POLYHEDRAL ANALYSIS OF STREAMING TASK-PARALLEL APPLICATIONS

A thesis submitted to The University of Manchester for the degree of Doctor of Philosophy in the Faculty of Science and Engineering

2021

Nuno Miguel Nobre

Department of Computer Science
Contents

Abstract 7
Declaration 8
Copyright 9
Acknowledgements 10

1 Introduction 11
   1.1 The Challenge of Parallel Programming . . . . . . . . . . . . . . . . 12
   1.2 Towards a Solution: The Dataflow Paradigm . . . . . . . . . . . . . . 13
   1.3 Tuning Streaming Dataflow: The Polyhedral Model . . . . . . . . . . 15
   1.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
   1.5 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

2 Context and Related Work 19
   2.1 A Zoo of Data-Driven Models of Computation . . . . . . . . . . . . . . 19
   2.2 Hardware Implementations . . . . . . . . . . . . . . . . . . . . . . 24
   2.3 Control-Driven Streaming Dataflow . . . . . . . . . . . . . . . . . . 25
   2.4 Static Analysis of Data-Driven Programs . . . . . . . . . . . . . . . . 26
      2.4.1 Polyhedral Optimisations . . . . . . . . . . . . . . . . . . . . 26
      2.4.2 Graph Algorithms . . . . . . . . . . . . . . . . . . . . . . . 28
   2.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

3 The Polyhedral Model 30
   3.1 An Algebraic Representation of Programs . . . . . . . . . . . . . . . . . 30
      3.1.1 Iteration Domains . . . . . . . . . . . . . . . . . . . . . . . 32
      3.1.2 Scheduling and Placement Mappings . . . . . . . . . . . . . . . 33
      3.1.3 Access Relations . . . . . . . . . . . . . . . . . . . . . . . . 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Program Transformations</td>
<td>37</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Dependence Relations</td>
<td>38</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Finding a Legal Transformation</td>
<td>40</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Scheduling Strategies</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Code Generation</td>
<td>48</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Scanning Polyhedra using Fourier-Motzkin Elimination</td>
<td>48</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Modern Polyhedra Scanning</td>
<td>52</td>
</tr>
<tr>
<td>3.4</td>
<td>Polynomial Extensions: Semi-Algebraic Sets</td>
<td>52</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Handelman and Schweighofer’s Theorems</td>
<td>53</td>
</tr>
<tr>
<td>3.4.2</td>
<td>A Scheduling Algorithm</td>
<td>55</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>OpenStream: Dataflow Task-Parallelism</td>
<td>58</td>
</tr>
<tr>
<td>4.1</td>
<td>The Base Language</td>
<td>58</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Flowing Data: Streams and Tasks</td>
<td>59</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Spawning Concurrency: The Control Program</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Dependence Analysis</td>
<td>64</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Dataflow Dependencies</td>
<td>64</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Barriers</td>
<td>65</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Task Graphs</td>
<td>66</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Happens-Before Relation and Execution Schedules</td>
<td>66</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Deadlocks</td>
<td>68</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Absence of Consumers or Producers</td>
<td>68</td>
</tr>
<tr>
<td>4.3</td>
<td>A Note on Syntax</td>
<td>69</td>
</tr>
<tr>
<td>4.4</td>
<td>The Polyhedral Fragment of the OpenStream Language</td>
<td>71</td>
</tr>
<tr>
<td>4.5</td>
<td>The Case for Polyhedral Dependence Sets</td>
<td>71</td>
</tr>
<tr>
<td>4.6</td>
<td>Summary</td>
<td>75</td>
</tr>
<tr>
<td>5</td>
<td>Scheduling for Granularity Control</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>Tiling the Control Program</td>
<td>78</td>
</tr>
<tr>
<td>5.2</td>
<td>New Tasks, new Streams, new Communications</td>
<td>87</td>
</tr>
<tr>
<td>5.3</td>
<td>Assembling the Work of Fused Tasks</td>
<td>100</td>
</tr>
<tr>
<td>5.4</td>
<td>Lessons from Preliminary Experimental Results</td>
<td>107</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary</td>
<td>108</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6</td>
<td>Scheduling for Stream Bounding</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>6.1 Back-Pressure Dependences</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>6.2 The Problem with Non-Causal Schedules</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>6.3 Minimising Stream Sizes</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>6.4 Guidelines for Practical Applicability</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>6.5 Summary</td>
<td>120</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion and Perspectives</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>7.1 Summary of the Thesis and its Contributions</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>7.2 Future Work</td>
<td>123</td>
</tr>
</tbody>
</table>

Index of Notation 128

Personal Bibliography 132

Bibliography 132

Word Count: 32314
## Figures

2.1 An SDFG, a DPN and a MWEG representing the same computation  .  21

3.1 Polynomial multiply: abstract syntax tree  . . . . . . . . . . . . . . .  36
3.2 Polynomial multiply: dependences  . . . . . . . . . . . . . . . . . . . .  39
3.3 Polynomial multiply: wavefront parallelism  . . . . . . . . . . . . . .  44
3.4 Polynomial multiply: coarse-grain parallelism  . . . . . . . . . . . . .  45
3.5 Polynomial multiply: rectangular tiles in the transformed space  . . .  47
3.6 Polynomial multiply: skewed tiles in the original space  . . . . . . . .  48

4.1 Example dataflow patterns induced by four tasks  . . . . . . . . . . .  63
4.2 Dynamic evolution of the task graph with the control program  . . .  66
4.3 Simulation of the execution of a task graph  . . . . . . . . . . . . . .  67
4.4 Self-dependent task: to or not to deadlock?  . . . . . . . . . . . . . .  68

5.1 Iteration domains for Gauss-Seidel 1d task creation statements  . . .  80
5.2 Tasks that contribute to advancing the window first indices  . . . . .  82
5.3 Dependence patterns for the Gauss-Seidel 1d kernel  . . . . . . . . .  84
5.4 Tiled iteration domain for the Gauss-Seidel 1d kernel  . . . . . . . . .  85
5.5 Cross-tile dependences for the Gauss-Seidel 1d kernel  . . . . . . . .  89
5.6 Communicating iterations for write to left-peek dependence  . . . . .  92
5.7 Iterations with external reads for the Gauss-Seidel 1d kernel  . . . .  95
5.8 Overlapping external communication streams  . . . . . . . . . . . . . .  97
5.9 Minimum bounding boxes for two fused tasks  . . . . . . . . . . . . . . 102

6.1 Back-pressure dependences serialise an otherwise concurrent task graph 112
6.2 Back-pressure dependences for a program with no non-causal schedule 115
6.3 Back-pressure dependences for Pareto minimal stream sizes  . . . . . . 118
Listings

3.1 Polynomial multiply kernel ........................................... 32
3.2 Conditional array initialisation ................................. 33
3.3 Parallel polynomial multiply kernel ......................... 51

4.1 Control program for the four-task example .............. 70
4.2 SCoP fragment expressing a polynomial first index on a stream ... 72
4.3 Symbolic bursts do not imply non-affine first indices .... 73
4.4 Higher dimensional spaces do not imply non-affine first indices ... 73

5.1 OpenStream implementation of Gauss-Seidel 1d ....... 78
5.2 Applying the tiling transformation to the Gauss-Seidel 1d kernel ... 86
5.3 Building an access window for a new coalesced read .... 91
5.4 Declaring a stream array for the overlapping peek and read accesses 98
5.5 Building a new window for the overlapping peek and read accesses ... 99
5.6 Declaration of a shared array for intra-tile communication ....... 103
5.7 Aggregating the work for the new fused tasks ............... 105
5.8 Reading the write to left-peek input of each fused task ...... 107

6.1 SCoP with independent writes and independent reads .... 111
6.2 Triangular SCoP with independent writes and independent reads ... 112
6.3 SCoP whose tasks must execute in strict non-causal order .... 113
6.4 SCoP with intertwined stream accesses ................. 117
Abstract

POLYHEDRAL ANALYSIS OF STREAMING TASK-PARALLEL APPLICATIONS
Nuno Miguel Nobre
A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy, 2021

The microprocessor industry constantly welcomes new hardware features. Within the last two decades alone, it witnessed the revolution that took desktop computing from the traditional uniprocessor to the ubiquitous multiprocessor, the pervasive adoption of the energy-minded heterogenous multiprocessor on mobile devices, and the emergence of on-chip specialised accelerators. Supporting the programmer in the development of parallel, efficient and portable applications has thus never been more relevant.

Task-parallel programming is an increasingly popular solution to unleash the parallel processing power of modern architectures. By omitting platform-specific code and leaving the choice of when and where to execute tasks to a runtime system, it enhances performance portability. Yet, the programmer is still responsible for defining tasks and their interactions, ultimately dictating performance-sensitive properties such as the available concurrency, data access patterns and data communication volume.

We leverage polyhedral compilation techniques to boost programmer productivity for OpenStream, a streaming dataflow language supporting the specification of concurrent tasks communicating through streams. First, we introduce a task fusion algorithm which improves performance by increasing data reuse within tasks and reducing communication volume across tasks. Second, we study program execution within limited memory by considering a subset of runtime behaviour where stream sizes fall below specified bounds, and exploit polynomial extensions to the polyhedral model to devise a strategy capable of conservatively certifying execution under such conditions.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.library.manchester.ac.uk/about/regulations/) and in The University’s policy on presentation of Theses.
Acknowledgements

I would like to begin by thanking the mental pressure I incurred during this time of my life. It has certainly made me not only a better runner, but also a much better baker than I was just four years ago. I can make croissants! Can you imagine?

Most importantly, however, I must express my gratitude to all the amazing people who were by my side during this journey, including my supervisory team, my friends and my family.
Chapter 1

Introduction

Over the years, computer designers have leveraged both technological and architectural improvements to deliver ever-increasing computing performance. In particular, starting in the mid-1980s, both contributed to an unprecedented performance growth [BC11, HP17].

On the one hand, the rapid evolution of semiconductor manufacturing process technology brought exponential transistor density growth, the major driver behind Moore’s law [Moo65], while keeping the power density roughly constant, a trend otherwise known as Dennard scaling [DGY+74]. On the other hand, the recently emerging RISC (Reduced Instruction Set Computer) architectures leveraged their fixed-length instructions to capitalise on the abundance of transistor-integration capacity to employ the use of caches and of an array of instruction-level parallelism techniques: initially pipelining, then multiple instruction issue and later out-of-order execution, which became common in the early 1990s.

Risking obsolescence, the CISC (Complex Instruction Set Computer) x86-family of architectures rose to the challenge by internally breaking down their complex instructions into micro-operations that were amenable to the same innovations. As transistor counts soared, the hardware overhead of translating the more complex x86 instructions became negligible. Since power and silicon area were not a big issue on the desktop and server markets, uniprocessor performance grew steeply.

The fairytale ended around 2003/04, when chip clock rates started to flatten, pressured by thermal-limiting power utilisation: it was the end of Dennard’s scaling. All major manufacturers ceased their high-performance uniprocessor projects and acknowledged the road to higher performance would rather be via multiprocessor chips\(^1\) [KBT04, HP17].

\(^1\)Also multicore processors.
Adv05, KAO05, Ram06]. This marked a momentous switch from relying solely on instruction-level parallelism to coarser, application-level data and task parallelism. Whereas the compiler and hardware are able to exploit the former implicitly, without the programmer’s intervention, the latter frequently require the development of explicit parallel programs. This is often a major burden for programmers and has been the object of many research efforts and, incidentally, of this thesis. Before delving into the software details, however, we recap recent developments in the hardware landscape.

Since the mid 2010s, the power-conscious performance demands of the skyrocketing mobile devices market provided the circumstances for RISC architectures to catch up. In fact, the mobile space has always been dominated by RISC machines, whose power and size constraints could never afford the x86 translation overhead. With the increasing transistor density, mobile phones and tablet computers benefited from their simpler fixed-length instruction set to dramatically increase the width of their decoding units which, by the end of the decade, fed deeper out-of-order windows than their desktop counterparts. Currently, RISC processors are capable of comparable sequential performance at a fraction of the clock frequency and power draw of their x86 counterparts, a trend which is leading to their adoption for desktop-class applications.

There were a few other reasons contributing to this transition. First, the business model and market of traditional x86 manufacturers, which cherishes backwards compatibility, prevented the simplification and improvement of the instruction set and microarchitecture. Second, the fabless model employed by mobile vendors meant they were better prepared to embrace the newest semiconductor manufacturing process technology developed by specialised foundries. Lastly, mobile vendors are increasingly licensing and assembling third-party intellectual property to design in-house System on a Chip (SoC) solutions which include various heterogeneous and specialised hardware accelerators. These are bound together via local, on-chip, shared memory, in an effort to simultaneously mitigate the memory latency issues of the high-performance uniprocessor and the memory bandwidth issues aggravated by multiprocessor chips.

1.1 The Challenge of Parallel Programming

As mentioned, the industry’s rapid shift to multiprocessor architectures and specialised hardware with differing capabilities hinders program development. Since programmers are simultaneously expected to uncover parallelism in the application and to develop both scalable and performance-portable code, approaches continue to emerge
that strive to promote their productivity. The tendency is to allow programs to be expressed in architecture agnostic programming models and leave the generation of well-orchestrated machine code for a specific target architecture to optimised libraries, compilers and runtime systems. In this spirit, multiple programming models were created. Some target both centralised and distributed\textsuperscript{2} shared-memory multiprocessors, such as Pthreads (POSIX threads) which allows fine-grained control over thread management, or OpenMP (Open Multi-Processing) which, by providing a higher-level abstraction, tends to be both more portable and scalable. Others, such as MPI (Message Passing Interface), target distributed memory systems, such as those in clusters or warehouse-scale computers.

In the context of shared-memory multiprocessors, OpenMP [Ope97] popularised the fork-join model of parallel computing. In this paradigm, a programmer can dynamically create or fork concurrent units of work, or tasks, and later terminate or join them. This enables the easy expression of data loop parallelism and the incremental parallelisation of existing serial code. In addition, the code is portable across a range of hardware owing to its architecture agnostic nature, the continuous evolution of the standard and widespread compiler support.

However, join (or barrier) synchronisation is a control flow mechanism, with no awareness of the tasks’ workload duration or data communication patterns. It might, therefore, over-approximate dependences and cause resource underutilisation, harming performance scalability. Furthermore, parallel programming is prone to bugs which are difficult to track and correct. Incidentally, the shared-memory model underlying OpenMP is liable to race conditions which may induce concurrency bugs with low reproducibility. This underlines the importance to productivity of creating parallel programs that are reproducible by construction and, ideally, whose correctness can be checked statically, i.e. at compile-time.

1.2 Towards a Solution: The Dataflow Paradigm

Hardware history tells us high-performance computing benefits from considering the limitations of the memory subsystem and from being guided by data availability. High speed caches aim at reducing latency by exploring the temporal and spatial locality of reference in computer programs. On-chip memory is closer to the processor’s die,\textsuperscript{2}The former are typical of single-chip multicore systems, such as consumer devices, and usually provide Uniform Memory Access (UMA). The latter are usual in multiple-chip multicore designs and provide NonUniform Memory Access (NUMA).
reducing latency, and brings the high bandwidth memory typical of graphics processing units to all the processing elements on the SoC.

Lastly, out-of-order execution relaxes the textual order of instructions in a program to allow the overlap of independent operations with memory accesses, avoiding stalls, hiding latency. The decision to execute an instruction is data-driven, i.e. based on operand availability and not a program counter, and might result in the scheduling of an instruction appearing later in the program text ahead of earlier appearing instructions. We say this execution paradigm reflects the dataflow of the program instead of strictly following its inherent control flow. In fact, the dataflow model of execution can be seen an extension of this same concept beyond the scope of a processor’s out-of-order window and the instruction-level granularity.

In the last decade, task-parallel dataflow programming models have risen in popularity, as shown by StarSs [PBAL09], in particular its OpenMP-based implementation, OmpSs [DAB+11], QUARK [YKD11], the dataflow runtime of the PLASMA dense linear algebra library [BLKD09], PaRSEC [BBD+11], StarPU [ATNW11], X-KAAPI [GLFR13], LIBKOMP [BGD12] and OpenMP 4.0 [Ope13]. Programs can be represented by directed acyclic task graphs where computing tasks are connected by edges modelling data dependences. Unlike task-parallel fork-join models, e.g. Cilk [BJK+96] and OpenMP 3.0 [Ope08], these unlock precise, point-to-point synchronisation of parallel activities, avoiding the expensive and poorly scalable global consensus of fork-join patterns and enabling the simultaneous exploitation of pipeline, data, and task parallelism. Additionally, their data-driven approach mitigates the impact of the memory subsystem by naturally increasing resource usage, hiding latency and favouring on-chip communication, bypassing the lower levels of the memory hierarchy. Notably, this model also frees the programmer from explicitly specifying parallelism. Instead, the compiler and runtime system cooperate to automatically exploit the inherent parallelism in the application as specified by the data (in)dependences between tasks.

In particular, task-parallel streaming dataflow languages, e.g. StreamIt [TKA02, KRA05], Concurrent Collections (CnC) [BBC+10], ΣC [GSLD11] and OpenStream [PC13], have gathered special interest. In these frameworks, data is communicated between tasks using unbounded, single assignment channels or streams. This property ensures all data dependences are (data)flow dependences: the edges of the task graph represent the flow of communicated data. It also avoids the race conditions generated by the

---

3Which, despite the model name, might include data anti and output dependences.
in-place writes of shared-memory models, paving the way to functional determinism\(^4\), a valuable asset in the race for productivity.

However, exploiting the advantages of such models incurs two major obstacles. First, it requires the programmer to explicitly define tasks, pressing them to make good choices regarding task granularity. On the one hand, finer-grained task graphs provide higher concurrency and better opportunities for load balancing, but on the other hand, coarser-grained task graphs are usually characterised by fewer inter-task communications, superior data locality and reduced scheduling overhead.

Second, in order to realise streaming programs on a realistic chip with bounded resources, streams cannot be allowed to grow beyond the available memory of the device. Limiting the amount of in-flight data on the streams might cause a task’s write action to block, which, in turn, may lead to a deadlock state.

Both these issues bring back familiar problems: they either compromise the productivity of the programmer or, if left unattended, the application’s performance portability and scalability.

1.3 Tuning Streaming Dataflow: The Polyhedral Model

Ideally, the aim should still be to relieve the programmer from the mentioned burdens by leaving the analysis, transformation and generation of optimised code to a compiler. To this end, every compiler leverages an internal program model to represent and reason about the application’s source code. Selecting such a model is frequently a compromise between a) expressive power and b) analysis and transformation efficiency and tractability. The abstract syntax tree is a common syntactical representation suitable for statement level analysis but is inadequate to accurately analyse the precise dependences between the task instances spawned by said statements. Similarly, in the presence of task graphs with infinite, very large or a priori unknown number of tasks, relying on data structures extensionally\(^5\) describing the graph might easily lead to unmanageable memory requirements and code generation [Pou13]. Hence the pursuit for program representations in which expressiveness is traded off against effectiveness. The polyhedral model is such a representation.

\(^{4}\)A program is said functionally deterministic if it computes a function of its inputs, i.e. if it is free of side effects and if its output is entirely determined by its inputs. If used as a subprogram in a larger codebase, we also say it is referentially transparent.

\(^{5}\)Explicitly listing or specifying every object that falls under a definition. For example, adjacency matrices are a common representation of finite graphs which effectively list all connected vertices.
CHAPTER 1. INTRODUCTION

The polyhedral model (also known in the literature as the polytope model) [Pug91, Fea91, FL11] is an algebraic framework that supports exact dependence analysis and complex transformations of imperfectly nested counted loops for automatic parallelisation and data locality optimisation. Born in the context of classical imperative languages, its flexibility in transformations is achieved at the expense of restricting the expressiveness of the model. In its classical setting, loop iteration domains, memory accesses, dependences, execution order and schedules are all represented, in intension\(^6\), by \(\mathbb{Z}\)-polyhedra, i.e. sets of integer solutions to a system of affine (in)equalities. Nonetheless, since its inception, it has met with a fair measure of success [Fea92a, Fea92b, Lim01, Gri04, PBCV07, PBCC08, BHRS08].

The motive of this thesis is to adapt and apply existing polyhedral techniques to task-parallel streaming dataflow applications. Contrary to the polyhedral model’s traditional program targets, whose dataflow analysis is the study of explicit array references, streaming programs generally leave channel management and addressing to the compiler and runtime system which are responsible for ensuring every task accesses its rightful data as prescribed by the language’s semantics. While an advantage for the programmer, the implicit memory management is a clear challenge for polyhedral compile-time analysis.

1.4 Contributions

Our work targets programs written in OpenStream [PC13], a state-of-the-art task-parallel dataflow streaming language\(^7\). Past research has studied the applicability of polyhedral techniques to OpenStream [CDF16]. The authors showed that, even for the polyhedral subset of the language, dependences among task instances lead to polynomials of arbitrary degree, impairing traditional polyhedral techniques, only suitable for linear polynomials.

First, we demonstrate that, for those programs not expressing polynomial dependences, it is possible to adapt known affine abstractions to provide:

**Task Fusion** through an algorithm capable of generating efficient streaming code expressing task graphs whose granularity is well suited to a target architecture. This involves a novel procedure that enables the compilation of coarser-grained tasks and

---

\(^6\)By predication, describing a subset of a known set.

\(^7\)This choice was driven not only by a survey, presented in chapter 2, of alternatives in the streaming language landscape, but also by the close collaboration with Antoniu Pop, the author of OpenStream.
stream communications from an initial finer-grained and programmer-friendly OpenStream specification.

Second, in the context of polyhedral OpenStream programs expressing polynomial dependencies, we leverage the polynomial extensions to the polyhedral model developed in [Fea15] and the results related to deadlock detection in [CDF16] to support:

**Bounded Scheduling** providing compile-time deadlock-freedom guarantees regarding the memory bounded execution of streaming programs. This entails searching for a (possibly polynomial) schedule, i.e. a logical date prescribing the execution of each task, whose existence certifies the absence of deadlocks when the capacity of buffers is limited. A new strategy is devised which, despite over-constraining and unable to conclude on the existence of a schedule for every program, can conservatively establish the execution on memory constrained hardware.

1.5 Thesis Outline

This thesis is laid out as follows. Chapter 2 presents an overview of related work on data-driven models of computation, including previous efforts on applying polyhedral techniques to such models, task fusion and bounded memory scheduling.

Chapter 3 describes the classical polyhedral representation of imperative programs and a slight extension allowing polynomial constraints and relations. Two recent mathematical theorems allowing the construction of polynomial schedules, an essential ingredient of chapter 6, are briefly reviewed.

Chapter 4 introduces OpenStream, the task-parallel streaming dataflow language we have selected for the study of the concepts proposed in this thesis, with a focus on the semantics of task communication using streams and the resulting dataflow dependences. The polyhedral fragment of the language and further restrictions, required in chapter 5, guaranteeing polyhedral dependence sets are also described.

Chapter 5 presents our solution for automatically fusing dataflow tasks based on polyhedral tiling techniques while illustrating it on a Gauss-Seidel OpenStream implementation. Tasks are coalesced, thereby reducing the volume of communicated data, simplifying runtime scheduling and instigating intra-task data reuse. In addition, streams are also unified based on the program’s dependence patterns, simplifying dependence resolution at runtime.
Chapter 6 restricts the range of schedules available at runtime by augmenting the intrinsic dataflow dependences of the program with back-pressure task dependences. These new dependences model the now bounded stream capacities, in turn reflecting the bounded nature of computer memory. Guaranteeing execution feasibility relies on the ability to find a polynomial schedule, a process which is known to be undecidable and found to be hindered by the over-constraining back-pressure dependences. In spite of these issues, a conservative method is derived which is capable of providing a certificate for correct execution.

Chapter 7 summarises the main contributions of this work and offers some insight into possible future work directions.
Chapter 2

Context and Related Work

Historically, the first published work introducing the data-driven concept as a precise representation for computer programs and systems appeared in the late 1960s [KM66, Ada69, Rod69]. In the coming years, a plethora of related computational models were published but, fundamentally, they all describe programs with a common structure: a directed flow graph\(^1\) of computing nodes connected through edges representing data paths. We start by an overview of the many incarnations of this paradigm which is not restricted to streaming dataflow but also covers related models, such as process networks and Petri nets. We also report on hardware endeavours through a description of a few of the most popular dataflow architectures. Finally, we review related work on polyhedral techniques, task fusion and bounded memory execution in the context of data-driven models of computation.

