Since Θ is a nonnegative affine function defined on $\mathcal{D}_s$ it is a nonnegative affine combination of the affine forms used to define $\mathcal{D}_S$. According to Lemma 1 there exist some nonnegative values $\mu_{S,0}, \ldots, \mu_{S,p_S}$ such that:

$$\Theta(S, j, N) \equiv \mu_{S,0} + \sum_{i=1}^{p_S} \mu_{S,i} (A_{S,i} x + B_{S,i} N + c_{S,i}) \tag{B.4}$$

Considering a dependence from a statement S to a statement T the following constraint must hold:

$$j \in \mathcal{D}_e \implies \Theta(S, h_e(j, N), N) + 1 \leq \Theta(T, j, N) \tag{B.5}$$

The latter can be rewritten as an affine function that is nonnegative over a polyhedron because the schedules are affine functions:

$$j \in \mathcal{D}_e \implies \Theta(T, j, N) - \Theta(S, h_e(J, N), N) - 1 \geq 0 \tag{B.6}$$

Defining D_e as follows:

$$D_e = \{x | \forall i \in [1, p_e], A_{e,i} x + B_{e,i} N + c_{e,i}\} \tag{B.7}$$

Then according to Lemma 1 there exist nonnegative values $\lambda_{e,0}, \ldots, \lambda_{e,p_e}$ such that:

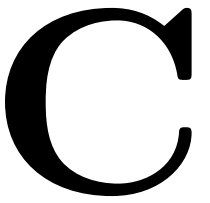
$$\Theta(T, j, N) - \Theta(S, h_e(J, N), N) - 1 \equiv \lambda_{S,0} + \sum_{i=1}^{p_S} \lambda_{S,i} (A_{e,i} j + B_{e,i} N + c_{e,i}) \tag{B.8}$$

The left-hand side of this equation can be rewritten using the earlier results:

$$\begin{aligned} &\mu_{T,0} + \sum_{i=1}^{p_T} \mu_{T,i}(A_{T,i}j + B_{T,i}N + c_{T,i}) \\ &\mu_{S,0} + \sum_{i=1}^{p_S} \mu_{S,i}(A_{S,i}j + B_{S,i}N + c_{S,i}) - 1 \\ &\equiv \lambda_{S,0} + \sum_{i=1}^{p_S} \lambda_{S,i}(A_{e,i}j + B_{e,i}N + c_{e,i}) \end{aligned} \tag{B.9}$$

From this equality a system of linear equations and inequalities can be constructed which can be solved by a solver for linear systems.

Classic Loop Transformations



Loop Reversal

Original Code:

```
for (int i=0; i<N; i++)
  S(i);
```

Instance set: $[N] \rightarrow \{S[i] : 0 \leq i < N\}$
Original Schedule: $\{S[i] \rightarrow [i]\}$
Transformation: $\{[i] \rightarrow [(N-1) - i]\}$

Transformed Code:

```
for (int i=N-1; i>=0; i--)
  S(i);
```

Loop Fusion

Original Code:

```
for (int i=0; i<N; i++)
  S(i);

for (int i=0; i<N; i++)
  T(i);
```

Instance Set: $[N] \rightarrow \{S[i] : 0 \leq i < N; T[i] : 0 \leq i < N\}$
Original Schedule: $\{S[i] \rightarrow [0, i]; T[i] \rightarrow [1, i]\}$
Transformation: $\{[0, i] \rightarrow [i, 0]; [1, i] \rightarrow [i, 1]\}$

Transformed Code:

```
for (int i=0; i<N; i++){
  S(i);
  T(i);
}
```

Loop Fission

Original Code:

```
for (int i=0; i<N; i++){
  S(i);
  T(i);
}
```

Instance Set: $[N] \rightarrow \{S[i] : 0 \leq i < N; T[i] : 0 \leq i < N\}$
Original Schedule: $\{S[i] \rightarrow [i, 0]; T[i] \rightarrow [i, 1]\}$
Transformation: $\{[i, 0] \rightarrow [0, i]; [i, 1] \rightarrow [1, i]\}$

Transformed Code:

```
for (int i=0; i<N; i++)
  S(i);

for (int i=0; i<N; i++)
  T(i);
```

Strip Mining

Original Code:

```
for (int i=0; i<N; i++)
  S(i);
```

Instance Set: $[N] \rightarrow \{S[i] : 0 \leq i < N\}$
Original Schedule: $\{S[i] \rightarrow [i]\}$
Transformation: $\{[i] \rightarrow [\lfloor i/4 \rfloor, i \bmod 4]\}$

Transformed Code:

```
for (int i=0; i<(N/4); i++)
  for (int j=0; i<4; j++)
    S(4*i+j);
```