2.1 A Zoo of Data-Driven Models of Computation

It was Dennis’ [Den74] seminal work that ignited the research field of streaming dataflow programming. The idea envisioned a language capable of exposing high levels of concurrent activity without sacrificing expressiveness and while maintaining a guarantee of determinacy.

In dataflow, programs can be described using a directed graph, called a Dataflow Graph (DFG) [LP95, Den11]. A program is divided into pieces (also actors, nodes, blocks or tasks) defining a functional quantum of computation, which are connected through edges (also arcs) representing unidirectional FIFO channels (also buffers or

\(^{1}\text{Not to be confused with the directed acyclic task graph, mentioned earlier, whose edges represent precedence constraints between dependent tasks.}\)
streams) of (a priori) unbounded capacity. The execution, or firing, of an actor is specified by a set of firing rules which precisely dictate how many, and possibly what, data units (also tokens or samples) must be present at its input arcs to allow firing. Once triggered, a firing can be viewed as an atomic, run-to-completion task, that produces some output tokens as a function of its input tokens. The amount of tokens consumed or produced in each arc are called sample rates and, in general, may change with each firing. Arcs may have an initial population of tokens, called delays, since they are read before produced tokens. Finally, the granularity of the actors will limit the amount of available parallelism since, even though different actors can sometimes be executed concurrently, the function inside each actor executes sequentially.

Dataflow graphs rose in popularity in large part due to their natural and convenient way of describing digital signal processing (DSP) systems. In addition, because actors lack side effects and compute a function of their input tokens, the whole graph can be proven functionally deterministic on its input streams, a desirable property for programmers and DSP applications. Unfortunately, however, the freedom granted to firing rules, which can depend on the data circulating on the channels, hinders static (compile-time) analysis and scheduling. Dynamic (runtime) scheduling is a costly approach, specially if each actor spends little time computing, increasing the relative importance of the runtime overhead.

The solution, as is often the case, was to reduce the expressiveness of the model. Synchronous\textsuperscript{2} DataFlow (SDF) [LM87a, LM87b] differs from traditional dataflow in that the amount of data tokens produced and consumed by an actor is specified a priori for each input and output channel. Under such conditions, it is possible to statically decide whether the execution is free of deadlocks and to statically schedule the execution in bounded memory. Figure 2.1 shows an SDF Graph (SDFG).

Over the years, researchers devised multiple extensions to SDF with the aim of increasing its expressiveness. Boolean DataFlow (BDF), which augments SDF by adding a select actor, routing one of two inputs to its output, and a switch actor, routing its input to one of two outputs, both under the control of a boolean input, has been shown Turing complete [Buc93]. As an important consequence, critical questions related to deadlock freedom or bounded memory execution become undecidable. For this class of dataflow graphs, some responsibilities must be inevitably left to a runtime scheduler.

\textsuperscript{2}The synchronous ‘moniker’ stems from the DSP world where it conveys the graph’s sample rates are constant and known.
2.1. A ZOO OF DATA-DRIVEN MODELS OF COMPUTATION

Figure 2.1: An SDFG (Synchronous DataFlow Graph), a DPN (Dataflow Process Network) and a MWEG (Marked Weighted Event Graph) representing the same computation. Actors \( a_A \) and \( a_B \) and the directly equivalent transitions \( t_A \) and \( t_B \), constitute well-defined quanta of computation. By contrast, processes \( p_A \) and \( p_B \) define a contiguous strand of execution. The arc with delays on the SDFG is replaced by a place with an initial marking on the MWEG. Stream \( F \) on the DPN must be initially populated by process \( p_A \). The interface with the unmodeled external environment is represented by triangular nodes, also known as cold transitions in the context of Petri nets.

Other models are more restrained, chasing increased expressiveness whilst not sacrificing analysability and the ability to statically schedule the execution. The most popular among these is probably the Cyclo-Static Dataflow (CSDF) [EBLP94, BELP95] model, which allows the actors’ sample rates to vary cyclically, as a periodic sequence of constant integers. StreamIt [TKA02, KRA05] and \( \Sigma C \) [GSLD11] implement the CSDF model. The former builds on the strong static restrictions of CSDF to enable aggressive compiler optimisations, achieving excellent and portable performance across a variety of targets [GTA06], even if only for applications amenable to this model.

On a more contemporary note, the Boolean Parametric DataFlow [BFGL13] model, for example, is a recent instantiation allowing parametric sample rates and dynamic changing graph topologies while still keeping the static guarantees of SDF.

Kahn Process Networks (KPNs) [Kah74] were proposed to aid on the parallel programming of distributed systems composed of a set of concurrent processes, interconnected by unidirectional FIFO channels of (a priori) unbounded capacity. Processes can be seen as generalisations of dataflow actors\(^3\) implementing an arbitrary sequential

\(^3\)Although the work of Dennis and Kahn was contemporary and independent.
program reading from and writing to streams. A read (also \textit{wait}) is blocking, it stalls until the buffer has sufficient tokens, whereas a write (also \textit{send}) always succeeds owing to the buffers’ unbounded capacity.

Kahn processes are functional maps from a set of input streams to a set of output streams, whereas dataflow actors are functions on tokens, the stream atoms. There is no well-defined quantum of computation, i.e. no notion of atomic firing, implying the existence of state, a program counter at a minimum. If a process stalls on a read and if there are more processes than processors, the stream’s source process must be activated, requiring a context switch. The repeated process suspension and resumption in this style of execution is thus relatively expensive\footnote{Dataflow actors can also store state, yet \textit{implicitly} on a feedback loop, i.e. an arc from the actor back onto itself, whose tokens can be used to affect the next firing. However, since such actors are still atomic and functions of their (augmented) inputs, this does not require any house-keeping when an actor terminates and another fires.}.

Dataflow Process Networks (DPNs) \cite{LP95} are a special case of Kahn Process Networks. They construct a process as a sequence of atomic dataflow actor invocations and are, essentially, a formalisation of DFGs as KPNs. As a result, for this class of process networks, as for DFGs, the context switch overhead may be entirely avoided since a compile-time or runtime scheduler may interleave the actor firings with no need to store state. Figure 2.1 shows a DPN with the same semantics as the SDFG from before.

Finally, on the one hand, since BDF graphs are a particular case of KPNs, the latter are also Turing complete \cite{Par95} and it is undecidable to detect if their execution will deadlock or happen within bounded memory. On the other hand, the functional determinism of DFGs can be seen as a consequence of the functional determinism of KPNs.

\textbf{Petri nets} \cite{Pet62, Rei13} are a very well-known model of concurrent and distributed systems. Petri nets are directed bipartite graphs whose arcs connect two nodes, a place and a transition, each from disjoint and independent sets. Places store or accumulate data tokens with no FIFO property. A marking is a distribution of tokens across places. Transitions are the computational units and work just like dataflow actors, consuming and producing tokens according to the labelings of the arcs connecting them to their input and output places, respectively. A transition is enabled, i.e. may occur (also \textit{fire}), if the current marking of its input places satisfies the transition’s preconditions, i.e. its input arc labelings. Like actors, transitions are also atomic, run-to-completion tasks, that produce some output tokens as a function of its input tokens. However, since
multiple transitions might connect and be enabled by the same places, there might be conflicts. Thus, in general, Petri nets are non-deterministic.

However, there are instantiations of the Petri net concept which are deterministic. For instance, in a Marked Weighted Event Graph (MWEG) [TCWCS92], every place is restricted to a single incoming and outgoing arc weighted by what can be seen as the sample rates of the connected transitions. This subclass restores the single-consumer, single-producer semantics of the previous models. In fact, it is formally equivalent to SDF. A MWEG for the same example presented before is shown in fig. 2.1.

It is clear that, in the past 50 years alone, a variety of models has been built on data-driven techniques. Each has their own particular features, preferred fields of application, user groups and tools. However, it is hard not to notice their similarities, as evidenced by fig. 2.1.

Notice that none of the models above specifies relative timings for actor, process or transition execution. They all define the preconditions for execution but, in the absence of a schedule, the underlying computation model does not itself prescribe any execution or token consumption/production timing guarantees. By construction, the correctness of programs expressed in these asynchronous languages does not depend on the pace of its different steps. In contrast, the real-time processing guarantees of reactive systems have been the main motivation for the development of synchronous models.

Synchronous Dataflow languages, such as Signal [BL90] and Lustre [CPHP87, Pla92], bring a notion of time front and centre and have been fairly successful in industrial applications due to their determinism, deadlock-freedom, safety and boundedness guarantees. Each data stream is associated with a clock which defines the alignment of tokens in different streams. These streams do not behave like the FIFO communication channels of DFGs or KPNs and are sampled by each computational process (also node) rather than consumed.

Operation under the ‘Synchronous Hypothesis’ assumes infinitely fast processors, i.e. that nodes respond instantaneously to their input or, equivalently, that no new input events occur until the currently executing nodes have completed. This is in stark contrast with asynchronous models, where the notion of time cannot be given a precise semantics, since they are designed to make the behaviour of parallel systems independent of the speed of their elements. However, the same inherently tight synchronisation which makes this paradigm a good match for safety-critical real-time applications, also

\[5\text{Or continuing execution in the case of a Kahn process.}\]
over-specifies some applications, limiting its scope and making distributed systems hard to model [Lee02].

2.2 Hardware Implementations

In addition to stimulating the development of multiple models of computation, the inherent parallelism of data-driven approaches was also soon applied at the instruction level to build dataflow parallel architectures [AC86]. At least on paper, they promised more efficient exploitation of fine-grain parallelism in application programs when compared to traditional von-Neumann, control-flow machines.

Classic dataflow machines are very close to the abstract DFG model. However, since the arcs are assumed unbounded FIFO queues, in practice, any implementation must find pragmatic solutions to manage the circulating tokens. Traditionally, machines have been classified as either static or dynamic.

In Dennis’ Static Dataflow Machine [DM74], each arc can hold at most one token at a time. The inability to accumulate data allows no more than one instance of an actor to be enabled for firing, preventing this architecture from fully supporting iterative constructs and reentrancy, hindering performance.

In contrast, the dynamic approach of the Manchester [GKW85] and MIT [AN87] tagged-token dataflow architectures and their heir, the explicit token store architecture realised in the Monsoon machine [PC90], allow concurrent activation of multiple instances of the same actor. In fact, they modify the dataflow graph abstract model by associating a tag with each produced token, distinct from all other tags for tokens on the same stream. As firing of actors is enabled only when inputs with matching tags are available, tags keep track of the production ordering and there is no need for a FIFO discipline in the channels. Remarkably, token production or consumption need not happen in order, allowing the execution of dataflow graphs that would deadlock under the FIFO scheme but not under the tagged-token scheme [LP95]. The matching mechanism, however, traditionally limited the sample rates and the fan-in and fan-out of each actor to at most one token in each of two streams.

Although research into computer architectures for executing dataflow programs never proved capable of delivering performant machines [HCAA93], the legacy of the dataflow concept lived through the convergence of the control-flow and dataflow models of execution [LH93]. The idea is to allow conventional thread-level execution on control-flow processors to provide the infrastructure for coarse-grained dataflow,
mitigating the inefficiencies of pure hardware-based instruction-level dataflow. Instructions are grouped into larger grains for control-flow scheduling while grains are dataflow-scheduled via appropriate support from programming models and compiler and runtime technology.

2.3 Control-Driven Streaming Dataflow

The result of the convergence of the control- and dataflow paradigms can be readily evidenced by the previously mentioned StreamIt or ΣC implementations. However, some authors pushed this notion further and developed models of computation where the task graph, whose execution is data-driven, is dynamically and incrementally constructed by a control-driven strand of execution: either a standalone program or as part of the tasks’ body.

Concurrent Collections (CnC) [BBC+10] is based on the dataflow concept in that tasks (also steps) are functional on their inputs and execution can proceed whenever data is ready. It is also like streaming in the sense that a producer (resp. consumer) task need not know its consumer (resp. producer) tasks, it just needs knowledge of what streams (also data collections) to write to (resp. read from). However, contrary to conventional streaming dataflow models, control is a first-class construct, capable of dynamically prescribing task execution — either from user code external to the CnC graph specification (also the environment) or as part of the execution of a task body — and, like dynamic dataflow architectures, production and consumption of tokens (also data items) need not be performed in FIFO fashion in order to maintain determinism. The extra flexibility is gained by modelling streams as associative, tagged data collections\(^6\) whose data items cannot be overwritten. The caveat is on the requirement to efficiently manage these explicitly addressable data collections, which can hold irregular and dynamic amounts of items.

Control-Driven DataFlow (CDDF) OpenStream [PC12] follows a very similar philosophy to CnC. However, CDDF relies on DFG- and KPN-like unbounded streams for communication even though, like CnC data collections, CDDF streams support variable and unbounded sets of producers and consumers. Determinism is achieved not by enforcing a FIFO discipline on the channels, but rather by a deterministic interleaving of task accesses in streams governed by the control-flow of a standalone sequential

\(^6\)A generalisation of I-structures [ANP89] beyond integer indexing.
program (also control program). Thus, contrary to CnC, a task solely indicates the accessed streams but does not indicate the placement of its data on the streams. The order of attribution of stream indices to each task is completely guided by the control program. This mechanism simplifies data management at runtime, allowing the efficient implementation of classical streaming applications, e.g. indefinitely executing digital signal processing applications. However, its implicit addressing nature complicates static analysis.

The flexibility of control-driven dataflow which enables advanced patterns for parallel programming, such as dynamic pipelining, together with the purer streaming nature of OpenStream determined its use as the target for the contributions of this thesis.

2.4 Static Analysis of Data-Driven Programs

Finally, we review related work on static analysis, scheduling and program transformations. We start by previous applications of polyhedral techniques to data-driven applications.

2.4.1 Polyhedral Optimisations

Combining data-driven models of computation with polyhedral optimisations is an idea that has long been around. In [FMG17], the authors apply polyhedral transformations to the body of $\Sigma$C individual actors (also agents). As $\Sigma$C, like OpenStream, uses one-dimensional stream channels, the major hurdle is handling linearised accesses. To overcome this, the authors suggest either manual delinearisation or an automatic delinearisation method limited to two-dimensional arrays. The optimisations, however, remain limited to the work functions of each agent and have no awareness of the position of the agent in the dataflow graph, hindering inter-agent data locality optimisations.

In [SSPS16], a framework is developed to build OpenMP code from a program specified in DataFlow Graph Language (DFGL), which is a dependence based notation for dataflow parallelism based on the Concurrent Collections (CnC) model [BBC+10]. The language is further restricted to allow static polyhedral analysis. As mentioned, a key difference to CDDF OpenStream is that tasks access data items by addressing them explicitly, via tags, in this case restricted to integer tuples. Therefore, step specifications can be reordered in a DFGL program specification without hurting dependences and altering the task graph. This contrasts with OpenStream, where the order of task
creation determines dependences and the partial execution order of the task-parallel dataflow program. This subtle distinction is sufficient to avoid the limitations discussed in chapter 5 and makes the code generation process conceptually simpler. In addition, the explicit addressing avoids the linearisation functions appearing whenever mapping multidimensional arrays to one-dimensional streams, simplifying dependence analysis.

In [KPP+15], an integrated solution is developed that allows the generation of coarse-grain OpenStream tasks from polyhedral-amenable sequential code. There are two essential differences to our approach: (1) the framework’s input is sequential code, which means it cannot take advantage of explicit parallelism annotations to extend the scope of the polyhedral model and (2) the generated code only uses OpenStream annotations to pass flagging tokens between tasks to signal dependence satisfaction. This means that the program’s data lives in shared memory and is thus not channeled between tasks through streams. This is closer in behaviour to the OpenMP 4.0 task-parallel execution model than to streaming dataflow languages. For this reason, the code will still expose false dependences and may not be amenable to powerful data placement optimisations like the ones devised in [DPH+16] for NUMA machines.

The literature also includes similar efforts for task-parallel non-streaming dataflow programming models. The work in [CSS15] aims at applying polyhedral transformations to explicitly parallel OpenMP 4.0 code by leveraging explicit parallelism annotations. This again uses shared memory data structures, thus not avoiding false dependences and hindering more aggressive data placement strategies. However, as during dependence extraction, the analyser looks at and intersects the dependences derived from both the explicitly parallel constructs as well as the host C code, including within loop and task parallel regions, this can help prevent dependence overestimation. On the contrary, dataflow dependences in OpenStream are the only available information. On the one hand, this allows complete abstraction from the computational body of tasks, which thus can be treated as black boxes. On the other hand, if the initial task instantiation statements overestimate dependences, the optimisation opportunities lost as a result are, henceforth, unattainable.

In [RCAE+20], an automated framework is presented which, starting from polyhedral code, parallellises and assembles computations into tasks for distributed execution. An input sequential Python description is transformed into a task-parallel dataflow program specification for distributed memory systems in COMPSs [LTE+14]. Similarly to our work, the programmer has the option to request coarser-grained tasks but, since
it analyses sequential code, the framework cannot leverage explicit parallelism annotations to extend its scope beyond the traditional restrictions of the polyhedral model.

Finally, the technique in [Bon13] is similar but instead extracts its initial polyhedral representation from a C, C++ or Fortran program and targets distributed memory executions over MPI. There is no actual task creation but the groups of iterations executed by different processors act as such, receiving and sending data according to computation needs. However, contrary to our work, only wavefront parallelism surrounded by blocking communications is supported.

2.4.2 Graph Algorithms

In the context of static task graph scheduling and placement, i.e. when considering the problem of assigning a logical start time and allocating a processor to each task, list scheduling and clustering are two fundamental heuristics [Sin07]. List scheduling algorithms [ACD74, CG72, Gra69] simply sort tasks in a priority list, while respecting their dependences, and then successively schedule them to a processor, e.g. the one allowing the earliest start time for that task.

Clustering algorithms [DRV00, ERLA94, GY92], however, first try grouping tasks that depend on each other by idealising a system with as many processors as individual tasks, the initial clusters, and then incrementally merging clusters if the reduction in communication costs results in a shorter total execution time for the graph, i.e. compensates the eventual loss in concurrency. The final clusters are finally mapped to the physical processors and the tasks allocated on the same processor ordered by a list scheduler. Even if these algorithms only logically group tasks without actually fusing them, clustering algorithms, like the work in this thesis, still try to exploit the compromise between minimising interprocessor communication and maximising task execution concurrency.

In the context of static boundedness analysis, the work in [MKN05] in buffer minimisation for a restricted form of KPNs with finite processes goes beyond bounded execution guarantees, finding optimal buffer sizes and placing tight lower bounds on the required memory resources. The problem is proven NP-complete for as few as three tasks.

Lastly, buffer minimisation for SDF and CSDF graphs is known NP-complete [BLM96] and, under real-time throughput constraints, NP-hard [MMK10]. The work for SDF in [BMMU10] and CSDF in [BMK13] proposed approximations to find sufficient, close to optimal buffer sizes, in polynomial time. The bounded capacity of buffers
2.5. **SUMMARY**

is modelled, for every arc in the original graph, using a backward arc whose tokens represent the free spaces in the buffer.

The approach presented in chapter 6 to conservatively validate execution in bounded memory for OpenStream is similar, in that we aim at showing execution under specific buffer bounds and not the slightly more relaxed property of the possibility of bounded execution on finite, but arbitrary memory resources.

### 2.5 Summary

Data-driven models of computation are characterised by controlling program execution by the availability of data. The recent convergence with control-driven approaches saw the advent of programming models, such as CDDF OpenStream [PC12], capable of dynamically spawning a task graph from a sequential program specification, unlocking new levels of programming expressiveness. Due to being quite recent and to its peculiar multi-producer/consumer stream access policy, there is still little research on static program analysis and automated program tuning, with the exception of the notable work in [CDF16]. This thesis is a first effort on expanding on the task fusion and stream boundedness ideas brought forward in that document.
Chapter 3

The Polyhedral Model

Scientific programs are known to spend a large portion of their running time on loops operating on arrays. Since with present day applications, the number of iterations might be very large or even unknown (e.g. dependent on vector, matrix or discretisation sizes), any compiler facilitating static analysis and code restructuring and optimisation cannot aim to thoroughly describe the loop iterations and their addressing patterns in extension, i.e. by explicitly enumerating or listing all of them. A solution is to describe the set of operations in intension, i.e. by predication. The polyhedral model is such a description. It uses an algebraic representation for programs which comprises of \( \mathbb{Z} \)-polyhedra, sets obtained by affine predication of \( \mathbb{Z}^n \).

After reviewing how to represent a program in the model, we outline how the compact representation enabled by polyhedra unlocks efficient, scalable and automatic program parallelisation and optimisation, e.g. for better data locality. Finally, we are guided by the observation that, as we will see in chapter 4, dataflow dependence analysis for OpenStream programs touches on the model’s Achilles’ heel, its expressiveness. Indeed, the one-dimensional nature of streams leads to polynomial linearisation functions for which we require recent results extending the model to polynomial constraints.

3.1 An Algebraic Representation of Programs

Program fragments for which the model encodes the exact semantics are known as Static Control Parts (SCoPs).
3.1. AN ALGEBRAIC REPRESENTATION OF PROGRAMS

Static Control Part [Fea91] Consider a set of structure parameters, i.e. constants that are unknown at compile time and whose value depends solely on other structure parameters or on the outside world. A program part has static control if:

1. All its control statements are:
   a. Selection statements (also conditional or if statements) predicated by quasi-affine constraints on structure parameters and outer loop iteration counters (also loop index variables or loop iterators);
   b. Counted loop iteration statements (also for loops) limited by quasi-affine functions of the structure parameters and outer loop iteration counters;

2. All its simple (also atomic or computational) statements are scalar or array assignments. These can only involve expressions on other scalars, array references and, possibly, pure function calls (lacking side effects) whose arguments are also expressions of this kind. All indices in array references are restricted to quasi-affine functions of the loop counters and structure parameters.

Exactly what is understood by a quasi-affine expression can be formalised by the language of Presburger formulas [Ver16]. In a few short words, consider the definitions of:

- affine expressions, $\beta_i = \sum_{1 \leq j \leq n} a_{ij} x_j + \sum_{1 \leq k \leq m} b_{ik} p_k + c_i$ in $n$ loop iteration counters $\{x_j\}$ and $m$ parameters $\{p_k\}$ with coefficients $\{a_{ij}, b_{ik}, c_i\}$ in $\mathbb{Z}$;
- $\mathbb{Z}$-hyperplanes, $\{x \in \mathbb{Z}^n \mid \beta_i(x) = 0\}$; the subspaces on either side:
- closed $\mathbb{Z}$-half-spaces, $\{x \in \mathbb{Z}^n \mid \beta_i(x) \geq 0\}$; and their intersection:
- closed convex $\mathbb{Z}$-polyhedra, $\{x \in \mathbb{Z}^n \mid \forall i. \beta_i(x) \geq 0\}$.

For added expressiveness, one can define quasi-affine expressions by allowing integer division by a numerical constant and, from here, disjunctions of quasi-affine constraints and piecewise quasi-affine functions predicated by quasi-affine constraints (also quasi-affine selection trees or quasts [Fea91]). In the general case, these allow the description of unions of $\mathbb{Z}$-polyhedra which are highly common in polyhedral analysis.

Once a SCoP is identified, we abstract each statement of interest, typically simple assignment statements, using three main mathematical objects. So-called iteration domains, space-time mappings and access relations [Bas12].
3.1.1 Iteration Domains

A key aspect in the polyhedral model is the distinction between statements, the syntactic parts of the program text, and an operation or statement instance, which corresponds to a particular dynamic execution of a statement. The aim is to describe a program’s operations succinctly but comprehensively.

Our first need is an unambiguous designation of a statement instance. Since, in a SCoP, the only repetitive construct is the `for` loop, a statement instance can be uniquely defined by the name of the statement and the iteration vector, a tuple of the surrounding loop counters, whose dimension must equal the number of loops surrounding the statement. Together, they constitute an operation’s coordinate, such as \( S(a) \) for the \( a \)-th iteration of statement \( S \). For instance, in the following polynomial multiply kernel, \( S(k) \) and \( T(i,j) \) designate arbitrary instances of statements \( S \) and \( T \), respectively, while \( S(1) \) represents the particular instance \( c[1] = 0; \):

```plaintext
for (int k = 0; k < 2*N - 1; k++)
    c[k] = 0;  // S
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        c[i + j] = c[i + j] + a[i] * b[j];  // T
```

Listing 3.1: The product of two univariate polynomials of degree \( N - 1 \).

The statement’s iteration domain (also iteration space or even instance set) is the set of all its dynamic instances, as given by all the possible values of its iteration vector. It is described by the constraints on all the loop iterators surrounding the statement which, for a SCoP, are affine forms. Thus, the iteration domain for a given statement \( S \) surrounded by \( n \) loops is a convex \( \mathbb{Z} \)-polyhedron:

\[
D_S = \{ a \in \mathbb{Z}^n \mid \forall i \in \{1...n\}, \text{lb}_i(a[1..i-1]) \leq a[i] \leq \text{ub}_i(a[1..i-1]) \},
\]

where \( \text{lb}_i \) and \( \text{ub}_i \) are, respectively, the lower and upper bounds of the \( i \)-th loop surrounding \( S \) and \( a[i..j] \) is the subvector of \( a \) built from components \( i \) to \( j \).

---

Footnote: Given a vector of structure parameters \( p \), some authors explicitly denote polyhedra and affine forms as functions of said vector, e.g. \( D_S(p) \) instead of \( D_S \) and \( \text{lb}_i(a[1..i-1]; p) \) instead of \( \text{lb}_i(a[1..i-1]) \). Here, and in the remainder of this thesis, even though we chose to hide the parametric dependence, the reader should still assume it in the general case. For instance, as a function argument, \( a[i..j] \) can be seen as \( a[i..j] \equiv (a[i..j]; p) \).
For the polynomial multiply kernel from listing 3.1, we get:

\[ D_S = \{ k \in \mathbb{Z} \mid 0 \leq k \leq 2N - 2 \}, \]
\[ D_T = \{ (i, j) \in \mathbb{Z}^2 \mid 0 \leq i, j \leq N - 1 \}. \]

In general, we consider unions of polyhedra. These occur, for instance, when a disjunctive affine conditional statement in the input code splits the iteration domain into a union of disjoint polyhedral sets. To see this, briefly suppose the polynomial product above is being added to an existing polynomial with non-zero coefficients for terms of degree between \( L > 0 \) and \( M < 2N - 2 \). Then, the code should prevent \( S \) from re-assigning that range of array \( c \):

```c
for (int k = 0; k < 2*N - 1; k++)
  if (k < L or k > M)
    c[k] = 0; // S
```

Listing 3.2: Assuming \( c \) contains the non-zero coefficients of some polynomial for terms of degree between \( L \) and \( M \), statement \( S \) should not overwrite that region.

In which case the iteration domain would be written:

\[ D_S = \{ k \in \mathbb{Z} \mid 0 \leq k \leq 2N - 2 \land (k \leq L - 1 \lor k \geq M + 1) \} \]
\[ = \{ k \in \mathbb{Z} \mid 0 \leq k \leq L - 1 \} \cup \{ k \in \mathbb{Z} \mid M + 1 \leq k \leq 2N - 2 \}. \]

Finally, if statement \( S \) is not surrounded by any loop statement, but executes (e.g not surrounded by a trivially false conditional), its iteration vector is necessarily zero-dimensional. Its only instance is denoted \( S() \) and its domain \( D_S = \{ () \} = \mathbb{Z}^0 \).

### 3.1.2 Scheduling and Placement Mappings

Given a set of statement instances, there is still no information on when or where to execute each instance. Scheduling functions assign a logical date (also time stamp or epoch) for executing each instance, whereas placement (also allocation or distribution) functions determine which processor is responsible for execution.

A schedule for statement \( S \) is a function that to each statement instance associates an \( m \)-tuple of non-negative integers representing its \((m\text{-dimensional})\) logical execution time stamp:

\[ \theta_S : D_S \to D_{ST} \subseteq \mathbb{N}^m \]
\[ a \mapsto \theta_S(a). \]  

(3.2)
It defines a partial execution order on the elements of the instance set according to the lexicographic ordering, \(\prec\), of their time stamps. As hinted by the notation, it is common to simply see the image set \(D_{\Sigma} \subseteq \mathbb{N}^m\) of time stamps as a new iteration domain \(^2\) with a built-in, implicit execution order, the lexicographic ordering of its elements. From this perspective, we may say the statement’s iteration domain is transformed into a target domain, and indeed that the schedule map defines a program transformation.

Classically, the \(i\)-th one-dimensional component, \(\theta_i^S : D_S \rightarrow \mathbb{N}\), of a schedule, also known as a one-dimensional transform, is restricted to an affine form of the structure parameters and loop iterators. In these circumstances, a scheduling function can be written:

\[
\theta_S(a) = T_S a + c_S,
\]

where \(T_S\) is the transformation matrix, \(T_S \in \mathbb{Z}^{m \times n}\) for \(a \in D_S \subseteq \mathbb{Z}^n\), and \(c_S \in \mathbb{Z}^m\) is a constant vector, possibly parametric. In such cases, a schedule is simply a transformation between a polyhedral iteration domain and a target polyhedron of non-negative integer time stamps. Scheduling functions of this form can encode (and compose) a wide range of usual loop transformations, including skewing, interchange, reversal, tiling, fusion or splitting. Observe that it is not necessarily the case that the space embedding the target iteration domain preserves the dimension of the original iteration domain, i.e. that \(m = n\). For instance, when applying loop fusion or loop splitting, \(m > n\), since new information regarding the textual placement of statements must be given (see below), whereas when encoding parallelism implicitly, i.e. when only the sequential dimensions are explicit, \(m < n\).

A very useful example is the schedule of the original SCoP. Assuming the starting point is a sequential program or a program with the serial elision property, the lexicographic order of the iteration vectors encodes the sequential execution order between instances of the same statement. Consequently, the original schedule of such a statement is the identity function on the original iteration domain \(^3\), which is equivalent to simply equipping the original iteration domain with the lexicographic order from the start.

However, such construction still ignores the context of the statement within the

---

\(^2\) Although multiple statement instances might be mapped to the same time stamp since the mapping \(\theta_S\) is not necessarily injective.

\(^3\) Since the iteration domain is a subset of \(\mathbb{Z}^n\), such a schedule may include negative time stamps, thereby not conforming to definition (3.2). This poses no issue and the reader may assume a constant offset shifting every loop lower bound to the naturals for compliance with (3.2). In fact, requiring non-negative schedules is mere convention and does not affect code analysis or code generation.
program’s text, in particular regarding the relative position of other simple assignment statements. The solution is usually given by a mapping $\theta^\text{orig}_S$, the original schedule for statement $S$, directly extracted from the program text and labelled according to the program’s Abstract Syntax Tree (AST) [Fea92b]. Such mapping puts each statement in the larger context of the whole SCoP: it is the result of augmenting an identity map over the iteration domain of each statement with new, non-parametric constant label entries representing the textual order across different statements:

$$\theta^\text{orig}_S(a) = \pi_S || a, \quad (3.4)$$

where $\pi_S$ is the textual placement vector\footnote{Not to be confused with a processor placement function. Naturally, the textual information encoded in $\pi_S$ can also be expressed by appropriately built $T_S$ and $c_S$ in the affine form (3.3), thereby unlocking transformations such as loop fusion or loop splitting.} for $S$, a tuple of AST labels, and $||$ is the vector interleave operator which forms a new vector by taking an element at a time from each of its operands. The AST labels have the property that, if statement $S$ textually precedes statement $T$ and $N_{ST}$ is the number of common loops surrounding both statements $S$ and $T$,

$$\pi_S[1..N_{ST}] = \pi_T[1..N_{ST}] \ \land \ \pi_S[N_{ST} + 1] < \pi_T[N_{ST} + 1].$$

In particular, $\pi_S$ is unique and does not prefix $\pi_T$ for any statement $T \neq S$.

The original schedule of the polynomial multiply kernel from listing 3.1 is

$$\theta^\text{orig}_S(k) = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} k + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ k \\ 0 \end{pmatrix}$$

$$\theta^\text{orig}_T(i,j) = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} i \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ i \\ j \end{pmatrix}$$

The constant entries denote textual placement, as dictated by the arc labels in the program’s AST in fig. 3.1. In general, if $D_S \subseteq \mathbb{Z}^d$, then $\theta^\text{orig}_S(D_S) \subseteq \mathbb{Z}^{2d+1}$. Accordingly, such labelling is also commonly known as the ‘2d + 1 notation’ for iteration vectors [Bas12, FC18].
Figure 3.1: The abstract syntax tree for the polynomial multiply in listing 3.1. Each node corresponds to a statement in the program, recursively starting from the root statement, i.e. the whole program, down to the loop iteration statements and the assignment statements modifying the program’s data of interest, $S$ and $T$.

Armed with the original schedule for each statement, deciding which of two instances of any two, possibly different, statements, $S(a)$ and $T(b)$, is executed first, still boils down to a simple lexicographic comparison. Since the textual order is subsumed by the new iteration vectors, $S(a)$ precedes $T(b)$ if and only if $\theta^\text{orig}_S(a) \prec \theta^\text{orig}_T(b)$.

In this text, whenever we intend to decide which of two instances from any two statements comes first in a program, we will implicitly assume such a construction for the iteration vectors, so that lexicographic comparison always suffices. This means $a$ and $b$, in predicates such as $a \prec b$, will stand for the augmented iteration vectors $\theta^\text{orig}_S(a)$ and $\theta^\text{orig}_T(b)$ whenever $S \neq T$. Clearly, if $S = T$, $a \prec b \iff \theta^\text{orig}_S(a) \prec \theta^\text{orig}_T(b)$ and the distinction bears no relevance.

Finally, we remark that the mechanics behind placement is the same as for scheduling, just with different semantics. Instead of a logic date, a processor identifier is returned. Usually, in the polyhedral model, both problems are treated simultaneously under the same space-time mapping with some dimensions devoted to scheduling, others to processor placement. For this reason, we will henceforth use ‘scheduling’ to refer to both problems/mappings. At code generation time, the loops dedicated to processor allocation can be marked parallel.

---

5 As seen for the polynomial multiply kernel above, it may happen the tuples under comparison have different lengths. The lexicographic order predicate between $s \in \mathbb{Z}^n$ and $t \in \mathbb{Z}^m$, $s \prec t$, is equivalent to $s[1..\min(n,m)] \prec t[1..\min(n,m)] \lor s[1..\min(n,m)] = t[1..\min(n,m)] \land n < m$. For augmented iteration vectors, the latter never occurs, since it would imply prefix inclusion.
3.2. PROGRAM TRANSFORMATIONS

3.1.3 Access Relations

Lastly, access relations map statement instances from an iteration domain to the elements of some data array accessed by a particular array reference in the statement. For simplicity, we introduce a particular case of access relations, access functions. However, common language features like array slicing, or some approximation techniques do require mappings to multiple memory locations and, thus, non-functional relations.

Let $R_S = \{r_x\}$ and $W_S = \{w_x\}$ be the disjoint sets of read and write array references in $S$. $x$ is a unique label distinguishing between declared accesses of the same type. It could be the reference number in the statement if $x \in \mathbb{N}$, or perhaps a more exotic descriptor using symbols in a set $V$ if $x \in V^*$, e.g. a somehow relatable string or, if redundant, just the empty string $\epsilon$. $C_S = R_S \cup W_S$ is the set of all references in $S$.

Based on this abstraction, two functions are defined. Let $A$ be the set of all arrays in the program. The first gives, for each statement $S$, the array name for each syntactic array reference $c \in C_S$:

$$a^S_c : \{()\} \rightarrow A$$
$$() \mapsto a^S_c.$$  \hfill (3.5)

The second gives the array indices which the $a$-th instance of $S$ uses for access $c \in C_S$, one for each of the $p$ dimensions of the array:

$$i^S_c : D_S \rightarrow \mathbb{N}^p$$
$$a \mapsto i^S_c(a).$$  \hfill (3.6)

When $i^S_c$ is defined by affine functions, its image set is also a polyhedron.

Applying both definitions to the running polyhedral multiply example we get:

$$a^S_w = c \quad a^T_w = a^T_w = c \quad a^T_a = a \quad a^T_b = b$$
$$i^S_w(k) = k \quad i^T_w(i, j) = i^T_e(i, j) = i + j \quad i^T_a(i, j) = i \quad i^T_b(i, j) = j.$$

3.2 Program Transformations

Program transformations, whether performed with optimising or parallelising purposes, are all encoded by a scheduling mapping. We say a given transformation is valid when, and only when, it preserves the original program semantics. Thus, it is crucial we formalise the information that can guarantee it is preserved.
3.2.1 Dependence Relations

For a program transformation to be valid it is sufficient that it respects the data dependences in the program, as first formulated by Bernstein [Ber66]. Two instances \( S(a) \) and \( T(b) \) are in dependence, sometimes written \( S(a) \delta T(b) \), if \( a \prec b \), they both access a shared memory cell and at least one of them modifies it. Given their sets of read \( R_{S(a)} \) and written \( W_{S(a)} \) memory cells, we have:

\[
W_{S(a)} \cap R_{T(b)} \neq \emptyset \quad S(a) \delta^f T(b) \quad \text{flow or read after write dependence (RAW)}
\]

\[
R_{S(a)} \cap W_{T(b)} \neq \emptyset \quad S(a) \delta^a T(b) \quad \text{anti or write after read dependence (WAR)}
\]

\[
W_{S(a)} \cap W_{T(b)} \neq \emptyset \quad S(a) \delta^o T(b) \quad \text{output or write after write dependence (WAW)}
\]

(3.7)

The dependence set between statements \( S \) and \( T \), associated with accesses \( c \) in \( S \) and \( d \) in \( T \) where at least one is a write, i.e. \( c \in W_S \) or \( d \in W_T \), is then naturally defined as a binary relation over \( D_S \) and \( D_T \):

\[
\mathcal{D}^{S \times T}_{c \rightarrow d} = \left\{ a \rightarrow b \in D_S \times D_T \mid a \prec b \land a^S_c = a^T_d \land i^S_c(a) = i^T_d(b) \right\}.
\]

(3.8)

This builds on the simplifying assumption that two array references address the same memory location if and only if they are references to the same arrays and their indices are equal. In other words, programs with possible array name aliasing are not studied.

As before, if the iteration domains are polyhedral and all constraints are affine, the dependence set will also be a polyhedron or, more precisely, a union of polyhedra. This owes, most commonly, to the nature of the lexicographic order predicate, which is a disjunction of affine predicates. However, other possible culprits may include iteration domains expressed as unions of polyhedra or array accesses expressed through quasi-affine selection trees, e.g. by using the ternary operator.

The dependence relation corresponding to the read after write conflict between \( S \) and \( T \) for the polynomial multiply kernel in listing 3.1 can be obtained by replacing in (3.8) all the results we derived thus far:

\[
\mathcal{D}^{S \times T}_{w \rightarrow w} = \left\{ k \rightarrow (i, j) \in D_S \times D_T \mid \theta^\text{orig}_S(k) < \theta^\text{orig}_T(i, j) \land a^S_w = a^T_d \land i^S_c(k) = i^T_d(i, j) \right\}
\]

\[
= \{ k \rightarrow (i, j) \in D_S \times D_T \mid (0, k, 0) < (1, i, 0, j, 0) \land c = c \land k = i + j \}
\]

\[
= \{ k \rightarrow (i, k - i) \in D_S \times D_T \}
\]

\[
= \mathcal{D}^{S \times T}_{w \rightarrow w}.
\]
Similarly, the read after write dependences between instances of $T$ are written:

$$P^T_{w \rightarrow rc} = \{ (i, j) \rightarrow (i', j') \in D_T \times D_T \mid (i, j) \prec (i', j') \land i + j = i' + j' \}$$

$$= \{ (i, j) \rightarrow (i', j') \in D_T \times D_T \mid (i < i' \lor i = i' \land j < j') \land i + j = i' + j' \}$$

$$= \{ (i, j) \rightarrow (i', j') \in D_T \times D_T \mid i < i' \land i + j = i' + j' \}$$

$$= \{ (i, j) \rightarrow (i', j') \in D_T \times D_T \mid i' = i + 1 \land j' = j - 1 \}^+$$

$$= \{ (i, j) \rightarrow (i + 1, j - 1) \in D_T \times D_T \}^+$$

$$= P^T_{w \rightarrow w} = P^T_{w \rightarrow w}.$$  

where we used $Q^+$ to denote the transitive closure of relation $Q$. These dependences are depicted in the graph presented in fig. 3.2.

Figure 3.2: Dependences for the polynomial multiply kernel in listing 3.1. To avoid cluttering the illustration, we rely on the transitive closure of the drawn dependences to give the full dependence relations. Note that while only RAW dependences are explicitly drawn, WAR and WAW between the same instances are to be understood.

Let $N_{ST}$ be the number of common loops surrounding both $S$ and $T$. For any two instances, $S(a) \delta T(b)$, a dependence is said to be (carried) at depth or level $l \in \{1..N_{ST}\}$,
if and only if:

\[
  a[1..l - 1] = b[1..l - 1] \land a[l] < b[l].
\]  \hspace{1cm} (3.9)

A dependence may also be carried textually if \( N_{ST} = 0 \) or \( a[1..N_{ST}] = b[1..N_{ST}] \).

It is now possible to define a purely polyhedral dependence relation, i.e. one consisting of a single polyhedron, by splitting the lexicographic sequencing predicate into its conjunction-only predicates. When \( N_{ST} > 0 \), \( P_{S,T}^{ds \times T} \) contains all dependences at level \( l \in \{ 1..N_{ST} \} \), and is obtained just as \( P_{S,T}^{ds \times T} \) with the added restriction that the involved instances must share their first \( l - 1 \) iteration vector entries, \( a[1..N_{ST}] = b[1..N_{ST}] \). We write \( P_{S,T}^{ds \times T} \) as shorthand for \( \bigcup_j P_{S,T}^{ds \times T,j} \cup P_{S,T}^{ds \times T,\infty} \) or when \( N_{ST} = 0 \).

### 3.2.2 Finding a Legal Transformation

A valid or legal schedule map is such that for any two operations, \( S(a) \) and \( T(b) \):

\[
  S(a) \delta T(b) \rightarrow \theta_S(a) < \theta_T(b)
\]  \hspace{1cm} (3.10)

or, in other words, if \( \exists c, d. a \rightarrow b \in P_{S,T}^{ds \times T} \) then \( \theta_S(a) < \theta_T(b) \). Assuming a common loop nest in the target code (otherwise the dependence must be satisfied by the textual order), we say a dependence described by polyhedron \( P_{S,T}^{ds \times T,j} \) is satisfied at level \( l \) if and only if:

\[
  \forall k \in \{ 1..l - 1 \}. \quad \theta^k_T(b) - \theta^k_S(a) \geq 0, \quad a \rightarrow b \in P_{S,T}^{ds \times T,j} \quad \land \quad \theta^l_T(b) - \theta^l_S(a) \geq 1, \quad a \rightarrow b \in P_{S,T}^{ds \times T,j}.
\]  \hspace{1cm} (3.11)

Since every dependence must eventually be satisfied, the scheduling problem is thus, essentially, one of finding non-negative affine functions, \( \theta^k_T(b) - \theta^k_S(a) - \delta^k_{ST,cd,j} \), over polyhedral sets: each of the polyhedra \( P_{S,T}^{ds \times T,j} \) composing the dependence relation \( P_{S,T}^{ds \times T} \). Farkas’ Lemma can be used to characterise the space of legal affine schedules.

**Farkas’ Lemma (Affine Form)** [Far02] Let \( \{ \beta_i \}_{i \in \{ 1..n \}} \) be a set of affine forms in \( d \) variables. In the context of a non-empty convex polyhedron:

\[
  S = \{ x \in \mathbb{R}^d \mid \forall i. \beta_i(x) \geq 0 \},
\]  \hspace{1cm} (3.12)
an affine form \( f \) is non-negative everywhere in \( S \) if and only if it can be written as a non-negative linear combination of members of \( \{ \beta_i \} \):

\[
f(x) = \lambda_0 + \sum_{i=1}^{n} \lambda_i \beta_i(x), \quad \lambda_0, \lambda_i \geq 0.
\]  

(3.13)

The coefficients of the linear combination, \( \lambda_0, \lambda_i \), are known as Farkas’ multipliers.

Equating (3.12) to a dependence set, (3.13) can be used to find one-dimensional affine transforms. Making the vector \( p \) of structure parameters explicit:

(F1) Given:

- A set of affine forms \( \{ \beta_i(a, b; p) \}_{i \in \{1,...,n\}} \) defining a polyhedral dependence relation \( P_{c=d}^{S \times T[l]} \);
- A template for the difference of two one-dimensional affine transforms:

\[
t_{ST}(a, b; p) = \left( \mu_0 + \sum_j \mu_j a[j] + \sum_k \mu_k p[k] \right) - \left( \nu_0 + \sum_{j'} \nu_{j'} b[j'] + \sum_{k'} \nu_{k'} p[k'] \right).
\]

The two transforms need not differ, in which case all \( \mu = \nu \). Such situation occurs, for instance, if the dependence relation characterises a conflict between instances of the same statement, \( S = T \);

(F2) Invoke Farkas’ Lemma for some affine upper (* = +) or lower (* = −) bound \( \delta_{ST,c=d,l}(p) \):

\[
* \left( \delta_{ST,c=d,l}(p) - t_{ST}(a, b; p) \right) = \lambda_0 + \sum_{i=1}^{n} \lambda_i \beta_i(a, b; p), \quad \lambda_0, \lambda_i \geq 0;
\]

(F3) Sum and equate to zero the coefficients \( \{ \mu_0, \mu_j, \mu_k; \nu_0, \nu_{j'}, \nu_{k'}; \lambda_0, \lambda_i \} \) on the left and right hand sides corresponding to the same monomial on the components of \( a, b \) and \( p \);

(F4) Solve the resulting system for the \( \mu, \nu \) and \( \lambda \) using any convenient Linear Programming (LP) software. Some authors force integer \( \mu \) and \( \nu \), e.g. [BHRS08].

\footnote{Note that, given an iteration domain and a template for a single one-dimensional affine form, the same procedure could equally be applied to manipulate properties of a statement’s schedule (instead of those of a schedule difference over a dependence polyhedron), such as its maximum latency.}
CHAPTER 3. THE POLYHEDRAL MODEL

requiring an ILP solver, such as PIP [Fea88]. However, because the LP has rational coefficients, if it has a finite (optimum) solution, there is always a rational (optimal) solution [Tre11]. Then, the floor \( \lfloor \cdot \rfloor \) of the obtained one-dimensional schedule can be taken as a valid solution producing integer logical execution dates [Fea92a].

3.2.3 Scheduling Strategies

Given the machinery above, schedules can be tuned to optimise many different objectives. Examples include optimising a parametric upper bound to minimise a schedule’s latency or its delay between a dependences’s source and target instance, e.g. to maximise cache reuse or to minimise inter-tile communication volume; or choosing to satisfy dependences at shallower or deeper levels of the target loop nest to change the granularity of the exploited parallelism.

We assume parallel (i.e. placement) loops are given explicitly, meaning the number of linearly independent one-dimensional affine transforms, i.e. the number of linearly independent rows in the transformation matrix, is the same as the dimension of the original iteration domain, which is thus conserved in the target domain\(^7\).

We seek linearly independent one-dimensional transforms by constructing the orthogonal subspace of the transformation rows determined thus far, i.e. their null space. This is done using the pseudoinverse \( T_S^+ \) (also Moore–Penrose inverse) [Pen55, BHRS08] of the transformation matrix \( T_S \) computed so far, and forcing the new transform to include a non-zero combination of the rows of \( T_S^\perp \):

\[
T_S^\perp = I - T_S^+ T_S = I - T_S^T (T_S T_S^T)^{-1} T_S.
\]

Possible redundancy between multiples of a transform is avoided by minimising the absolute value of the transforms’ coefficients as a secondary objective of the LP\(^8\). Likewise, any trivial solution, i.e. constant transform, may be avoided as needed by augmenting the LP with the constraint that the sum of the absolute coefficients in the respective row of the transformation matrix is strictly positive.

\(^7\)Although the target polyhedron might be embedded in a higher dimensional space whose extra dimensions may account for, e.g. inter-tile traversal when tiling or textual order when doing loop fusion.

\(^8\)This is equivalent to finding the lexicographic minimum of a tuple with the main objective in its leading position followed by the schedule coefficients’ absolute values. Once a minimum for the main objective is found, it becomes a constraint on the involved variables and the resulting problem can be solved. The result is an optimal solution to the first objective that has been tuned, a posteriori, by a second objective.
Alternatives exist to build and traverse a legal space of program transformations. For example, instead of directly minimising the mapping coefficients, they can be bounded. The resulting space traversal can be exhaustive if small enough or, if large, conducted by heuristic or even genetic algorithms, iteratively guided by runtime execution metrics, e.g. extracted from hardware counters [PBCV07, PBCC08, Bas12]. In the following, we adopt the previous coefficient minimisation approach, but the results in this thesis do not depend on this choice.

We give a few examples guided by our running polynomial multiply kernel. For simplicity, only the iteration domain of statement $T$ is transformed, i.e. the SCoP is restricted to the loop nest surrounding $T$.

Finding Fine-Grained Parallelism

Maximising fine-grained parallelism equates to trying to satisfy the maximum number of dependences as eagerly as possible. In other words, we seek loop nests of outer sequential loops surrounding parallel loops, expressing wavefront, barrier-type parallelism.

Choosing the lower bound regime of (F2), the corresponding objective at some schedule level would be \( \max \sum_{ST,cd,l} \delta_{ST,cd,l} \) subject to \( \delta_{ST,cd,l} \in \{0,1\} \) for all dependence relations describing the SCoP under transformation. Then, the algorithm proceeds by removing all the already satisfied dependences, and solving for the next schedule dimension. When all dependences are satisfied, explicit parallel affine transforms are added to cover the remaining dimensions of the iteration domain.

Incidentally, for the polynomial multiply code in listing 3.1, the outer loop for $T$ satisfies all dependences, implying the original schedule for $T$ exposes fine-grained parallelism:

\[
\theta_T(i,j) = \binom{i}{j} = \binom{t}{p}.
\]

As illustrated in fig. 3.3, the new loop iterator $t$ represents time sequencing while $p$ denotes parallelism on multiple processors. Parallelising the original inner loop, e.g. by adding a \#pragma omp parallel for directive, is sufficient to reveal wavefront parallelism.
CHAPTER 3. THE POLYHEDRAL MODEL

Figure 3.3: Wavefront parallelisation of the polynomial multiply kernel in listing 3.1. The original schedule’s outermost loop satisfies all dependences, making parallelisation of the inner loop immediate.

Finding Coarse-Grained Parallelism

Coarse-grained parallelism works on the opposite premises. An optimising compiler should instead aim at lazy dependence satisfaction, surrounding sequential loops by parallel loops distributing the work across a multiprocessor.

In this case, we can reuse the work for finding fine-grained parallelism to determine the sequential loops. For the parallel loops we invoke (F2) in both its upper and lower bound versions for $\delta_{ST,cd,t} = 0$ over the set of all dependence relations describing the SCoP under transformation. These affine transforms define hyperplanes along which all dependences lie: no dependence is satisfied nor violated. Therefore, they can be permuted up to the upper levels of the loop nest.

Optimising our polynomial multiply code in listing 3.1 for coarse-grained parallelism yields a skewing of the original iteration domain for $T$:

$$\theta_T(i, j) = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i + j \\ i \end{pmatrix} = \begin{pmatrix} p \\ t \end{pmatrix}.$$
The new loop traversal is illustrated in fig. 3.4. The outer loop, indexed by $p$, is parallel.

Figure 3.4: Coarse-grain parallelisation of the polynomial multiply kernel in listing 3.1. Skewing the iteration domain reveals a dimension along whose constant hyperplanes all dependences lie and, thus, can be moved up the loop nest and parallelised.

### Tiling for Locality Optimisation

Tiling [IT88, WL91, Xue00] is a popular loop transformation technique in optimising for parallelism and data locality. Tiling partitions an iteration domain into smaller blocks (tiles) allowing data reuse, specially when the tile fits in the lower levels of the cache hierarchy. Tiles can also be used to expose efficient coarse-grain parallelism: independent tiles may execute concurrently on different processors and their higher operation counts reduce the frequency and volume of communication between dependent tiles in different processors.

The goal is to find affine transforms defining sets of hyperplane instances for which grouping, i.e. tiling, is valid. This requires looking for Forward Communication Only schedules [GFG05], i.e. those for which no dependence crosses an hyperplane into its lower half-space. In (F2), we choose the lower bound regime and set $\delta_{ST,cd,l} = 0$ over
every dependence relation and proceed iteratively, without removing previously satisfied dependences, finding new linearly independent transforms, until all dependences have been satisfied.

Of particular interest is the discovery of tiles with little or no cross-communication along a particular direction. To this end, we additionally leverage (F2) in its upper bound regime with a single parametric form for all dependences, \( \delta_{ST,cd,l}(p) = \delta(p) \), and set the minimisation of the parameter coefficients as the LP problem’s objective [BHRS08]. The aim is to minimise the reuse distance, i.e. the delay between the source and target iterations, in turn reducing the cross-communication volume: ideally, if zero, the reuse is internal to the tile for this dimension.

Sometimes, there may be circumstances in which we aim to tile only a sub-nest, i.e. the loops deeper than some level \( m \). In that case, only dependence sets of the form \( \mathcal{P}_{c\cdot d \cdot l, l \geq m} \), need be considered.

For the polynomial multiply of listing 3.1, the same affine transforms used before are valid tiling hyperplanes:

\[
\theta_T(i, j) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i \\ i + j \end{pmatrix} = \begin{pmatrix} i_t \\ j_t \end{pmatrix}.
\]

Then, we can simply define rectangular groups of iterations in the transformed space of size \( B_i \times B_j \). The tile sizes need not be numerical constants, since it is possible to generate parameterised tiled code [BHT10, THB10]. The bulk effect of composing the skewing transformation and the iteration grouping can be described by:

\[
\mathcal{D}_T \to \mathcal{D}_{T^\otimes} \\
(i, j) \mapsto (i_T, j_T) \mapsto (i_t, j_t) = \begin{pmatrix} i_t \\ j_t \end{pmatrix} = \begin{pmatrix} \frac{i}{B_i} \\ \frac{j}{B_j} \end{pmatrix} = \begin{pmatrix} i_T \\ j_T \end{pmatrix} = \begin{pmatrix} i + j \\ B_j \end{pmatrix} = \begin{pmatrix} i_t \\ j_t \end{pmatrix}.
\]

When displaying the new coordinates along orthogonal axes, the tiles are always rectangular, as seen in fig. 3.5. From here, nothing stops us from applying the approaches explained above for fine- or coarse-grain parallelism at the tile level: we need only reframe the problem to instead consider the inter-tile dependences\( ^9 \).

\( ^9 \)These are obtained by a projection of the original dependences onto the inter-tile dimensions, e.g. by Fourier-Motzkin Elimination, as presented in section 3.3, although in that section with the different purpose of code generation.
Assume we choose a coarse-grain approach, and assign every vertically aligned tile, i.e. every constant \( j \) hyperplane, to the same processing unit. At first sight, if compared to the coarse-grain parallelism of fig. 3.4, this alternative might seem counter-productive: by grouping constant hyperplanes for \( j \), i.e. columns of the iteration domain, some concurrency is lost. However, the compromise might be advantageous. In the event the target architecture lacks the \( 2N - 1 \) processors necessary to exploit all the available parallelism, the static placement might prove speedier than a dynamic alternative. In addition, although there is no (true) cross-communication along this dimension, false-sharing on cache-coherent systems is still possible. An appropriate tile size \( B_j \) might mitigate such shortcomings.

Lastly, note that if we choose to visualise the new coordinate space using the orthogonal axes describing the original domain, we get the (equivalent) depiction in fig. 3.6. Despite encoding the same information, the data layout, which for human-designed programs tends to follow the control flow of the program, might make this representation more intuitive in some cases.
CHAPTER 3. THE POLYHEDRAL MODEL

\[ \text{Statement: } S \]
\[ \text{Tiles in } T \]
\[ \text{Cross-tile dependences: } T \rightarrow T : w \rightarrow r_c \text{ and } r_c \rightarrow w \text{ and } w \rightarrow w \]

Figure 3.6: Tiling for statement \( T \) of the polynomial multiply kernel in listing 3.1. If viewed over the original axes, the iteration domain preserves its layout but the tiles appear skewed.

3.3 Code Generation

Once a transformation has been found by an optimisation heuristic, the final stage translates the transformed program from its polyhedral representation back to a syntactic representation. Code generation amounts to finding a strategy to scan the target polyhedra, i.e. to writing a set of nested loops visiting each of their integral points once and only once.

3.3.1 Scanning Polyhedra using Fourier-Motzkin Elimination

The original method, as pioneered by Ancourt and Irigoin [AI91], used the Fourier-Motzkin Elimination (FME) algorithm [Sch86] and was limited to a single target polyhedron and unimodular transformation matrices\(^{10}\), i.e. those preserving the polyhedron’s volume. Despite these restrictions, however, the method’s relative simplicity

\(^{10}\)A unimodular matrix \( A \) is a square integer matrix such that \( \det(A) = \pm 1 \) and defines a bijection over the integers.
3.3. **CODE GENERATION**

means it is also particularly insightful. This is true even in light of today’s solutions, mentioned later, which although leveraging different machinery, still produce the same results for those programs amenable to traditional FME.

The principle is quite simple: projecting the target polyhedron from its innermost dimension to its outermost dimension, deducing the loop bounds for each level in terms of the outer loop iterators (and any structure parameters) along the way.

The usual starting point is, for every statement $S$, its original domain $D_S \subseteq \mathbb{Z}^n$ and $a$, traditionally invertible, unimodular and thus square, $n \times n$ transformation matrix $T_S$, determining the scheduling map $\theta_S$. Setting $t = \theta_S(a) = T_S a + c_S$ with $t \in D_{S\theta} \subseteq \mathbb{N}^n$, $a \in D_S$, then we can speak of the original $\{a[i]\}_{1 \leq i \leq n}$ and the new, target $\{t[j]\}_{1 \leq j \leq n}$ loop iterator symbols. These form a pool of $2n$ variables in terms of which the target polyhedron is currently described.

First, we use the inverse transformation $T_S^{-1}$ to write each original loop iterator in $a$ in terms of the target iterators $t$:

$$a = T_S^{-1}(t - c_S).$$

Then, we replace the resulting forms in the constraints defining $D_S$ such that we obtain a description of $D_{S\theta}$ solely in terms of constraints on the $t[j]$.

Next, we initiate a succession of FME projections. Each target loop iterator $t[j]$ is eliminated, starting from the deepest up to the shallowest level, while using the determined bounds to build the new loop statement. Starting from the innermost iterator:

(FM1) FME partitions the set of affine constraints defining the target iteration domain into three new sets:

- Lower bounds for $t[j]$, $\text{lb}_i(t[1..j-1]) \leq t[j]$;
- Upper bounds for $t[j]$, $t[j] \leq \text{ub}_k(t[1..j-1])$;
- Constraints not involving $t[j]$, $\text{nb}_l(t[1..j-1]) \geq 0$.

(FM2) The maximum of the lower bounds and the minimum of the upper bounds determines the loop trip of this level:

$$\max_j \text{lb}_i(t[1..j-1]) \leq t[j] \leq \min_k \text{ub}_k(t[1..j-1]).$$

(FM3) Then, all lower and upper bounds are compared against each other, creating new
constraints:

\[ \text{lb}_i(t[1..j-1]) \leq \text{ub}_k(t[1..j-1]) \text{ for all } i \text{ and } k. \]

The result is a new system of inequalities in \( j-1 \) variables describing the projection of the polyhedron into its leading \( j-1 \) dimensions. This projection is itself a polyhedron. The algorithm then recurse, and bounds for the next level are computed in a similar manner with the remaining constraints, including those initially set aside for not involving \( t[j] \).

(FM4) The algorithm finishes when the bounds for the outermost loop level are computed.

Let us illustrate the procedure by generating code scanning the iteration domain illustrated in fig. 3.4. This is the version of the polynomial multiply kernel exposing coarse-grain parallelism. The aim is to find bounds of the form:

\[
\max_i \text{lb}_i \leq p \leq \min_k \text{ub}_k
\]

\[
\max_i \text{lb}_i(p) \leq t \leq \min_m \text{ub}_m(p).
\]

Recall the original iteration domain for \( T \) and the employed transformation are:

\[ \mathcal{D}_T = \{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i, j \leq N-1\} \]

\[
\begin{pmatrix}
  p \\
  t
\end{pmatrix}
= \begin{pmatrix}
  1 & 1 \\
  1 & 0
\end{pmatrix}
\begin{pmatrix}
  i \\
  j
\end{pmatrix}.
\]

Since the transformation matrix:

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

is unimodular, \( \det(T_T) = -1 \), it is invertible, \( \det(T_T) \neq 0 \), and we may write:

\[
\begin{pmatrix}
  i \\
  j
\end{pmatrix} = T^{-1}_T \begin{pmatrix}
  p \\
  t
\end{pmatrix} = \begin{pmatrix}
  0 & 1 \\
  1 & -1
\end{pmatrix}
\begin{pmatrix}
  p \\
  t
\end{pmatrix},
\]

giving \( i = t \) and \( j = p - t \), which, when substituted in the constraints defining \( \mathcal{D}_T \), determine:

\[ 0 \leq t, p-t \leq N-1. \]
FME works its way by initially eliminating iterator \( t \). First, we list the lower and upper bounds for \( t \):

\[
0 \leq t \leq N - 1
\]

\[
p - (N - 1) \leq t \leq p.
\]

From here, we can write the loop bound for \( t \) in terms of \( p \) (and the structure parameter \( N \)):

\[
\max(0, p - (N - 1)) \leq t \leq \min(p, N - 1).
\]

Next, eliminate \( t \), by combining every lower with every upper bound:

\[
0 \leq N - 1 \\
p - (N - 1) \leq N - 1 \\
0 \leq p \\
p - (N - 1) \leq p.
\]

The polyhedron is now described in terms of one fewer variable. Dropping redundant constraints and repeating the procedure for \( p \), we find its lower and upper bounds and set all other constraints aside:

\[
0 \leq p \leq 2N - 2 \quad N - 1 \geq 0.
\]

Finally, the respective loop trip for \( p \) is:

\[
0 \leq p \leq 2N - 2.
\]

The reader might refer to fig. 3.4 for an illustration of these results. Building the respective code still requires writing the array access indices as functions of the new iterators and choosing appropriate loop strides. The former use the expressions for the original iterators obtained earlier using the inverse transformation, \( i = t \) and \( j = p - t \). The latter are unitary, since, owing to the unimodular nature of the transformation matrix, there are no ‘holes’ in the new iteration space. The resulting (parallel) code is:

```c
for (int k = 0; k < 2*N - 1; k++)
c[k] = 0;       // S

#pragma omp parallel for
for (int p = 0; p <= 2*N - 2; p++)
for (int t = max(0, p - (N - 1)); t <= min(p, N - 1); t++)
c[p] = c[p] + a[t] * b[p - t];       // T
```

Listing 3.3: Coarse-grain parallel version of the product of two polynomials.
3.3.2 Modern Polyhedra Scanning

Subsequent work extended this FME-based framework to invertible but non-unimodular transformation matrices [Xue94, Ram95], i.e. those increasing the polyhedron’s volume, leaving ‘holes’ in the target polyhedron, requiring loops with non-unit stride. These developments used the lower triangular Hermite Normal Form [Sch86] of the transformation matrix to avoid scanning integers points with no matching instance in the initial space. Soon after, further work relaxed these requirements, allowing multiple target polyhedra simultaneously [KPR95] and non-invertible transformation matrices [GLW98]. An extension of FME for inequalities with loop iterators multiplying unknown symbolic quantities has also been used to generate parameterised tiled code [BHT+10, THB+10].

The main drawback with code generation methods based on FME is the potentially large amount of redundant control. Let there be \( d \) affine constraints defining the new, target polyhedron in \( \mathbb{N}^n \). Consider the worst case scenario, happening for an equal split of lower and upper bounds and no other constraints. This procedure creates \( d^2/4 \) new constraints after the first step and up to \( 4(d/4)^{2n} \) constraints by the time it terminates: this is double exponential complexity! [Lav19]. Many of these constraints are redundant and can be removed by linear programming to generate efficient control flow [LF96].

However, over the years, the focus has turned to directly generating efficient code without redundant control, instead of improving initial redundant specifications. The QRW algorithm [QRW00] with extensions by Bastoul [Bas04] is such an algorithm and is the current state-of-the-art. By taking advantage of high-level polyhedral operations based on polyhedral duality and Chernikova’s algorithm [LV92], e.g. projection but also union and intersection of polyhedra, it is able to generate efficient code directly. In fact, it is the polyhedral code generator of choice in production optimising compilers such as GCC [PCB+06], LLVM [GGL12] and IBM XL [RBD+09].

3.4 Polynomial Extensions: Semi-Algebraic Sets

The one-dimensional nature of channels in dataflow streaming languages, in particular OpenStream, might give rise to polynomial linearisation functions whenever a multi-dimensional sequence of objects is mapped onto them. Since access indices become polynomial, so do dependence sets. We review two mathematical theorems for the positivstellensatz, the algebraic description of all positive polynomials over a certain
3.4. POLYNOMIAL EXTENSIONS: SEMI-ALGEBRAIC SETS

These results play a similar role for polynomial inequalities as Farkas’ Lemma does for affine inequalities.

3.4.1 Handelman and Schweighofer’s Theorems

Let \( \{p_i\}_{i \in \{1 \ldots n\}} \) be a set of polynomials in \( d \) variables. The set:

\[
S = \{ x \in \mathbb{R}^d \mid \forall i, p_i(x) \geq 0 \}
\]

is a semi-algebraic set. Polyhedra are semi-algebraic sets for which all elements of \( \{p_i\} \) are of first degree. Sets of this nature will appear in chapter 4 as part of the dataflow dependence analysis for OpenStream programs. Problems such as showing a program has no dependence or building a schedule satisfying existing dependences can be solved, respectively, by emptiness and positivity checks over such sets. The following two theorems provide some answers.

**Handelman’s Theorem** [Han88] Let \( S \) be a compact polyhedron of the form (3.14) where all elements of \( \{p_i\} \) are of, at most, degree one. Define the monomial induced in the \( \{p_i\}_{i \in \{1 \ldots n\}} \) by \( k \in \mathbb{N}^n \) to be the polynomial in \( x \) that can be expressed as:

\[
p^k(x) = \prod_{i=1}^n p_i^{k[i]}(x).
\]

A polynomial \( f \) is strictly positive everywhere in \( S \) only if it can be written as:

\[
f(x) = \sum_{k \in \mathbb{N}^n} \lambda_k p^k(x), \quad \lambda_k \geq 0, \quad \sum_{k \in \mathbb{N}^n} \lambda_k > 0.
\]

Conversely, if a polynomial \( f \) can be written in such form, it is non-negative in \( S \) and strictly positive in the interior of \( S \).

**Schweighofer’s Theorem** [Sch02] Let \( S \) be of the form (3.14), allowing \( \{p_i\} \) of degree greater than one, provided the polynomials in \( \{p_i\} \) of degree one define a compact polyhedron. A polynomial \( f \) is strictly positive everywhere in \( S \) only if it has a representation (3.16). Conversely, if a polynomial \( f \) can be written in such form, it is non-negative in \( S \) and strictly positive in the interior of \( S \).
Caveats

Although these results have similarities to Farkas’ Lemma, there are some key differences, preventing them to acquire the same status [Yuk19]:

1. The domain $S$ must be compact and, therefore, bounded - albeit only a requirement in the ‘only if’ direction;

Most importantly, however:

2. Both Handelman’s and Schweighofer’s Theorems characterise polynomials with no roots in the interior of their domain $S$;

3. Representation (3.16), used by both theorems, does not set a limit on the degree of the monomials $p^k$. In practice, when applying these results, we bound the search space by an ‘order’ $M$ such that $\sum_i k[i] \leq M$.

Both these differences have the important side effect of potentially excluding polynomials of interest from the characterisable search space. Consider, for instance:

$$S = \{ x \in \mathbb{R} \mid p_1(x) \geq 0 \land p_2(x) \geq 0 \}, \quad p_1(x) = x + 1, \quad p_2(x) = 1 - x$$

$$= \{ x \in \mathbb{R} \mid -1 \leq x \leq 1 \}$$

and,

$$f_1(x) = x^2, \quad f_2(x) = 4x^2 + 1.$$ 

Polynomial $f_1$ has a root for $x = 0$, an interior point of $S$ and, therefore, cannot possibly be described by a combination (3.16). Polynomial $f_2$ on the other hand, despite being of degree two, requires extending the search space to, at least, $M = 5$: 

$$4x^2 + 1 = \left[ 8(1-x)^3 + 4(1-x)(x+1)^3 + 10(x+1)^4 + 2(1-x)(x+1)^4 + 5(1-x)^4(x+1) + 3(1-x)^5 \right] / 32.$$ 

Perhaps counterintuitively but as shown by this example, a polynomial of a given degree might require relatively high degree monomials, i.e. a high order $M$. This does not detract from the fact that $M$ bounds the maximum degree of the described polynomial. Here, since $S$ is a one-dimensional polyhedron, setting $M = 5$ prevents the description of any polynomial of degree six or above.
Thus, an unsuccessful attempt at proving positivity for a particular polynomial or an unsuccessful exploration for the coefficients of a polynomial template (e.g. when scheduling, see below), can only conservatively conclude the polynomial is not positive over the domain $S$ or that no such polynomial form is positive over $S$, respectively.

### 3.4.2 A Scheduling Algorithm

Feautrier [Fea15] adapted the scheduling algorithm for Farkas’ Lemma to semi-algebraic sets. The following is the algorithm we refer to in chapter 6 in the context of Schweighofer’s Theorem, even though its validity also holds for Handelman’s Theorem.

(S1) Given:

- A set of polynomials $\{p_i(a, b; p)\}_{i \in \{1...n\}}$ defining a semi-algebraic dependence relation $\mathcal{P}_{c,d}^{S \times T}$ (if a union, each component must be considered separately);
- A template for the difference of two one-dimensional polynomial transforms $t_{ST}(a, b; p)$ where some unknown coefficients $\{\mu_j, \nu_l\}$ appear linearly;
- An ‘order’ $M$ specifying the number of polynomials in the monomials $p^k$ over the $\{p_i\}$;

(S2) Apply Handelman/Schweighofer’s Theorem for polynomial products under $M$ and some polynomial upper ($\ast = +$) or lower ($\ast = -$) bound $\delta_{ST,c,d}(p)$:

$$\ast (\delta_{ST,c,d}(p) - t_{ST}(a, b; p)) = \sum_{k \in \mathbb{N}} \lambda_k p^k(x), \quad \lambda_k \geq 0, \quad \sum_{k \in \mathbb{N}} \lambda_k > 0;$$

(S3) Sum and equate to zero the coefficients $\{\mu_j; \nu_l; \lambda_k\}$ on the left and right hand sides corresponding to the same monomial on the components of $a$, $b$ and $p$;

(S4) Solve the resulting system for the $\mu$, $\nu$ and $\lambda$ using any convenient Linear Programming (LP) software. Take the floor of the resulting form.

Polynomial schedules, as those obtained through this procedure, are exactly what multidimensional affine schedules were designed to avoid [Fea92b]. Polynomial code generation is hard and still in its infancy, with notable early attempts in [CAK17, FC18].
Nonetheless, in the context of task-parallel dataflow programs, where the job of coordinating task execution is typically left to a dynamic scheduler and not to the programmer or a static tool, there is no immediate need to generate code implementing the new schedule. Indeed, we may simply be interested in knowing if there is a schedule for a specific set of task dependences, not necessarily what that schedule is. This is exactly the approach taken in chapter 6.

However, even when disregarding code generation, the problem of finding a one-dimensional polynomial schedule is still hindered by the current procedure. Since the algorithm is unable to find all non-negative functions over a semi-algebraic set, some valid schedules may never be identified irrespectively of how arbitrarily high $M$ is set. Consequently, we can at most give execution certificates upon a successful schedule search, but resort to conservative negative answers otherwise.

### 3.5 Summary

Program optimisation in the polyhedral model is, generally speaking, a succession of three phases. First, an amenable program part, a SCoP, is translated to an algebraic representation consisting of polyhedra describing the sets of dynamic statement instances, their execution order and the memory reference behaviour of their accesses. Next, an optimising algorithm computes a transformation in the model, e.g. maximising data reuse or parallelism, possibly describing a composition of multiple classical loop transformations. Finally, the algebraic abstraction is translated back to a computer program.

A good review of past and recent developments in the polyhedral model is given in [Bas12]. Amongst other issues, the document describes extensions to the model relaxing the static control constraints, e.g. by allowing data-dependent if conditionals or arbitrary loop bounds as typically found in while loops. Although these commonly involve conservatively approximating statements’ iteration domains or data dependences, they are shown to be beneficial in a range of use cases.

The task fusion algorithm in chapter 5 relies on two tools based on the polyhedral model: PLuTo [BBK+08, BHRS08], a source-to-source compiler for automatic parallelisation and locality optimisation used to find tiling hyperplanes as explained in section 3.2.3; and the state-of-the-art isl/barvinok library [VSB+07, Ver10a] to count the elements of, manipulate and project parametric affine sets and relations.
3.5. SUMMARY

In fact, the isl/barvinok library already supports some operations with polynomials. For example, the Bernstein expansion can be used to get bounds on multivariate polynomials defined over a parametric polyhedron [CFGV09]; and Barvinok’s rational functions [VSB+07] or Ehrhart polynomials [Cla96] can be used to compute the cardinal of a parametric polyhedron. Indeed, the latter are instrumental in statically obtaining stream access indices in OpenStream.

Finally, for polynomial scheduling in chapter 6, in addition to taking advantage of isl/barvinok for access index counts, we also implemented the algorithm in section 3.4.2 from [Fea15] and benefited from Feautrier’s own implementation that he kindly provided.
Chapter 4

OpenStream: Dataflow
Task-Parallelism

OpenStream is a task-parallel streaming dataflow language supporting the specification of fine-grained task, data and pipeline parallelism [PC13]. Its implementation as an extension to OpenMP C has morphed since OpenStream’s first incarnation [PC11, Pop11], resulting in an ever more expressive programming model [PC13]. Added expressiveness, however, usually comes at the expense increased static analysis complexity. Here, we avoid the syntax inherited from OpenMP and other possibly non-future-proof implementation details, and instead focus the discussion on the semantics of a subset of the allowed program specifications. We restrict the scope of this text to those OpenStream programs following the well-defined Control-Driven DataFlow (CDDF) model of computation [PC12]. We will accompany informal descriptions of the main concepts of CDDF OpenStream: streams, tasks and the control program, with the precise notation unlocking the algebraic code analysis and transformations that follow in the coming chapters. Next, we present the additional restrictions placed upon CDDF OpenStreams programs for compatibility with the polyhedral model. Finally, we review the technical solutions used for computing and handling task dependences which, despite the polyhedral restrictions, will be shown polynomial in the general case.

4.1 The Base Language

We describe a flavour of OpenStream closely resembling the semantics of the CDDF model with a few, small exceptions. On the one hand, we introduce a set of restrictions
imposed by the current runtime implementation which guarantee immediate compatibility, barring syntax, of the examples and code constructs provided here with OpenStream’s current implementation [PP20]. On the other hand, we suggest a few minor additions to CDDF which increase the productivity of the programmer and simplify code generation in the coming chapters. Moreover, we will make an effort to conciliate our description of OpenStream with the polyhedral notions introduced in chapter 3. For these reasons, we caution the reader for the slight discrepancies that may exist between the current text and previous reports [PC12, Dre15, CDF16, FC18].

OpenStream is a two-level language in which a sequential control program directs the initialisation of concurrent tasks that communicate through streams. We start with the latter.

4.1.1 Flowing Data: Streams and Tasks

Streams are one-dimensional arrays of indefinite size that work as the exclusive channels for data. A stream can be seen as a partial function, \( \mathbb{N} \rightarrow \mathcal{T} \), that at any instant during the execution holds values of some type \( \mathcal{T} \) over a finite subset of indices. Let \( \mathcal{S} \) denote the set of all streams.

In turn, streams themselves can be grouped in one-dimensional arrays, functions \( \mathbb{N} \times \mathcal{S} \), of an arbitrary size \( \mathcal{N} \). A stream behaves as a 0-ary stream array.

Conceptually, every stream \( s \) has a read position, \( r_s \), and a write position, \( w_s \), that define which elements of the stream are affected by subsequent read and write accesses. Each element of a stream is written using Dynamic Single Assignment (DSA), i.e. each position is written at most once so that no two write operations share the same \( w_s \). Elements that have not been written are undefined and remain inaccessible for read accesses. These policies — write-once, deferred reads — ensure no read-write races. All reads of the same stream position return a single, determinate value, even if after an arbitrary delay.

1. We will not use different font symbols or ranges of the alphabet to distinguish between streams and stream arrays, but will take care to avoid confusion whenever it might arise: e.g. a stream \( s \) vs. a stream array \( s \), for which \( s[i] \) would denote its \( i \)-th stream.

2. Here, for the sake of clarity, we explicitly name read and write positions for each stream. However, they can be entirely handled (and hidden, as in OpenStream [PC13]) by the compiler and runtime.

3. A program is in Static Single Assignment form, SSA (resp. Dynamic Single Assignment form, DSA), when there is at most one assignment for every variable or array location in its text (resp. at runtime). Particularly, contrary to SSA, DSA assures no reuse for assignments within iteration statements.

4. This behaviour resembles that of promises and futures [FW76, BH77], and I-Structures [ANP89]. Like the latter, streams are non-functional data structures — they are populated as the program unfolds — easing the development of functional task parallel programs without compromising determinism.
Tasks are the units of work: stateless functions transforming some input into some output. A task is a particular instance of a task creation statement. Each such statement $S$ is associated with an iteration domain $D_S$, the set of all tasks instances it spawns. These statements are composed of both:

(T1) A finite list of accesses on streams, $C_S$. This list may contain non-consuming read accesses or peeks, $p$, consuming read accesses, $r$, and write accesses, $w$. As we will soon see, this list effectively determines the task dependences. In fact, it is the sole synchronisation instrument within a task creation statement;

(T2) A body or work function: a sequence of instructions acting on local variables and on elements of the referenced streams which cannot include new task creation statements. The work function executes atomically: once a task starts executing, it runs to completion, there is no additional synchronisation.

We group the declared peek and consuming read accesses in the read accesses class, $R = \{p_x, r_x\}$, whereas write accesses are grouped in a class of their own, $W = \{w_x\}$. $x$ is a unique label distinguishing between declared accesses of the same type. It could be an index on the accesses’ list if $x \in \mathbb{N}$, or perhaps a more exotic descriptor using symbols in a set $V$ if $x \in V^*$, e.g. a somehow relatable string or, if redundant, just the empty string $\epsilon$. The set of all accesses in the program is $C = W \cup R$. Accesses are thus organised into three types, but only two classes. The list of read (resp. write) accesses in $S$ is then $R_S = C_S \cap R$ (resp. $W_S = C_S \cap W$).

Each access is described by a reference to a stream and a window connecting the task’s body to multiple stream elements. The former are formalised by stream access functions, $a_S^c(t) : D_S \rightarrow S$, which associate a unique stream in $S$ with each task in $S$ for a given access $c \in C_S$. We remark that, without loss of expressiveness, for the same task, no two accesses of the same class may reference the same stream. We say a task $t \in D_S$ is a consumer (resp. producer) on stream $s$ if there exists a read (resp. write) access $c \in C_S$ for which $a_S^c(t) = s$.

---

5 CDDF OpenStream programs written in the language presented in section 4.3 may, in fact, connect a task to multiple streams on the same declared access. This is achieved by defining an array of windows over an array of streams, using so called variadic clauses. Here, we avoid this slight complication by implicitly splitting the declared access into multiple single-stream accesses with distinct labels whenever possible, i.e. when the number of accessed streams is statically known. In future chapters, however, we will tacitly define stream access relations under the same access label whenever they are more convenient to describe multi-stream accesses.
4.1. THE BASE LANGUAGE

A window, \( w_S^c(t) \), determines what elements in stream \( a_S^c(t) \) are to be accessed by task \( t \). Windows are characterised by their first index on a stream, which can be automatically retrieved by the runtime, and a burst and a horizon, which are specified by the programmer:

(W1) The first index, \( f_S^c(t) : \mathcal{D}_S \rightarrow \mathbb{N} \), is a non-negative integer specifying the position of the first accessed stream element. For read (resp. write) accesses, it is set, by the runtime system, to the read (resp. write) position, \( r_s \) (resp. \( w_s \)), of stream \( s = a_S^c(t) \) at the time of task creation;

(W2) The burst, \( b_S^c(t) : \mathcal{D}_S \rightarrow \mathbb{N} \), is a non-negative integer specifying the amount by which the read, \( r_s \), or write position, \( w_s \), of stream \( s = a_S^c(t) \) is shifted after each task creation. Together with (W1), this implies that for the very first task \( t' \) created after \( t \) such that one of its accesses \( c' \) is of the same class as \( c \) and \( a_S'^c(t') = a_S^c(t) \), then \( f_S'^c(t') = f_S^c(t) + b_S^c(t) \). Since this effectively slides windows into position for succeeding accesses, we may call them sliding windows;

(W3) The horizon, \( h_S^c(t) : \mathcal{D}_S \rightarrow \mathbb{N} \), is a non-negative integer specifying the size of the window. Contrary to previous CDDF literature, this definition allows the specification of empty windows, i.e. we allow \( h_S^c(t) = 0 \), streamlining code generation in chapter 5. For a non-empty window, the window’s last index on a stream can be written \( l_S^c(t) = f_S^c(t) + h_S^c(t) - 1 \).

A peek is a read access with a burst of zero elements. In other words, a peek is a non-destructive read allowing data broadcasts: a burst of zero does not advance the stream’s read position, allowing successive consumers to colocate their windows, thereby reading the same stream elements.

Finally, the current OpenStream runtime implementation imposes some constraints on windows that we take into account for compatibility purposes. With the exception of guaranteeing the single assignment property on streams, these are simplifying assumptions that circumvent technical implementation issues which go beyond the scope of this discussion [PC13, Dre15]:

(C1) The burst of a read access window is either zero or equal to its horizon. A burst of zero corresponds to a peek access, while the latter describes a consuming read: there is no mixed-type read access;
(C2) The burst of a write access window is equal to its horizon, guaranteeing a) the CDDF (dynamic) single assignment property, since the burst is at least the horizon; and b) that only initialised stream elements are published, as the burst is at most the horizon;

(C3) The first index of any read access window must coincide with the first index of some write access window on the same stream. This guarantees no write event targets multiple non-colocated read windows. Note that the reverse does not need to hold.

Figure 4.1 depict how these concepts are used in OpenStream to express the flow of data between producer and consumer tasks.
4.1. THE BASE LANGUAGE

Figure 4.1: Dataflow patterns induced by the successive creation of tasks $t_{RW}$, $t_{W}$, $t_{P}$ and $t_{R}$. We assume each of these tasks is created by its own task creation statement, such that the iteration domain of each such statement is a singleton. This allows us to name tasks after their creation statements, $D_{S} = \{t_{S}\}$, and to use functions as constants.

Steps 1 and 6 show, respectively, the initial and final states of streams $s_{1}$ and $s_{2}$. For each task created in step $n \in \{2, 3, 4, 5\}$, their windows’ first indices are assigned to the relevant stream’s read or write positions at that instant. The latter are subsequently updated in step $n + 1$ using the window bursts. Notice how $t_{RW}$’s produce is broadcast to both $t_{P}$ and $t_{R}$ as a result of $t_{P}$’s non-destructive, zero-burst read.

4.1.2 Spawning Concurrency: The Control Program

The core of our model is the control program, whose role is akin to that of the main program in OpenMP or Cilk: spawning tasks. The control program is a piece of imperative, sequential code, expanded with task creation statements.

The first important insight is that the order in which the control program encounters
the task creation statements fully defines the order of task creation, which we denote \( \prec \). This order is both strict and total. We can now deduce the order in which the underlying control program encounters the different task creation sites: \( t_{\text{RW}} \prec t_{\text{W}} \prec t_{\text{P}} \prec t_{\text{R}} \).

The second is that the arithmetic relationship stated in (W2) between window bursts and first indices of successively created tasks depends solely on the task creation order. This implies that the control program dynamically but — because it describes a sequential strand of execution — *deterministically* dictates the position of each task’s windows on the program’s streams. This order of attribution of indices in streams is called the schedule of access indices, a central element of CDDF which opens the door for static program analysis. Actually, the window placement in each step of fig. 4.1 took advantage of these properties\(^6\).

The programmer is free to tailor the control program to their needs:

1. Several flows of control may be implemented, allowing arbitrarily predicated stream and task creation;
2. OpenStream supports barrier synchronisation. Once a barrier is encountered by the control program, it is suspended until all tasks created thus far terminate;
3. There are no constraints on the amount of work performed by the control program. Nevertheless, we tend to forgo this functionality since a) by Amdahl’s law, it reduces the efficiency of the program by increasing its sequential fraction and b) since it is often possible to embrace the streaming spirit of the language by moving this code into tasks and using streams to distribute the results.

### 4.2 Dependence Analysis

We now study task dependences and their influence on task execution.

#### 4.2.1 Dataflow Dependences

Since the read and write window indices are deterministically known for every task, we can statically determine which task produces the stream elements consumed by another task. These producer-consumer relationships determine (indeed, are) the task *dataflow dependences*.

\(^6\)Consider an OpenStream control program such that \( t_{\text{P}} \prec t_{\text{R}} \prec t_{\text{RW}} \prec t_{\text{W}} \). What would the dataflow patterns look like? As a matter of fact, just like those in fig. 4.1. This is always true as long as, for each stream, the order of the reads and, independently, the writes remains unchanged.
4.2. DEPENDENCE ANALYSIS

A task $t_{\text{prod}} \in D_S$ is a producer for task $t_{\text{cons}} \in D_{S'}$ if and only if at least one of the non-empty write windows in $t_{\text{prod}}$ falls within a non-empty read window in $t_{\text{cons}}$: \[ \exists c \in W_S, \exists c' \in R_{S'}, a_c^S(t_{\text{prod}}) = a_{c'}^{S'}(t_{\text{cons}}) \land h_c^S(t_{\text{prod}}), h_{c'}^{S'}(t_{\text{cons}}) \geq 1 \land f_c^S(t_{\text{cons}}) \leq f_c^{S'}(t_{\text{prod}}) \land l_c^S(t_{\text{prod}}) \leq l_{c'}^{S'}(t_{\text{cons}}). \] (4.1)

The dependence set between statements $S$ and $S'$, associated with accesses $c \in W_S$ and $c' \in R_{S'}$ is then naturally defined as a binary relation over $D_S$ and $D_{S'}$: \[ D_{c \rightarrow c'}^{S \rightarrow S'} = \{ t_{\text{prod}} \rightarrow t_{\text{cons}} \in D_S \times D_{S'} \mid a_c^S(t_{\text{prod}}) = a_{c'}^{S'}(t_{\text{cons}}) \land h_c^S(t_{\text{prod}}), h_{c'}^{S'}(t_{\text{cons}}) \geq 1 \land f_c^S(t_{\text{cons}}) \leq f_c^{S'}(t_{\text{prod}}) \land l_c^S(t_{\text{prod}}) \leq l_{c'}^{S'}(t_{\text{cons}}) \}. \] (4.2)

Similarly to the corresponding classical definition in (3.8), we assume no stream name aliasing. The stream access functions and the horizons can be extracted from the program’s text. The last indices follow from (W3). Finally, since the first index of any access window only depends on the bursts of prior task creations accessing the same streams, a generalisation of the window sliding mechanics in (W2) yields \[ f_c^{S'}(t') = \sum_{S, (t, c) \in Pr_S(t', c')} b_c^S(t) \] where the set of contributing task access pairs is \[ Pr_S(t', c') = \{ (t, c) \in D_S \times C_S \mid t \prec t' \land (c, c') \in W^2 \cup R^2 \land a_c^S(t) = a_{c'}^{S'}(t') \}. \] (4.3)

4.2.2 Barriers

Besides specifying the dataflow dependences, a programmer might use barriers. The presence of barriers does not change anything in the task creation $\prec$ order, hence has no impact on the counts in (4.3), and does not change the stream dataflow dependences in (4.1).

However, since the control program is suspended, awaiting termination of all earlier created tasks, new control-driven dependences are induced. A task $t'$ depends on $t$ via barrier $b$ if and only if the control program encounters the barrier in between the task creations: \[ t \prec b \prec t'. \] (4.4)

This differs from the dependence condition in [CDF16] as it takes into account constraint (C3), preventing the same write window from overlapping multiple non-colocated read windows.
CHAPTER 4. OPENSTREAM: DATAFLOW TASK-PARALLELISM

4.2.3 Task Graphs

The dependence relations just defined can be depicted in the task graph. Its vertices represent the task instances in the program and its directed edges, the dataflow and control-driven dependences. For any two tasks in the OpenStream program, we draw an edge between them if and only if they verify either (4.1) or (4.4).

As OpenStream programs allow the predicated creation of an arbitrary number of streams and tasks dynamically at runtime and as the instructions creating them can be fully embedded into the control-flow of the application’s control program, this graph cannot, in general, be determined statically. We say the control program facilitates the dynamic construction of task graphs as it unfolds and encounters task creation statements, in turn unveiling tasks’ and streams’ dynamic relationships.

Figure 4.2 depicts the history of the task graph for the example stream accesses shown in fig. 4.1.

![Figure 4.2: Evolution of the task graph showing the dependence patterns in fig. 4.1 as the different tasks are created. This depiction assumes no barrier in the control program.](image)

4.2.4 Happens-Before Relation and Execution Schedules

A task becomes ready for execution, i.e. is activated, when all its producers terminate their execution, thereby satisfying its dataflow dependences, and can be executed as soon as it is selected by the runtime scheduler. As a producer must always execute before its consumers, the transitive closure of the task dependences defines the tasks’ happens-before relation, a strict order. In general, and contrary to task creation, this order is only partial, exposing concurrency and potentially enabling parallel execution on a machine with multiple processing elements. Figure 4.3 shows a viable task execution trace for the running example.

A schedule is a function, \( \theta_S : D_S \rightarrow \mathbb{N} \), that to each task instance created by \( S \) associates a non-negative integer representing a logical time stamp for execution at
Figure 4.3: Possible execution sequence for the tasks illustrated in fig. 4.1. The task graph in fig. 4.2 forces $t_{RW}$ to succeed $t_W$, but does not connect $t_P$ and $t_R$. These two tasks may, therefore, execute in parallel.

runtime. The schedule is closely related to the happens-before order: if task $t' \in D_S$ depends, even if transitively, on task $t \in D_S$, then $\theta_S(t') \geq \theta_S(t) + 1$. The schedule then prescribes a (strict and partial) execution order for tasks at least as 'strong' as the happens-before relation. In other words, the schedule’s induced order must be a superset of the happens-before relation.

Studying this partial execution order will be the object of the coming chapters, whether by restricting it, by means of tuning execution schedules, or by verifying its feasibility, by finding valid execution schedules, guaranteeing the absence of deadlocks.
4.2.5 Deadlocks

According to [PC12], we say a CDDF OpenStream program deadlocks if it can make no progress. Thus, it must be the case that the control program is either suspended on a barrier or has terminated and the only outstanding, i.e. not yet executed, created tasks cannot be activated by the runtime due to lack of input data on any of their read access windows.

This definition does not preclude the existence of infinite dependence paths, e.g. cycles, on the task graph. In fact, it is possible to conceive programs that do not deadlock according to this definition and, yet, have cycles. This is feasible for infinite (but never finite) task graphs such as the one in fig. 4.4.

![Figure 4.4: Task graph including an inexecutable self-dependent task. Besides creating the pathological task, the control program also creates an infinite number of other producers and consumers. This pattern guarantees not only the continued execution of the control program but also the continued existence of activated tasks, ready to execute.](image_url)

An alternative, stricter definition for OpenStream deadlocks, found in [CDF16], only requires the existence of a task that can never be activated or, equivalently, the lack of a schedule for all tasks, thus including task graphs such as that in fig. 4.4. In the same publication, the authors proved it is in general undecidable (as a variant of Hilbert’s tenth problem) to detect if an OpenStream program deadlocks. Thus, the converse problem, on concluding on the existence of a schedule must also be undecidable. We will adopt the deadlock definition in [CDF16] in the remainder of this text.

4.2.6 Absence of Consumers or Producers

The construction in (4.1) does not capture consumers of stream elements with no producer. It would be incorrect, however, to assume these tasks have no dependence with respect to such access. In fact, while for classical, imperative programs a read on an uninitialised memory location is executed anyway, possibly returning garbage, an OpenStream reader should wait forever, resulting in a deadlock situation.

Determining whether a particular OpenStream program is ill-formed by lack of
producers amounts to determining if, for each stream $s$, its largest $w_s$ is greater than or equal to its largest $r_s$. Although performing this check is undecidable in general [CDF16], conservative algorithms can be given to provide static guarantees in some cases, such as the polyhedral subset of OpenStream, defined in section 4.4. Henceforth, we exclude this situation and assume all stream elements within the read access window of a task also belong to the write access window of some task.

We also assume the converse situation, i.e. the absence of a consumer for a produced stream element, does not occur. While not relevant for establishing dataflow dependences, it would have a similar status for the back-pressure dependences to be defined in chapter 6.

4.3 A Note on Syntax

In order to evade the semantical specifics of OpenStream’s current implementation as an extension to OpenMP C, we describe the control program in a classical, imperative, C-like language based around assignments and the flow of control as prescribed by the textual sequence, conditional and iteration statements. Support for CDDF OpenStream relies on simple extensions to the language:

\begin{align*}
(1) \quad access & := access; \text{access} | \text{stream, win}(\text{horizon}) \\
& \quad | \text{stream}\_\text{array}[\text{index}], \text{win}(\text{horizon}) \\
& \quad | \text{stream}\_\text{array}, \text{win}\_\text{array}(\text{horizon}) \\
(2) \quad clause & := \text{clause}\_\text{clause} | \epsilon \\
& \quad | \text{peek}(\text{access}) | \text{read}(\text{access}) | \text{write}(\text{access}) \\
(3) \quad lhs & := \text{stream}\_\text{array}[\text{index}] \\
& \quad | \text{win}[\text{elem}] | \text{win}\_\text{array}[\text{index}][\text{elem}] | ... \\
(4) \quad expr & := expr \oplus expr | \text{stream} | \text{stream}\_\text{array}[\text{index}] \\
& \quad | \text{win}[\text{elem}] | \text{win}\_\text{array}[\text{index}][\text{elem}] | ... \\
(5) \quad stmt & := \text{stmt}; \text{stmt} | \text{stream}\_\text{stream} | \text{stream}\_\text{stream}\_\text{array}[n\_\text{streams}] \\
& \quad | lhs = expr | \text{task}\_\text{clause} \text{do stmt} | \text{barrier} | ... \\
(\ast) \quad \text{stream} | \text{stream}\_\text{array} | \text{win} | \text{win}\_\text{array} := \text{identifier} \\
& \quad \text{index} | \text{horizon} | n\_\text{streams} | \text{elem} := expr
\end{align*}
Our extension to the original imperative language consists in the addition of rules (1) and (2) defining a syntax for connecting tasks to streams using windows; in introducing declarations for streams and stream arrays in rule (5); in allowing assignments of streams to stream arrays, and assignments from and to window elements via rules (3) to (5); and in adding the last two constructs in rule (5), providing the task and barrier constructs. The last options in (1) and (4) unlock the multi-stream, variadic accesses discussed in footnote 5, albeit with the implicit minor limitation that all streams in the array must utilise windows with the same horizon (and burst). This implies no loss of generality since we can subsample an existing stream array into new arrays.

As mentioned at the beginning of section 4.1.1, we alleviate the notation by allowing stream and stream array identifiers to belong to the same set. We also ease the notation on window identifiers in a similar manner. Since a read (resp. write) window’s first index is essentially a shadow of the stream’s read (resp. write) position, we follow the suggestion in footnote 2: we omit it and relegate its management to some compiler or runtime mechanism. The burst is omitted since it can be inferred from the horizon and constraints (C1) and (C2). The horizon can also be optionally omitted, in which case it is assumed one. Finally, we assume the existence of a context sensitive semantics analyser with the ability to rule out syntactically correct, but semantically invalid statements. For instance, as outlined in item (T2), task statements may not nest other task statements or barriers.

A possible control program with minimal task bodies for the example in figs. 4.1 and 4.2 can be specified as follows:

```plaintext
1 stream s1, s2;

3 task read {s2, win_r(5)} write {s1, win_w(4)} do // RW
4 for (int i = 0; i < 4; ++i)
5   win_w[i] = -i;

7 task write {s1, win_w1(6); s2, win_w2(5)} do // W
8 for (int i = 0; i < 6; ++i)
9   { win_w1[i] = -i + -4; if (i < 5) win_w2[i] = -i; }

11 task peek {s1, win_p(4)} do ; // P
12 task read {s1, win_r(10)} do ; // R
```

Listing 4.1: Example control program for the dataflow patterns illustrated in fig. 4.1. As discussed in section 4.1.2, careful reorderings of the task creation statements are possible without changing the semantics of the program.
4.4 The Polyhedral Fragment of the OpenStream Language

The polyhedral fragment of CDDF OpenStream is defined by a set of restrictions on the control program which, if entirely compliant, is termed an OpenStream Static Control Program (SCoP). As in chapter 3, we consider unions and projections of polyhedra:

(P1) Control statements are restricted to selection statements and counted loop iteration statements which must be, respectively, predicated by and bounded by quasi-affine expressions (on the structure parameters and outer loop iteration counters). This implies that a) for any task creation statement \( S \), \( \mathcal{D}_S \) is a polyhedron in \( \mathbb{Z}^n \) where \( n \) is the number of loops surrounding \( S \)\(^8\) and b) that \( \prec \) is a disjunction of quasi-affine constraints specifying the lexicographic and textual (see section 3.1.2 for AST-labelled schedules) orders between two task creations;

(P2) Stream array indices on task creation statements’ access lists must be represented by quasi-affine expressions — this applies semantically to the stream access functions in section 4.1.1 and syntactically to \( \text{index} \) on grammar rule (1), section 4.3;

(P3) No stream aliasing in different stream arrays, guaranteeing a stream can be identified uniquely, either as \( \text{stream} \) or \( \text{stream\_array}[\text{index}] \) — see grammar, section 4.3;

(P4) No variadic accesses on variable length stream arrays. This way, the trick in footnote 5 to treat these as a collection of single-stream accesses still applies, meaning they require no special treatment for dependence analysis either.

The polyhedral fragment of OpenStream places no restrictions on work functions or on horizons (or bursts) of stream access windows.

4.5 The Case for Polyhedral Dependence Sets

Given (P1), any dependence relation (4.2) is contained in \( \mathbb{Z}^n \times \mathbb{Z}^{n'} \), where \( n \) (resp. \( n' \)) is the number of loops surrounding \( S \) (resp. \( S' \)). However, dependence sets of this form

\(^8\) Nonetheless, tasks keep their non-bold typeface, e.g. \( t, t', t_S, ... \). Also, if the number of loops is zero, i.e. if \( n = 0 \), then \( \mathcal{D}_S \subseteq \mathbb{Z}^0 \). If \( \mathcal{D}_S \) is non-empty (e.g. \( S \) is not part of a selection statement with a trivially false condition), then it is a singleton, \( \mathcal{D}_S = \{ t_S \} = \{ () \} \).
are not necessarily unions of polyhedra but of (possibly polyhedral) semi-algebraic sets — even if we require quasi-affine horizons (and, thus, bursts).

Since (P1) and (P2) guarantee that iteration domains are polyhedral, that $\prec$ is a disjunction of quasi-affine predicates and that the access functions are quasi-affine, a window’s first index, as given by (4.3), may be easily computed by familiar techniques, e.g. Ehrhart polynomials [Cla96] or Barvinok’s rational functions [VSB+07], readily available in isl/barvinok. The result will, in general, be a non-linear polynomial. Consider, for example, the following SCoP fragment:

```plaintext
stream s;
for (int i = 0; i < N; ++i)
  for (int j = 0; j < i; ++j)
    task write (s, win_w) do; // S
Listing 4.2: SCoP fragment expressing a polynomial first index on stream s.
```

Since there is a single task creation statement $S$ with a single access, here named $w$, we may simply write:

$$
\mathcal{D}_S = \{(i, j) \in \mathbb{Z}^2 \mid 0 \leq i < N \land 0 \leq j < i\},
$$

$$
\forall t \in \mathcal{D}_S. a^S_w(t) = s, b^S_w(t) = h^S_w(t) = 1,
$$

$$
\forall t, t' \in \mathcal{D}_S. t \prec t' \iff (i, j) \prec (i', j') \iff (i < i' \lor i = i' \land j < j').
$$

From which the first index of every instance of $S$ may be easily computed:

$$
f^S_w(t') = \sum_{t \in \mathcal{D}_S \atop t \prec t'} b^S_w(t) = \sum_{i=0}^{i'-1} \sum_{j=0}^{j'-1} 1 + \sum_{j=0}^{j'-1} 1 = (i' - 1)/2 + j'.
$$

As the count is a second degree polynomial on the loop iteration counters, any dependence set of the form in (4.2), built for an appropriate extension of listing 4.2 including some consumer tasks, will not be a polyhedron. This behaviour is common whenever a multidimensional array is mapped into linear memory, producing polynomial linearisation functions, and undermines the use of state-of-the-art tools such as isl/barvinok for further analysis or code transformations. There are two possible solutions. In chapter 6, for no loss of generality, we use the techniques outlined in section 3.4 to cope with semi-algebraic sets. In chapter 5, however, since the aim is to provide a readily implementable task fusion algorithm, we avert the problem by solely focusing on polyhedra.
4.5. THE CASE FOR POLYHEDRAL DEPENDENCE SETS

In chapter 5 a task fusion algorithm for CDDF OpenStream programs is presented. The following outlines an extra set of restrictions imposed on the input programs such that their dependence sets are unions of polyhedra, in turn allowing transformations to be entirely devised with immediately available algebraic tools such as isl/barvinok.

Since (P2) already enforces affine stream array indices, the only problematic quantities in (4.1) and (4.2) are the access window properties. We require the following to be writable as quasi-affine forms:

(R1) The horizons of the windows;

(R2) The first indices of the windows. Window constraints (C1) and (C2) and restriction (R1) ensure the bursts are quasi-affine. As witnessed by listing 4.2, however, this is certainly not a sufficient condition to guarantee (4.3) is quasi-affine. Restricting bursts to known constants and considering control programs with either i) solely one-dimensional loop iteration statements or ii) rectangular loop statements with known bounds would suffice [CDF16], but this is far too restrictive:

(a) It is clear bursts need not be numerical constants. The following SCoP shows it is not a necessary condition:

```
stream s;

task write (s, win_w(h)) do ; // W

task write (s, win_w(h)) read (s, win_r(h)) do ; // WR

task read (s, win_r(h)) do ; // R
```

Listing 4.3: SCoP containing three tasks with symbolic horizons (thus bursts) on stream s but affine window first indices.

We have $f^W_W = f^WR_R = 0$ and $f^WR_W = f^R_R = h$, which are affine;

(b) Similarly, considering control programs with i) solely one-dimensional loop iteration statements or ii) known bounds is not a necessary prerequisite:

```
stream s[N];

for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
        task write (s[i], win_w) do ; // W
```

Listing 4.4: SCoP fragment with a two-dimensional iteration statement expressing affine first indices on each stream $s[i]$ of stream array s.

In this case, $f^W_W(i, j) = j$, despite the two-dimensional iteration space and the parametric bounds.
Ideally, our approach should reject examples such as listing 4.2 while still allowing the multi-dimensional and parametric control flow in listing 4.4. One possible procedure would use a syntactic pass to filter out the troublesome programs through a set of necessary and sufficient conditions. Finding these, however, might be a daunting task since the enumerator in (4.3) depends non-trivially on which loop variables intervene in the form of the (quasi-affine) access functions or on which of these variables are parametrically bounded.

Nevertheless, we need not be concerned about such issues. In practice, we can simply use the machinery already in place for tractable input programs, directly compute the first indices through (4.3), inspect the form of the resulting count and reject non-conforming programs.

(R3) The last indices of the windows. This follows from (R1) and (R2) and the definition of a window’s last index in (W3).

Arguably, the previous set of restrictions is quite limiting. The hope is that, in the pursuit for code simplicity, the programmer will naturally tend to untangle task communication by using more streams laid out in possibly multiple stream arrays, in turn inducing less complex, perhaps non-polynomial stream indexing. This suggests that, as a simple rule of thumb, the more explicit and natural the original OpenStream implementation is, the more likely it is to be amenable to task fusion through the procedure in chapter 5.

Lastly, we remark that OpenStream programs fulfilling all these constraints are unlike traditional, imperative, polyhedral programs where a region of interest is delimited by a pair of annotations, say \#pragma scop to \#pragma endscop, indicating the compiler may ignore the surrounding code and still extract meaningful dependence information. On the contrary, a window’s first index depends on the history of the stream it connects to. A previous code region might offset read and write positions by different, possibly non-affine extents which must be taken into account. In addition, any transformation that reorders task creations might reorder data in the streams and, therefore, breach the assumptions of neighbouring code. For these reasons, we will abstain from using \#pragma (end)scop which could perhaps cause some confusion to the avid polyhedral user.
4.6 Summary

OpenStream is a state-of-the-art language extension to OpenMP enabling high-performance task-parallel programming [PC13]. Control-Driven DataFlow [PC12] is a model of computation capturing the semantics of a deterministic subset of OpenStream.

The general concept of specifying point-to-point data dependences between tasks is a common and growing trend for task-parallel languages [PBAL09, Ope13]. Similarly, control-driven dataflow approaches have been employed elsewhere to express dynamic task graphs [BBC+10].

From a more formal point of view, OpenStream shares strong similarities with Kahn Process Networks (KPNs) [Kah74]. However, OpenStream and KPNs have subtle differences and the equivalence of the models is still not fully understood [CDF16]. OpenStream’s unique support for multiple producers and consumers on the same one-dimensional stream is unlike KPNs and means the specifics of stream accesses using windows and deterministic matching of producers and consumers are very particular to its model of computation. Furthermore, unlike OpenStream, KPNs, which can represent Boolean DataFlow (BDF) graphs, allow conditional decisions on what streams to read from (and write to) based on the data that circulates on the streams. Unfortunately, as a result of these differences, it is not clear how to transfer important techniques known for KPNs onto OpenStream, most notably regarding the boundedness execution problem we visit in chapter 6.

It is clear OpenStream’s expressiveness is sufficient to implement simple but popular DataFlow Graphs (DFGs) such as Synchronous DataFlow (SDF) or Cyclo-Static DataFlow (CSDF) graphs. However, the literature is rich on work regarding stream boundedness in this realm [LM87b, BLM96, BMMU10, BMK13]. Consequently, and since for DFGs of this class stream indexing is, at most, quasi-affine, we expect most interesting future work following on this thesis footsteps to instead explore the task fusion technique from chapter 5 in SDF and CSDF applications.

On the bright side, dependence analysis for OpenStream does not involve any consideration regarding any task’s work function. We leverage the stream access information the programming model requires from the user to treat tasks as black boxes, allowing polyhedral analysis even for programs whose tasks include non-amenable data access patterns. As we will soon see in chapter 5, such remark does not mean we are always free from manipulating task bodies, in particular when code generation is required, e.g. to fuse the work functions of a group of tasks. It does mean, however, that any such transformation may be completely oblivious of internal access patterns.
Nonetheless, since tasks’ work functions are just traditional, imperative (frequently compound) statements, the well-known polyhedral techniques from chapter 3 might provide immediate solutions to improve the program’s performance at this level.
Chapter 5

Scheduling for Granularity Control

Tiling, as reviewed in section 3.2.3, coalesces multiple instances of a statement which is executed repeatedly in a loop program. In this chapter, we build on this idea to fuse tasks spawned by a task creation statement within a loop statement. The initial idea of task fusion in our setting is the same as in traditional tiling based on the polyhedron model, but its technical implementation has been extended to handle the streaming dataflow semantics of OpenStream.

We present a sequence of steps describing a compilation technique that given an OpenStream program produces a new OpenStream program with coarser parallelism. It starts with an OpenStream SCoP, i.e. an OpenStream specification falling into the restrictions of the polyhedral subset of OpenStream defined in section 4.4, further constrained to only express polyhedral dependences as discussed in section 4.5. Next, it statically fuses both tasks and stream communication, aiming at improving both the work-to-synchronisation and work-to-data balances of the given OpenStream program.

We identify some limitations of the current implementation and give pointers for improvement as part of future work. In addition, immediately after each step, we demonstrate the relevant concepts, mathematical objects and transformations on a naive OpenStream implementation of Gauss-Seidel 1d. We start by giving an overview of the procedure.

Overview After determining the task dataflow dependences, these are passed onto a state-of-the-art source-to-source polyhedral compiler that manipulates the control program as if a traditional imperative control flow program was at hands. The task directive is transferred outside the intra-tile loop nest and new access clauses are built
by associating new coalesced, tile-to-tile streams to each dependence polyhedron. Finally, the new fused task body is reworked to combine both the work of the original fine-grained tasks and their communication responsibilities.

5.1 Tiling the Control Program

Like other polyhedral source-to-source compilers, the current technique solely reworks the code enclosed between pairs of delimiters, here `#fuse` and `#endfuse`, leaving other source code untouched. Such target code regions may be imperfectly nested in loop statements but must contain a single, perfectly nested task creation statement\textsuperscript{1}.

```
void GS1D(double *P) {
    stream s[N];
    for (int j = 0; j < N; ++j)
        task write (s[j], win_w) do // SRC
            win_w = P[j];
    #fuse
    for (int i = 0; i < I; ++i)
        for (int j = 1; j < N - 1; ++j)
            task \ 
                peek (s[j-1], win_pl; s[j+1], win_pr) \ 
                read (s[j], win_r) write (s[j], win_w) do // WRK
                    win_w = (win_pl + win_pr) / 2;
    #endfuse
    for (int j = 0; j < N; ++j)
        task read (s[j], win_r) do // SNK
            P[j] = win_r;
    barrier;
}
```

Listing 5.1: OpenStream implementation of Gauss-Seidel 1d with a sequential and static control program including a single and perfectly nested task creation statement.

\textsuperscript{1}The issue of considering multiple, imperfectly nested task creation statements with different iteration domains is left for future work.
2. Extract the iteration domains for the target statement and for all the surrounding task creation statements that access the same stream arrays. The former defines the general scope within which tasks are to be fused while the latter defines the scope of the dependences to be considered by this procedure. Our algebraic description of these statements is then completed by directly parsing their windows’ horizons and bursts and their stream access functions.

We may, by direct examination of the control program, write the iteration domains for task creation statements SRC, WRK and SNK as:

\[
\begin{align*}
\mathcal{D}_{\text{SRC}} &= \{ j \in \mathbb{Z} \mid 0 \leq j \leq N - 1 \}, \\
\mathcal{D}_{\text{WRK}} &= \{ (i, j) \in \mathbb{Z}^2 \mid 0 \leq i \leq I - 1 \land 1 \leq j \leq N - 2 \}, \\
\mathcal{D}_{\text{SNK}} &= \{ j \in \mathbb{Z} \mid 0 \leq j \leq N - 1 \}. 
\end{align*}
\] (5.1)

These are depicted on the left of fig. 5.1.

We identify three read and one write accesses for each task instance \((i, j) \in \mathcal{D}_{\text{WRK}}\). These all happen on the same stream array \(s\) and will be designated \(p_l\), for the left-sided peek on stream \(s[j-1]\), \(r\), for the consuming read on \(s[j]\), \(p_r\), for the right-sided peek on \(s[j+1]\), and \(w\), for the write on \(s[j]\). These are summarised by the following stream access functions:

\[
\begin{align*}
a_{p_l}^{\text{WRK}}(i, j) &= s[j-1], \\
a_r^{\text{WRK}}(i, j) &= a_w^{\text{WRK}}(i, j) = s[j], \\
a_{p_r}^{\text{WRK}}(i, j) &= s[j+1].
\end{align*}
\] (5.2)

These are depicted in the context of stream array \(s\) on the right of fig. 5.1. Similarly, we can write the stream access functions for the single write access in SRC and the single read access in SNK:

\[
a_w^{\text{SRC}}(j) = a_r^{\text{SNK}}(j) = s[j].
\] (5.3)

In listing 5.1, the horizons and bursts of all windows are implicitly given. This means that the horizon is unitary, \(h_c^S = 1\), regardless of the access \(c\) and the statement \(S\) and that the burst is unitary for the reads and writes, \(b_r^S = b_w^S = 1\), but null for the peeks, \(b_{p_l}^{\text{WRK}} = b_{p_r}^{\text{WRK}} = 0\).

3. Valid tiling transformations must not disrupt any task dependence. To find them, we
first need to determine where each task’s read — peek or consuming read — and write windows are located in the accessed streams. These windows may be identified by their first and last indices, just as explained in section 4.2. We may use the isl/barvinok library to compute peek, read and write first indices for each stream accessed in each stream array. These must comply with the restrictions in section 4.5. Otherwise, the dependence sets in step 4 will not be polyhedral and the use of isl/barvinok for future projections and cardinal computations will not be an option.

The peek and read windows’ first indices for tasks created by WRK are solely dependable on previously declared consuming reads on the same stream, see (4.3). For this problem, since all non-zero bursts are unitary, summing bursts is the same as counting all those consuming tasks. In particular, every task \((i', j') \in D_{WRK}\) reads from \(a_r^{WRK} (i', j') = s[j']\). Thence, we need only count those for which \(s[j']\) coincides with the stream accessed by the task instance of interest \((i, j)\). These counts can be
expressed as (quasi-)affine selection trees (or quasts) [Fea91]:

$$ f^{WRK}_p(i, j) = \text{Card} \{ (i', j') \in D^{WRK} \mid (i', j') \prec (i, j) \land a^{WRK}_p(i', j') = a^{WRK}_p(i, j) \} $$

$$ = \text{Card} \{ (i', j') \in D^{WRK} \mid (i', j') \prec (i, j) \land s[j'] = s[j-1] \} $$

$$ = \text{Card} \{ (i', j') \in D^{WRK} \mid (i < i' \lor i' = i \land j' < j) \land j' = j-1 \} $$

$$ = \text{if} \ j \geq 2 \ \text{then} \ i + 1 \ \text{else} \ 0, $$

$$ f^{WRK}_r(i, j) = \text{Card} \{ (i', j') \in D^{WRK} \mid (i', j') \prec (i, j) \land a^{WRK}_r(i', j') = a^{WRK}_r(i, j) \} $$

$$ = i, $$

$$ f^{WRK}_p(i, j) = \text{Card} \{ (i', j') \in D^{WRK} \mid (i', j') \prec (i, j) \land a^{WRK}_p(i', j') = a^{WRK}_p(i, j) \} $$

$$ = \text{if} \ j \leq N - 3 \ \text{then} \ i \ \text{else} \ 0. $$

(5.4)

The write index for the same tasks can be obtained in a similar fashion by counting the writes preceding the current task:

$$ f^{WRK}_w(i, j) = \text{Card} \{ j' \in D^{SRC} \mid a^{SRC}_w(j') = a^{WRK}_w(i, j) \} $$

$$ + \text{Card} \{ (i', j') \in D^{WRK} \mid (i', j') \prec (i, j) \land a^{WRK}_w(i', j') = a^{WRK}_w(i, j) \} $$

$$ = 1 + i. $$

(5.5)

The tasks that contribute to these counts, i.e. those belonging to the sets we are taking the cardinals of, are depicted on the left of fig. 5.2. Finally, since all horizons are unitary, every window’s last index matches its first index, $$ l^{WRK}_c = f^{WRK}_w, $$ irrespectively of the particular access $$ c. $$ Together, they define one-element stream access windows, $$ w^{WRK}_c, $$ which are shown on the right of fig. 5.2.

4. For each stream array, determine the task dataflow dependences within the target region, #fuse to #endfuse. There is a dependence when, on the same stream, a read (peek or consuming read) window envelops the write window of another task, as per (4.1). We only admit uniform dependence patterns, i.e. those characterised by constant dependence distance vectors, as required by steps 7 and 8a).

Consider the loop nest whose body contains the target creation statement $$ S. $$ Let $$ n $$ be its full depth and $$ m $$ the first dimension of said loop lying inside the #fuse and #endfuse delimiters. From section 3.2.1, a dependence relation over $$ S $$ at and below
Figure 5.2: Illustration of the tasks that contribute towards the cardinals in (5.4) and (5.5) for an arbitrary task \((i, j) \in D_{WRK}\). These counts define this task’s first indices and determine the position of this task’s windows on the streams of array \(s\).

\[
\sum_{c, c'} P_{S} \times S_{c, c'}^{\geq m} = \bigcup_{l=m}^{n} P_{S} \times S_{c, c'}^{l}
\]

and is effectively a subset of \(\mathbb{Z}^{n-(m-1)} \times \mathbb{Z}^{n-(m-1)}\). When the entire loop nest is within the target region, we write \(P_{S} \times S_{c, c'}^{1}\) as shorthand for \(P_{S} \times S_{c, c'}^{\geq 1}\).

We assume every pair of producer and consumer tasks at each level \(l \in \{m..n\}\) differ by the same distance vector \(d \in \mathbb{Z}^{n-(m-1)}\):

\[
t \to t' \in P_{S} \times S_{c, c'}^{l} \rightarrow t' - t = d.
\]

This assumption further restricts the applicability scope of this procedure by specifying a subset of the OpenStream programs detailed in section 4.5. As before, tasks must express quasi-affine access functions, window first indices and horizons. In addition,
any pair of dependent tasks must now employ the same access functions and window indices aside from an arbitrary, but constant, offset.

To detect offending programs we adopt the same pragmatic approach as for the windows’ first indices in section 4.5. We carry out the dependence analysis disregarding the problem and inspect the resulting dependence set. If a non-compliant dependence pattern is found, we halt the procedure.

Finally, in preparation for future steps, we present the result as a collection of separate dependence polyhedra. We recall that $P_{c \times S|\geq m}^S$ will, in the general case, be a union of polyhedra. This can be due to the lexicographic sequencing predicate, but also due to, for instance, window horizon or indices given as quasts. This poses no difficulty since it only requires splitting the logic formula in disjunctive normal form given by isl/barvinok into its conjunction-only subformulae.

Since the entire loop nest is enclosed between the #fuse and #endfuse delimiters we must obtain the full dependence sets. In addition, the horizons are unitary, so equality on the first indices of producer and consumer windows suffices. We detail the non-empty dependence sets:

$$P_{w \rightarrow pl}^{WRK \times WRK} = \{(i, j) \rightarrow (i', f') \in D_{WRK}^2 | f_w^{WRK}(i, j) = f_{pl}^{WRK}(i', f')$$
$$\land a_w^{WRK}(i, j) = a_{pl}^{WRK}(i', f')\}\$$
$$= \{(i, j) \rightarrow (i', f') \in D_{WRK}^2 | i + 1 = i' + 1 \land j = f' - 1\}\$$
$$= \{(i, j) \rightarrow (i, j + 1) \in D_{WRK}^2\},$$

$$P_{w \rightarrow tr}^{WRK \times WRK} = \{(i, j) \rightarrow (i', f') \in D_{WRK}^2 | f_w^{WRK}(i, j) = f_{tr}^{WRK}(i', f')$$
$$\land a_w^{WRK}(i, j) = a_{tr}^{WRK}(i', f')\}\$$
$$= \{(i, j) \rightarrow (i', f') \in D_{WRK}^2 | i + 1 = i' \land j = f'\}\$$
$$= \{(i, j) \rightarrow (i + 1, j) \in D_{WRK}^2\},$$

$$P_{w \rightarrow pr}^{WRK \times WRK} = \{(i, j) \rightarrow (i', f') \in D_{WRK}^2 | f_w^{WRK}(i, j) = f_{pr}^{WRK}(i', f')$$
$$\land a_w^{WRK}(i, j) = a_{pr}^{WRK}(i', f')\}\$$
$$= \{(i, j) \rightarrow (i', f') \in D_{WRK}^2 | i + 1 = i' \land j = f' + 1\}\$$
$$= \{(i, j) \rightarrow (i + 1, j - 1) \in D_{WRK}^2\}.\$$

The dependence graph corresponding to these is depicted in fig. 5.3 for a subset of the iteration domain.
5. Tile the control program by providing to PLuTo’s algorithm the dependences obtained in step 4. The result is a set of interchangeable affine hyperplanes along which iterations can safely be grouped\(^3\). By construction, this new control program places consumer tasks at a lexico-larger position than their producers. For algebraic manipulation convenience in later steps of this procedure, in particular those requiring polyhedral projections, the tiled iteration domain of statement \(S\) is here denoted by a binary relation, \(D_{S\Omega}\), from the inter- to the intra-tile iterators, which live, respectively, in the domain, \(\text{dom} D_{S\Omega} = D_{T,S\Omega}\), and range, \(\text{ran} D_{S\Omega} = D_{t,S\Omega}\), of said relation. Note that, since the tiling map is injective into the intra-tile iterators alone, the range of this relation contains sufficient information to identify — but not order — each iteration. Hence, whenever tile information and order are unimportant, we will, for simplicity’s sake, identify each iteration by its tuple of intra-tile iterators alone on behalf of the pair of tuples of inter- and intra-tile iterators.

As all dependences are given by affine forms, a set of affine tiling hyperplanes can

\(^{3}\)Although polynomial dependences impair other steps of this procedure and, thus, the whole, they are not a problem in the search for affine hyperplanes: using Schweighofer’s Theorem in lieu of Farkas’ Lemma, albeit harder due to the unbounded order of the polynomial products, suffices. Regarding code generation, as iteration domains are polyhedral, traditional techniques still apply.
be found using Farkas’ Lemma. PLuTo produces the following map:

\[ \Delta_{WRK} \rightarrow \Delta_{WRK}^\odot \]

\[ (i, j) \mapsto (i_T, j_T) \rightarrow (i_t, j_t) = \left( \left\lfloor \frac{i}{B_i} \right\rfloor, \left\lfloor \frac{i + j}{B_j} \right\rfloor \right) \rightarrow (i, i + j). \] (5.8)

Figure 5.4 illustrates \( \Delta_{WRK}^\odot \) and the relationship between old and new iterators.

Figure 5.4: Tiled iteration domain for WRK. Tasks falling on the same tile lie on cells with the same background colour. Tasks created by both SRC and SNK remain untouched as they lie outside the interest region.

Symbolic FME allows us to generalise PLuTo’s numerical tile sizes, \( B_i \) and \( B_j \), to parametric tile sizes\(^4\). \( \Delta_{WRK}^\odot \) is the set of pairs \((i_T, j_T) \rightarrow (i_t, j_t) \in \mathbb{Z}^2 \times \mathbb{Z}^2\) such that

\(^4\)Unfortunately, this impairs automated projections and cardinality computations using isl/barvinok. Therefore, the tile sizes can either be understood as compile-time constants or those computations must, as they were here, be performed symbolically.
the inter-tile iterators satisfy:

\[ 0 \leq i_T \leq \left\lfloor \frac{I - 1}{B_i} \right\rfloor \]
\[ \left\lfloor \frac{i_T B_i + 2}{B_j} \right\rfloor - 1 \leq j_T \leq \min \left( \left\lfloor \frac{i_T B_i + (B_i - 1) + (N - 2)}{B_j} \right\rfloor , \left\lfloor \frac{(I - 1) + (N - 2)}{B_j} \right\rfloor \right) \]

and the intra-tile iterators satisfy:

\[ \max (i_T B_i, j_T B_j - (N - 2)) \leq i_t \leq \min (i_T B_i, j_T B_j + (B_j - 1) - 1, I - 1) \]
\[ \max (j_T B_j, i_t + 1) \leq j_t \leq \min (j_T B_j, i_t + (N - 2)) \cdot \]

(5.9)

6. Rewrite the loop statements according to step 5. Interchange the intra-tile loops with the task creation statement. The latter must be reorchestrated to provide tile-level synchronisation and the tile body rewritten in accordance to the fused communications and computation. This is the motive of the following steps. Henceforth, the term task will refer to the new, fused, tile-level tasks, whilst the individual, unfused tasks of the original program will now give place to iterations within each tile. Each iteration will keep the semantics of the original program’s task it replaces: it will operate on the same input to compute the same output. Parallelism is still exploited at the task level, but is now coarser, since iterations execute sequentially inside each tile.

Listing 5.2: Skeleton of the new fused WRK task creation statement and its surrounding loop nest, obtained by application of the tiling loop transformation.
5.2 New Tasks, new Streams, new Communications

We now focus on building the new tile-level task creation statement. Start by the inter-tile dependences, leaving communication with any sink or source tasks for step 10. The aim is to build new, coalesced, tile-to-tile communication channels for each dependence direction whilst ensuring, as required by the OpenStream runtime, recall constraint (C3), that a write window from some tile task only produces data for another single tile. Constrict the dependence subsets, written in terms of the new iterators, with the requirements that in each such subset:

a) Producer and consumer iterations belong to different tiles;

b) Consumer iterations for the same producer tile all lie within the same tile.

Since the schedule in step 5 guarantees consumer tiles are lexico-larger than their producers, we could use \( t_T < t'_T \) to force distinct tiles, with \( t_T \) and \( t'_T \), respectively, the producer and consumer tile iteration vectors. However, as in each dependence subset a producer tile should map to a single consumer tile, we instead require \( t_T \neq t'_T \) and one of the following:

a) Assume the tile size along the \( i^{th} \)-dimension, \( B_i \), is known statically, at compile-time. In such case, we exhaust all cross-tile dependences between any two tiles for which the respective iterators, \( i_T \) and \( i'_T \), conform to \( 0 \leq i'_T - i_T \leq 1 + \left\lfloor \frac{d[i]-1}{B_i} \right\rfloor \) with \( d[i] = i'_t - i_t \), the dependence distance\(^5\) in the \( i^{th} \)-dimension, which we assume constant, in terms of the intra-tile iterators, \( i'_t \) and \( i_t \);

b) Assume now that for the \( j^{th} \)-dimension, \( B_j \) is an unknown parameter. As it would be impossible to state all the possibilities extensionally, we assume a difference of at most one unit in the inter-tile iterator, i.e. we require \( 0 \leq j'_T - j_T \leq 1 \), implicitly assuming the tile size is greater than or equal to the dependence distance in this dimension. This limits the range of valid tile sizes\(^6\).

In terms of the new iterators, the old iterators are written:

\[
(i, j) = (i_t, j_t - i_t) . \tag{5.10}
\]

\(^5\)As this dependence distance is being evaluated in the tiled iteration domain, it is guaranteed to be non-negative. This is by construction and a requirement of PLuTo’s tiling algorithm [BHRS08], see section 3.2.3.

\(^6\)Note that lower limits for the tile size can be obtained statically, after dependence extraction. These can then be communicated to the user or even verified and enforced in the generated code.
Replacing these in (5.7) yields:

\[ p^{\text{WRK} \Rightarrow \text{WRK}}_{w \leftarrow p_i} = \left\{ (i_t, j_t) \to (i'_t, j'_t) \in D^2_{t, \text{WRK}} \mid i_t = i'_t \land j_t - i_t = j'_t - i'_t - 1 \right\} \]

\[ = \left\{ (i_t, j_t) \to (i_t, j_t + 1) \in D^2_{t, \text{WRK}} \right\}, \]

\[ p^{\text{WRK} \Rightarrow \text{WRK}}_{w \leftarrow t} = \left\{ (i_t, j_t) \to (i'_t, j'_t) \in D^2_{t, \text{WRK}} \mid i_t + 1 = i'_t \land j_t - i_t = j'_t - i'_t \right\} \]

\[ = \left\{ (i_t, j_t) \to (i_t + 1, j_t + 1) \in D^2_{t, \text{WRK}} \right\}, \]

\[ p^{\text{WRK} \Rightarrow \text{WRK}}_{w \leftarrow p_t} = \left\{ (i_t, j_t) \to (i'_t, j'_t) \in D^2_{t, \text{WRK}} \mid i_t + 1 = i'_t \land j_t - i_t = j'_t - i'_t + 1 \right\} \]

\[ = \left\{ (i_t, j_t) \to (i_t + 1, j_t) \in D^2_{t, \text{WRK}} \right\}, \]

and forcing distinct inter-tile iterators finally yields:

\[ p^{\text{WRK} \Rightarrow \text{WRK}}_{w \leftarrow p_i \times (0,1)} = \left\{ [(i_T, j_T) \to (i_t, j_t)] \to [(i'_T, f'_T) \to (i'_t, f'_t)] \in p^{\text{WRK} \Rightarrow \text{WRK}}_{w \leftarrow p_i} \mid (i'_T, f'_T) = (i_T, f_T + 1) \right\} \]

\[ = \left\{ [(i_T, j_T) \to (i_t, j_t)] \to [(i_T, j_T + 1) \to (i_t, j_t + 1)] \in D^2_{\text{WRK}} \right\}, \]

\[ p^{\text{WRK} \Rightarrow \text{WRK}}_{w \leftarrow t \times (0,1)} = \left\{ [(i_T, j_T) \to (i_t, j_t)] \to [(i_T, j_T + 1) \to (i_t + 1, j_t + 1)] \in D^2_{\text{WRK}} \right\}, \]

\[ p^{\text{WRK} \Rightarrow \text{WRK}}_{w \leftarrow t \times (1,0)} = \left\{ [(i_T, j_T) \to (i_t, j_t)] \to [(i_T + 1, j_T) \to (i_t + 1, j_t + 1)] \in D^2_{\text{WRK}} \right\}, \]

\[ p^{\text{WRK} \Rightarrow \text{WRK}}_{w \leftarrow p_t \times (1,0)} = \left\{ [(i_T, j_T) \to (i_t, j_t)] \to [(i_T + 1, j_T) \to (i_t + 1, j_t)] \in D^2_{\text{WRK}} \right\}. \] (5.12)

These five dependence directions are depicted in fig. 5.5.

8. Associate a new, single, coalesced stream with each set obtained at the end of step 7. This includes both write to consuming read dependences and write to peek dependences. Since the data written to these streams is read by a single tile task whichever the case, peek accesses will now be regarded as consuming reads. By construction, these new communication channels:

a) Guarantee no dependence is disrupted. In fact, since we are considering uniform
5.2. NEW TASKS, NEW STREAMS, NEW COMMUNICATIONS

Figure 5.5: Cross-tile dependences in $\mathcal{D}_{WRK}$. The write to read dependence set is split into three polyhedra, guaranteeing that on the same polyhedron and for the same producer tile, all of its consumer iterations lie on the same tile.

dependencies between instances of the same statement and leveraging affine transformations, dependences remain uniform in the target space. Since these dependences are one-to-one between tiles, reading and writing to the new streams inside each tile using the new order will, therefore, automatically ensure producer-consumer relationships are preserved;

b) Reduce runtime overhead. As stream array accesses are being replaced by larger, coalesced accesses on a reduced number of single streams, there are fewer stream queues to manage, simplifying dependence resolution [Dre15];

c) May increase data communication volume. As no distinction is made between peeks and reads, a new stream is created for each of these dependence channels. If the data accessed by a tile through both mechanisms happens to overlap, unnecessary data copies need both be written and read;

d) Ensure a producer tile window writes to a single consumer tile, as required by the OpenStream runtime [PC13].

Item a) means the relations obtained in step 7 can, in fact, be seen as the cross-tile
CHAPTER 5. SCHEDULING FOR GRANULARITY CONTROL

iteration-level dependences in the transformed SCoP. The inter-tile dependences would be the result of projecting these into the inter-tile iterators.

9. Consider the access windows for the new coalesced accesses. Observe that, as we now handle peeks as consuming reads, the burst equals the horizon for all accesses. To find the horizon for a given access, project the dependence relations in step 7 into either their domain or range, i.e. producer or consumer iterators, respectively, depending on which end of the dependence we are currently analysing. Then, compose the resulting relation between inter- and intra-tile iterators with the function mapping each iteration in the tiled domain to its corresponding original task’s horizon on this access. Each tile is now related to the horizons of the individual, communicating iterations within it. The tile’s lump horizon for said access is the sum thereof. As the dependence relations of step 7 only include the communicating iterations for each tile instance, the horizon will likely be a function of the inter-tile iterators. Corner cases, e.g. incomplete boundary tiles, are thus automatically covered. This allows us to version each tile’s dependences at runtime and to transparently handle zero-horizon accesses, i.e. those tile instances which do not participate in the dependence relation in question.

Let us determine the read horizon (and burst) for $p^{WRK\times WRK}_{w-p_l \times (0,1)}$ which corresponded to a peek access in the original program. Its range, i.e. consumer projection, is

$$\text{ran } p^{WRK\times WRK}_{w-p_l \times (0,1)} = \{ (i_T, j_T) \to (i_l, j_l B_l) \in D^{WRK\times WRK}_{WRK} \mid 2 \leq j_l B_l - i_l \}. \quad (5.13)$$

Since all horizons are unitary, the lump horizon for each tile is simply the cardinal over the range of this relation. This count is expressed as a quast on the inter-tile iterators:

$$h^{WRK\times WRK}_{w-p_l \times (0,1)} (i_T, j_T) = h^{WRK\times WRK}_{w-p_l \times (0,1)} (i_T, j_T)$$

$$= \text{Card} \left( \text{ran } p^{WRK\times WRK}_{w-p_l \times (0,1)} \right)$$

$$= \text{Card} \left\{ (i_T', j_T') \to (i_l', j_l') \in \text{ran } p^{WRK\times WRK}_{w-p_l \times (0,1)} \mid (i_T', j_T') = (i_T, j_T) \right\}$$

$$= \begin{cases} \text{if } (B_i - 1) + 2 \leq j_T B_l - i_t B_l \leq N - 2 \\
\quad \land \ i_T B_l + (B_i - 1) \leq I - 1 \text{ then } B_i \end{cases}$$

$$= \begin{cases} \text{if } (I - 1) + 2 \leq j_T B_l \leq N - 2 \\
\quad \land \ i_T = 0 \land I \leq B_i - 1 \text{ then } I \end{cases}$$

$$= \begin{cases} \text{if } \ldots \text{ then } \ldots \ [\text{additional corner cases}] \text{ else } 0. \end{cases} \quad (5.14)$$
5.2. **NEW TASKS, NEW STREAMS, NEW COMMUNICATIONS**

Figure 5.6 depicts those tasks involved in this cardinal. The obtained form can then be directly used to build new code:

```
#fuse
stream wrk_wpl;

for (int iT = 0; iT <= floor(I - 1, Bi); iT++)
  for (int jT = ceil(iT * Bi + 2, Bj) - 1;
       jT <= min(floor(iT * Bi + (Bi - 1) + (N - 2), Bj),
                 floor((I - 1) + (N - 2), Bj)); jT++)
  {
    int hor_wrk_wpl_pl = (Bi - 1) + 2 <= jT * Bj - iT * Bi
      && jT * Bj - iT * Bi <= N - 2
      && iT * Bi + (Bi - 1) <= I - 1 ? Bi :
        (I - 1) + 2 <= jT * Bj && jT * Bj <= N - 2
      && iT = 0 && I <= Bi - 1 ? I :
        . . . ? . . . :
    0;

    task peek ( . . . ) \
       read (wrk_wpl, win_wrk_wpl_pl(hor_wrk_wpl_pl) . . . ) \
        write ( . . . ) do // WRK (fused)
          { . . . }
  }
#endfuse
```

Listing 5.3: Skeleton of the target region after building a new access window win_wrk_wpl_pl for the new coalesced read. We compute its horizon (and burst) hor_wrk_wpl_pl and use it to access a new stream wrk_wpl. This replaces the left-sided inter-tile peek access in the running Gauss-Seidel 1d example.

Notice how some quantities, e.g. $i_T B_i$, are frequently used, e.g. to compute the window's horizon or the loop bounds. This presents an obvious opportunity for static optimisation, e.g. through a syntactic pass that assigns repeatedly used quantities to variables. This is left for future work.

10. Consider communication with tasks external to the target, tiled region, i.e. tasks created outside `#fuse` and `#endfuse`. These communication channels require a different approach from those between tiles in steps 8 and 9: as we do not wish to modify task creation statements lying outside the tiled region, these accesses must use the streams of the original program. Furthermore, we must guarantee the individual tile iterations that consume from source and write to sink tasks access the same data as their
Communicating iterations:

Figure 5.6: Communicating iterations in $D_{WRK}$ for the dependence channel $w \rightarrow p_1 \times (0, 1)$. The iterations responsible for the left-peek, now consuming read, are those in (5.13) and whose intra-tile cardinal we obtained in (5.14). We also show the iterations responsible for the writes, to demonstrate how, within the same tile, the horizon on both ends of the same dependence channel might differ.

respective tasks did in the original program. To this end, any lexico-larger iteration must correspond to an original task whose first read (resp. write) index on a stream is not less than the last read (resp. write) index on the same stream in any lexico-smaller iteration. In other words, streams carrying external data must be read and written in the original order. This guarantees the external data connected to each tile is the collection of the external data each of its individual tasks would access in the original program.

i) Similarly to step 4, for each stream array involved, determine the data dependences between tasks created within the target region and external tasks;

ii) Project the obtained dependence relations onto the iterators of the target statement, i.e. the relations’ range or domain for tasks that consume from source or write to sink tasks, respectively;

iii) Write the resulting subset of the target statement iteration domain in terms of the new tiled domain iterators. The result is a a subset of the tiled iteration domain
5.2. NEW TASKS, NEW STREAMS, NEW COMMUNICATIONS

... describing the iterations involved in each of the dependence relations considered;

iv) For each dependence, consider the first and last indices of the original task corresponding to each iteration. Recall these indices are, in general, a function of both the accessed stream and the task (now iteration) iteration vector;

v) Show, for each stream, that the difference between first and last indices is non-negative over the lexicographic order on the iterations obtained in item iii).

We illustrate the procedure for the dependence channels between WRK and SNK. The first read index for every task created by SNK can be written (see step 3 or (4.3)):

\[
\begin{align*}
 f_{r}^{SNK}(j) &= \text{Card}\left\{ (i', j') \in \mathcal{D}_{WRK} \mid a_{r}^{WRK}(i', j') = a_{r}^{SNK}(j) \right\} \\
 &+ \text{Card}\left\{ j' \in \mathcal{D}_{SNK} \mid j' < j \land a_{r}^{SNK}(j') = a_{r}^{SNK}(j) \right\} \\
 &= \text{Card}\left\{ (i', j') \in \mathcal{D}_{WRK} \mid j' = j \right\} + \text{Card}\left\{ \right\} \\
 &= I. \\
\end{align*}
\]

The dataflow dependences from WRK to SNK can now be obtained by comparing (5.15) with (5.5). These are summarised by the following polyhedron:

\[
\begin{align*}
 p_{WRK \times SNK}^{*} &= \{(i, j) \rightarrow j' \in \mathcal{D}_{WRK} \times \mathcal{D}_{SNK} \mid f_{w}^{WRK}(i, j) = f_{r}^{SNK}(j') \\
 &\land a_{w}^{WRK}(i, j) = a_{r}^{SNK}(j') \} \\
 &= \{(i, j) \rightarrow j' \in \mathcal{D}_{WRK} \times \mathcal{D}_{SNK} \mid 1 + i = I \land j = j' \} \\
 &= \{(I - 1, j') \rightarrow j' \in \mathcal{D}_{WRK} \times \mathcal{D}_{SNK} \}. \\
\end{align*}
\]

The domain of this relation consists of the original tasks in WRK involved in this dependence:

\[
\text{dom } p_{WRK \times SNK}^{*} = \{(I - 1, j) \in \mathcal{D}_{WRK} \}. \\
\]

Writing this subset of \( \mathcal{D}_{WRK} \) in terms of the new iterators reveals the individual iterations involved in this dependence:

\[
\text{dom } p_{WRK \times SNK}^{\otimes} = \{(i_{T}, j_{T}) \rightarrow (I - 1, j_{T}) \in \mathcal{D}_{WRK}^{\otimes} \}. \\
\]

This concludes item iii) above. Next, recall that for the running example the first and last write indices match, since the horizon is unitary. By means of (5.5), they are written:

\[
 f_{w}^{WRK}(i_{T}, j_{T}) = f_{w}^{WRK}(i_{T}, j_{T}) = 1 + i_{T}. \\
\]
Despite being written in terms of the new iterators, these do not correspond to any window’s first or last index in new the control program. These still refer to the original, fine-grained tasks, even though, intuitively, they might clarify the role of the individual, after-tiling iterations in this dependence. Over the iterations of interest in (5.18),

\[ f_{w}^{\text{WRK}}(i_{T}, j_{T}) \rightarrow (I - 1, j_{i}) = i_{w}^{\text{WRK}}(i_{T}, j_{T}) \rightarrow (I - 1, j_{i}) = I \]  

(5.20)

irrespective of the particular stream. It follows immediately that the difference is non-negative for any two iterations, \([i_{T}, j_{T}) \rightarrow (I - 1, j_{i}) \prec ([i'_{T}, j'_{T}) \rightarrow (I - 1, j'_{i})].\]

In general, this check would require considering \(a_{i_{w}}^{\text{WRK}} = s[j_{i} - i_{i}],\) the stream access function for the (sole) write access in WRK expressed in terms of the new intra-tile iterators, see (5.2). Then, we restrict the space to only those tasks producing on the \(k^{th}\)-stream by fixing \(s[j_{i} - i_{i}] = s[k].\) Finally, Farkas’ Lemma can be deployed to provide the desired non-negativity check.

11. If the check in step 10 fails, we go back to step 5, rejecting the schedules tested thus far. New tiling transformations may be quickly obtained by modifying the current schedule, e.g. by interchanging the tiling hyperplanes\(^7\). Alternatively, however, we can look for a completely new schedule by restraining the schedule search space, i.e. by constraining the schedule coefficients to stricter subsets of \(\mathbb{Z}.\)

As all conditions were met at the end of step 10, we may conclude this is a suitable collection of hyperplanes and that no further action is necessary. Optionally, we could swap them for a better traversal order in the control program such that sooner-to-be-ready tile tasks were created first. Such considerations are left for future work.

12. Consider the access windows for communication from source and to sink tasks. For each stream array and each declared access, the aim is to describe, in intension, the set of streams involved within each tile. Project the external dependences into the iterators of the target statement, i.e. consider each of the objects in item iii) of step 10: relations from the inter- to the intra-tile iterators of the external communicating iterations. Compose these with the stream access functions written in terms of the intra-tile iterators. The result is the relation we seek between tile instances and involved streams.

Let us turn our attention to the data dependences between tasks created by SRC and

\(^7\)In a traditional shared-memory setting, any tiling hyperplane reorder is valid and only changes traversal order [Pou10]. Here, the interchange might change the stream access patterns, which can be advantageously used to transform an invalid schedule (in the sense of step 10) into a valid one.
5.2. NEW TASKS, NEW STREAMS, NEW COMMUNICATIONS

WRK. Similarly to the process conducted in step 10 that culminated in (5.18), we can obtain the sets of $D_{WRK^\otimes}$ iterations that take part in these communication channels. Notice how the quast in (5.4) leads to two polyhedra for the right-sided peek:

\[
\begin{align*}
\text{ran } P_{w-p_l} &= \{(i_T, j_T) \rightarrow (i_t, i_t + 1) \in D_{WRK^\otimes}\}, \\
\text{ran } P_{w-r} &= \{(0, j_T) \rightarrow (0, j_t) \in D_{WRK^\otimes}\}, \\
\text{ran } P_{w-p_l (i)} &= \{(0, j_T) \rightarrow (0, j_t) \in D_{WRK^\otimes} | j_t \leq N - 3\}, \\
\text{ran } P_{w-p_l (ii)} &= \{(i_T, j_T) \rightarrow (i_t, i_t + (N - 2)) \in D_{WRK^\otimes}\}.
\end{align*}
\]  

(5.21)

The iterations in each of these sets are depicted in fig. 5.7. From (5.2) in step 3, it is easy to write the stream access functions for each of the input accesses in WRK in terms of the new intra-tile iterators:

\[
\begin{align*}
i &= i_t \\
\end{align*}
\]

\[
\begin{align*}
0 & 1 2 N - 1 N - 2 \ldots \\
\end{align*}
\]

Task creation statement: SRC WRK Tiles in $D_{WRK^\otimes}$:

External communication: $w \rightarrow p_l$ $w \rightarrow r$ $w \rightarrow p_l (i)$ $w \rightarrow p_l (ii)$

Figure 5.7: External communicating iterations in WRK for dependence channels originating in SRC tasks. These iterations correspond to the description in (5.21).
Finally, composing each of the relations in (5.21) with the appropriate access function in (5.22), we can relate each tile task to the external streams each of them references:

\[
\begin{align*}
A^{\text{SRC} \times \text{WRK}}_{T, w \rightarrow p_l} &= a^{\text{WRK}}_{t, p_l} \circ \text{ran } P^{\text{SRC} \times \text{WRK}}_{w \rightarrow p_l} \\
&= \{ (l_T, j_T) \rightarrow s[0] \in D_{r, \text{WRK}} \times S \mid (l_T, j_T) \rightarrow (i_l, t_l) \in D_{r, \text{WRK}} \}, \\
A^{\text{SRC} \times \text{WRK}}_{T, w \rightarrow r} &= \{ (0, j_T) \rightarrow s[j] \in D_{r, \text{WRK}} \times S \mid (0, j_T) \rightarrow (0, j_l) \in D_{r, \text{WRK}} \}, \\
A^{\text{SRC} \times \text{WRK}}_{T, w \rightarrow p_r (i)} &= \{ (0, j_T) \rightarrow s[j] \in D_{r, \text{WRK}} \times S \mid (0, j_T) \rightarrow (0, j_l - 1) \in D_{r, \text{WRK}} \land j_l \leq N - 2 \}, \\
A^{\text{SRC} \times \text{WRK}}_{T, w \rightarrow p_r (ii)} &= \{ (l_T, j_T) \rightarrow s[N - 1] \in D_{r, \text{WRK}} \times S \mid (l_T, j_T) \rightarrow (i_l, t_l + (N - 2)) \in D_{r, \text{WRK}} \}. 
\end{align*}
\] (5.23)

13. Since, within the same tile, different accesses on the same stream array might involve some of the same streams, we need to determine the overlapping regions in order to combine bursts and horizons.

i) Compute, within the same stream array and access class, read or write, the intersections and complements thereof of the ranges of the stream relations in step 12. For \( n \) relations, this produces, at most, \( 2^n - 1 \) new non-empty relations. Writes and reads are then organised into new relations from tile instances to segments of the original stream arrays. These must be copied into new stream arrays associated with their own cumulative burst and horizon (computed in step 14);

ii) Generate code to initialise the new arrays. As stream arrays are limited to one dimension, so is the resulting loop iteration domain. This is characterised by a pair of lower, lba, and upper, uba, bounds and a stride, which may be computed by any usual technique, see section 3.3. Some arrays might be empty for some tile instances. In such cases, these are replaced with a dummy one-stream array.\(^9\)

It is easy to see that only \( A^{\text{SRC} \times \text{WRK}}_{T, w \rightarrow r} \) and \( A^{\text{SRC} \times \text{WRK}}_{T, w \rightarrow p_r (i)} \) in (5.23) might include

\(^8\)As many as the (potentially) non-empty regions of an \( n \)-set Venn diagram.

\(^9\)Since the new program is no longer compliant with (P3) and (P4), this means the current procedure is not closed on the polyhedral subset of OpenStream.

\(^{10}\)This is due to the inherently static nature of the task creation statement’s task directive which must include the access regardless of whether all tile instances require it or not, provided at least one does.
some of the same streams. Their intersection and complements are written:

\[
A^{SRC \times WRK}_{T, w=r \setminus pr (i)} = A^{SRC \times WRK}_{T, w=r} \setminus A^{SRC \times WRK}_{T, w=pr (i)} \\
= \{(0, j) \rightarrow s[lba = uba][\max(1, jT B)] \in D_{T, WRK} \times S \mid jT B \leq N - 2\},
\]

\[
A^{SRC \times WRK}_{T, w=r \cap pr (i)} = \{(0, j) \rightarrow s[j] \in D_{T, WRK} \times S \mid 1 + \max(1, jT B) \leq j \leq \min(jT B + (B - 1), N - 2)\},
\]

\[
A^{SRC \times WRK}_{T, w=pr (i) \setminus r} = \{(0, j) \rightarrow s[j] \in D_{T, WRK} \times S \mid jT B + B \leq N - 2\}.
\]

The streams in each of these relations are depicted in fig. 5.8.

Figure 5.8: Within the same tile, some iterations are responsible for external communications using the same streams. Here, we selected a tile reading from SRC and overlaid the streams for either consuming reads, right-sided peeks or both, see (5.24).
Code generation is particularly effortless. We exemplify the procedure on the overlapping peek and read accesses’ array. The lower and upper bounds, lba and uba, of the loop and the new stream array can be directly extracted from the description above. Since this relation has no ‘holes’, the stride is unitary, allowing us to syntactically replace the for loop with a memcpy block memory copy to set up the new stream array.

```c
stream s[N];

#pragma fuse
for (int iT ..)
   for (int jT ..)
      {
         ...
         int lba_src_r_pr = 1 + max(1, jT * Bj);
         int uba_src_r_pr = min(jT * Bj + (Bj - 1), N - 2);
         int len_src_r_pr = uba_src_r_pr - lba_src_r_pr + 1;

         bool bl_src_r_pr = iT == 0 && len_src_r_pr > 0;
         len_src_r_pr = bl_src_r_pr ? len_src_r_pr : 1;
         lba_src_r_pr = bl_src_r_pr ? lba_src_r_pr : 0;

         stream s_arr_src_r_pr[len_src_r_pr];
         memcpy(s_arr_src_r_pr, s + lba_src_r_pr, len_src_r_pr);
         ...
      }

#pragma endfuse
```

Listing 5.4: Declaring a new stream array `s_arr_src_r_pr` for the overlapping peek and read accesses. We compute its length `len_src_r_pr` using the lower `lba_src_r_pr` and upper `uba_src_r_pr` bounds extracted directly from (5.24). If, as determined by `bl_src_r_pr`, a tile does not participate in this communication channel with SRC, a dummy one-stream array is created and populated with a reference to `s[0]`.

14. Each of the declared arrays requires distinct variadic windows with possibly different bursts and horizons. The bursts are, for the regions with coinciding accesses, the sum of the bursts for each overlapping access and, for the complements of such regions, the original bursts. The horizons are equal to the bursts except for the ‘pure’ peaks, i.e. those from streams for which there is no consuming read in the current tile instance, for which they are zero. Dummy one-stream arrays are associated with a zero-burst, zero-horizon window, thus carrying no dependence. Later, inside the tile’s
5.2. NEW TASKS, NEW STREAMS, NEW COMMUNICATIONS

task body, we must conditionally direct each individual iteration to the appropriate window, depending on which set the accessed stream falls on.

For the running example, all of the original non-zero bursts and horizons are unitary, which streamlines the procedure. The bursts and horizons are still unitary for the read accesses: both for ‘pure’ reads, \( A_{\text{SRC} \times \text{WRK}}^{S_r \times \text{WRK}} \), and for those reads overlapping peek accesses, \( A_{\text{SRC} \times \text{WRK}}^{S_r \times \text{WRK}} \). The ‘pure’ peeks, e.g. \( A_{\text{SRC} \times \text{WRK}}^{S_r \times \text{WRK}} \), share the unit horizon but bear zero bursts. The window declaration for the overlapping read and peek accesses is given below:

```c
#fuse
for (int iT . . .)
  for (int jT . . .)
  {
    int len_src_r_pr = uba_src_r_pr - lba_src_r_pr + 1;
    // ... 
    bool bl_src_r_pr = iT == 0 && len_src_r_pr > 0;
    // ... 
    stream s_arr_src_r_pr[len_src_r_pr];
    // ... 
    int hor_src_r_pr = bl_src_r_pr ? 1 : 0;
    // task peek ( . . . ) \ 
    read (s_arr_src_r_pr, win_src_r_pr(hor_src_wr_r) . . .) \ 
      write ( . . . ) do // WRK (fused)
    { . . . }
  }
#endfuse
```

Listing 5.5: Building a new window \( \text{win}_{\text{SRC} \times \text{WRK}} \) for the overlapping peek and read accesses. We compute its horizon (and burst) \( \text{hor}_{\text{SRC} \times \text{WRK}} \). If, as determined by \( \text{bl}_{\text{SRC} \times \text{WRK}} \), a tile does not participate in this communication channel with SRC, a dummy zero-burst, zero horizon window is employed.

15. The non-‘pure’ peeks pose a problem, however. As dictated by constraint (C1), the current OpenStream implementation is limited to either exclusive peeks or consuming reads, but precludes combined accesses. As such, if a tile overlaps peeks with consuming reads, we must confirm that, within this tile, the maximum last peek index on a given stream is not greater than the maximum last read index on the same stream. This information can be retrieved by solving the given optimisation problem over the sets...
of iterations within each tile involved in external peek and read accesses, see item iii) of step 10. If this fails to be the case, retract the full procedure. On the contrary, the lump read subsumes the lump peek and the latter can be dropped: iterations whose corresponding tasks were responsible for the individual peeks need no longer carry it out. On successful completion, this step finalises the new control program and should produce a valid, albeit for the time being jobless, OpenStream program.

The iterations involved in external peeks and consuming reads are those in (5.21). From here, it would not be hard to show that the maximum last read index is indeed greater than the maximum last peek index for those channels mediating data communication between SRC and WRK. Note, however, that we can sidestep such effort. The first write index for every task created by SRC is a constant:

\[
\begin{align*}
    j_w^{SRC}(j) &= \text{Card}\left\{ j' \in D_{SRC} \mid j' < j \land a_w^{SRC}(j') = a_w^{SRC}(j) \right\} \\
    &= \text{Card}\{\} \\
    &= 0.
\end{align*}
\]

Therefore, every access in WRK that depends on data produced by SRC must share this index, including the overlapping reads and peeks. This concludes the argument.

5.3 Assembling the Work of Fused Tasks

Next, turn the focus to assembling the work function of the new fused (or tiled) tasks. The aim is to create a collection of data structures, shared by all iterations inside the same tile, which emulate each of the accessed stream arrays in the original program. Tile iterations will solely access these new shared arrays, so that communication falling within the same tile happens exclusively through them. Each iteration will index said arrays using their original window indices. For the time being, per stream array, create a two-dimensional shared array covering the minimum bounding box enclosing the original tasks’ stream accesses inside each tile. In other words, create an array of dimensions given by:

\[
\begin{align*}
    &\text{[greatest accessed stream (= ubs) - least accessed stream (= lbs) + 1]} \\
    \times &\text{[greatest last window index (= ubi) - least first window index (= lbi) + 1]}
\end{align*}
\]
where the range-bounding quantities are to be taken with respect to the original tasks’ stream accesses and will, in general, be a function of the inter-tile iterators. In a nutshell:

i) Use the stream access functions written in terms of the new iterators also leveraged in step 12 to relate each tile task in step 5 to the entire universe of streams accessed by its original tasks. Finally, take the union of the range of those relations, across all input and output accesses, and find its lower, lbs, and upper, ubs, bounds as a function of the inter-tile iterators;

ii) The strategy to find the extreme indices, lbi and ubi, is similar. Apply the functions mapping each iteration to its original task’s first and last indices to the tiled iteration domain in step 5. These are the same functions previously employed in item iv) of step 10. Take the union of the indices in the range of the resulting relations, across all input and output accesses, and find its lower, lbi, and upper, ubi, bounds as a function of the inter-tile iterators.

The new, contiguous data layout will hopefully reveal opportunities for static optimisation, e.g., vectorisation or better register allocation, that were likely not obvious to the compiler with the original fine-grained tasks and their scattered input and output locations.

From the access functions in (5.22), it is clear that lbs is a lower bound on \( a_{t,p_l}^{WRK} \) and ubs an upper bound on \( a_{t,r}^{WRK} \). Note that \( a_{t,w}^{WRK} = a_{t,t}^{WRK} \) and, as such, is of no interest. This remark avoids considering the union in item i) and circumscribes the problem to the range of the following relations:

\[
A_{T,p_l}^{WRK} = a_{t,p_l}^{WRK} \circ D_{WRK} = \{ (i_T, j_T) \rightarrow s[j_t - i_t - 1] \in D_{T,WRK} \times S \\
\quad | \max(1, j_T (B_j - (i_T B_i + (B_i - 1))) - 1) \leq j_t - i_t - 1 \leq \min(\ldots) \} \text{ with lbs},
\]

\[
A_{T,p_r}^{WRK} = \{ (i_T, j_T) \rightarrow s[j_t + 1] \in D_{T,WRK} \times S \\
\quad | \max(\ldots) \leq j_t - i_t + 1 \leq \min(j_T B_j + (B_j - 1) - i_T B_i, N - 2) + 1 \} \text{ with ubs}. \tag{5.26}
\]

Figure 5.9 presents these limits for two tiles. From the particularly simple form of the window indices in (5.4) and (5.5), and since \( i_t = i \), it is apparent that lbi is a lower
bound on the first indices of either the left- or right-sided peeks:

\[ f_{t, \min(p_i, p_j)}^{WRK}(i_t, j_t) = \begin{cases} 2 & \text{if } 2 \leq j_t - i_t \leq N - 3 \\ i_t & \text{else} \end{cases} \tag{5.27} \]

and that ubi is an upper bound on the writes’ last indices, see also (5.19):

\[ i_{t,w}^{WRK}(i_t, j_t) = 1 + i_t. \tag{5.28} \]

Composing these over \( D_{WRK} \), we map each tile to its original tasks’ first peek and last write indices, from which the desired bounds can be readily extracted:

\[
F_{T, \min(p_i, p_j)}^{WRK} = f_{t, \min(p_i, p_j)}^{WRK} \circ D_{WRK} = \begin{cases} (i_T, j_T) & \in D_{T, WRK} \times N \\
\{ (i_T, j_T) \rightarrow 0 & \in D_{T, WRK} \times N \mid \begin{array}{c} f_{T} B_j - \min ((i_T B_i + (B_i - 1)) I - 1) \leq 1 \\
\lor f_{T} B_j + (B_j - 1) - i_T B_i \geq N - 2;
\end{array} \\
\begin{array}{c} (i_T, j_T) \rightarrow i & \in D_{T, WRK} \times N \mid \max (i_T B_i, j_T B_j - (N - 2) + 1) \leq i_t \leq \min (...) \end{array} \} & \end{cases}
\]

\[ i_t \in D_{T, WRK} \times N \]

\[ j_t = i_t \]

\[ i = i_t \]

\[ I - 1 \]

\[ I \]

\[ j = i + j \]

\[ 0 \quad 1 \quad 2 \quad ... \quad N - 2 \quad N - 1 \]

Figure 5.9: The minimum bounding boxes for the stream accesses of the blue tile at the top, \((i_T, j_T) = (0, 1)\), and the singleton green tile at the bottom right, \((i_T, j_T) = (2, 4)\).
5.3. ASSEMBLING THE WORK OF FUSED TASKS

\[ L_{T,w}^{WRK} = \{(i_T, j_T) \to 1 + i_t \in D_{T,WRK} \times \mathbb{N} | \max(...) \leq 1 + i_t \leq 1 + \min(i_T B_i + (B_i - 1), j_T B_j + (B_j - 1) - 1, I - 1) \}. \]  

(5.29)

The boundary-touching tiles require a conditionally-defined lbi. Meanwhile, the straightforward form of (5.28) meant we could directly use the bounds in (5.9) to describe ubi.

Although it is clear from fig. 5.9 that the reserved block of virtual memory largely surpasses the actual memory needs, the lazy memory allocation policy of most operating systems might mitigate most of its impact. The following is code to create a shared array named data:

```c
#define fuse
for (int iT . . .)
    for (int jT . . .)
    {
        . . .
        task peek(. . .) read(. . .) write(. . .) do // WRK(fused)
        {
            int lbi = jT * Bj - min(iT * Bi + (Bi - 1), I - 1) <= 1 
            || jT * Bj + (Bj - 1) - iT * Bi >= N - 2 ? 0 : 
            max(iT * Bi, jT * Bj - (N - 2) + 1);

            int ubi = 1 + min(iT * Bi + (Bi - 1), 
                              min(jT * Bj + (Bj - 1) - 1, I - 1));

            int lbs = max(1, max(jT * Bj - (iT * Bi + (Bi - 1)), 
                                 jT * Bj - (I - 1))) - 1;

            int ubs = min(jT * Bj + (Bj - 1) - iT * Bi, N - 2) + 1;

            double (*data)[ubi - lbi + 1] = 
                malloc(sizeof(double[ubs - lbs + 1][ubi - lbi + 1]));
        . . .
    }
#endif
```

Listing 5.6: Declaration and allocation of data, a shared array for intra-tile communication. The tile-dependent bounds were extracted directly from the text.

17. To perform the fused task’s computational work, populate the intra-tile loop body
with the original tasks’ work function replacing every old window access with an access to the corresponding shared array. To index the latter:

i) For the first dimension, offset the original program’s stream reference, written in terms of the new iterators, by lbs;

ii) For the second dimension, offset the original window’s first index, written in terms of the new iterators, by lbi. Since windows might carry multiple data elements, and this particular access might address an arbitrary such element and not necessarily the first, add any explicit offset for this particular access.

As it is not necessarily the case that every iteration’s original window indices are written as the same closed form of its iterators, e.g. the quasts in step 3, selection statements choosing between one of several flows of control will, in general, be needed.

We recall the stream access functions written in terms of the intra-tile iterators from (5.22):

\[
\begin{align*}
\text{a}_{t,p_l}^{\text{WRK}}(i_t, j_t) &= s[j_t - i_t - 1], \\
\text{a}_{t,r}^{\text{WRK}}(i_t, j_t) &= \text{a}_{t,w}^{\text{WRK}}(i_t, j_t) = s[j_t - i_t], \\
\text{a}_{t,p_r}^{\text{WRK}}(i_t, j_t) &= s[j_t - i_t + 1].
\end{align*}
\] (5.30)

Once offset by lbs, these may be directly used to access data. Similarly, we recall the window first indices:

\[
\begin{align*}
\text{f}_{t,p_l}^{\text{WRK}}(i_t, j_t) &= \text{if } j_t - i_t \geq 2 \text{ then } i_t + 1 \text{ else } 0, \\
\text{f}_{t,r}^{\text{WRK}}(i_t, j_t) &= i_t, \\
\text{f}_{t,p_r}^{\text{WRK}}(i_t, j_t) &= \text{if } j_t - i_t \leq N - 3 \text{ then } i_t \text{ else } 0, \\
\text{f}_{t,w}^{\text{WRK}}(i_t, j_t) &= 1 + i_t.
\end{align*}
\] (5.31)

Each of these should be offset by lbi and used to address the second dimension of data. Note that the consuming read access function and first index remain unused, as per the original code in listing 5.1.

Generating code is, again, quite easy. The challenge is to generate efficient code. The following program uses selection statements to deal with the piecewise definitions in (5.31), but this is far from ideal. Section 5.4 further discusses these issues.
5.3. ASSEMBLING THE WORK OF FUSED TASKS

Listing 5.7: The work of the new fused tasks is a repetition of the initial kernel in listing 5.1, adjusted to access the data array using properly offset stream access functions and window first indices.

18. Since, as seen in steps 16 and 17, we shield the computation kernel of fused tasks from any stream communication, we must pre- and postfix it with code reading the tile’s input and writing the tile’s output, respectively. Thus, in addition to the computation kernel, the tile’s work function will include two more counted loop nests with the same bounds as the kernel’s loop nest, but with different control flow options. These options select those iterations which participate in a given communication channel, i.e. those which belong to the cross-tile dependence polyhedra in step 7 or the external dependence polyhedra in step 10. To achieve this:

i) For the input (resp. output) loop nest, compute the range (resp. domain) of the dependence polyhedra;
ii) Populate selection statements with the polyhedron-bounding conditions just obtained. These statements will then act as filters for which dependence polyhedra the current iteration falls on.

From here, we must distinguish between a) inter-tile, coalesced accesses and b) external accesses:

a) For an inter-tile access, because there is a single, unified stream per dependence, the filter above implicitly determines the accessed stream as well. For the input loop nest, do a horizon’s length memory copy from this stream’s window to the shared array created in step 16 (or vice-versa for the output loop nest), applying the same offset indexing as step 17. We keep a counter to address the stream’s window, which is advanced by the horizon at the end of each copy. This mimics the behaviour of advancing the stream’s access position by the burst since, per step 8, the burst equals the horizon for all accesses;

b) For external accesses, we first need to select the right window in step 13 for the stream of interest. This amounts to extra selection statements conditional on the stream-grouping relation bounds obtained there. As above, we do a horizon’s length memory copy from this stream’s window to the shared array created in step 16 (or vice-versa for output accesses) and keep counters to mimic the accesses’ burst behaviour. For ‘pure peeks’ we need no counter and for all the other situations, the burst is still the horizon.

This finalises the tiles’ task body and hence the new source code. The result is a semantically equivalent OpenStream program which, relative to the original, features superior performance at the expense of coarser tasks, communications and parallelism.

Let us write the code to populate data with the information received via the inter-tile communication channel \( w \to p_l \times (0,1) \) from step 9. The iterations of interest are the red-coloured iterations in fig. 5.6 given in (5.13):

\[
\text{ran } \mathcal{D}_{w \to p_l \times (0,1)}^{\text{WRK}} = \{(i_T, j_T) \to (i_t, j_T B_j) \in \mathcal{D}_{\text{WRK}} \mid j_T B_j - i_t \geq 2\}. \tag{5.32}
\]

From here, since the horizon is unitary, we simply require an assignment to the element of data corresponding to a left-sided peek, see (5.30) and (5.31) in the previous step. A counter is kept to track the next reading position for the following iteration in (5.32).
5.4 Lessons from Preliminary Experimental Results

Experimental evaluation of the Gauss-Seidel 1d kernel studied in this chapter shows some promising results. The performance data refers to the latest version of OpenStream [PP20] on a machine equipped with an Intel Core i7-6700 processor with four physical cores and eight logical threads, and 48 GiB of main memory.

For $N = I = 10000$ and double-precision floating-point data, the final program is
three orders of magnitude faster than the original specification. Since this applies for a large range of different fused task granularities, it indicates the performance gains are mainly due to the reduced number of tasks and simpler runtime dependence resolution.

Moreover, it is approximately 50\% slower than the manually optimised version of the same kernel provided with OpenStream [PP20] for empirically optimal task sizes. These tasks comprise 150 × 150 iterations for the version derived here and 1250 iterations for the externally provided code, which can only exploit loop strip-mining for submultiples of the whole loop trip. The discrepancy in execution time is likely due to the particularly inefficient control flow of the current implementation. In fact, the strategy in section 5.3 versions the different iteration domains used to compute and (un)load streams by populating for loop statements running over the entire space with numerous if statements. Instead, a state-of-the-art code generator would split and adjust the bounds of the loop statements themselves, thereby avoiding expensive choices between several flows of control. The engineering work required to bring these solutions to OpenStream, even if for non-parametric tile sizes, is left for future work.

5.5 Summary

We presented a task fusion procedure which allows the programmer to concentrate on issues that are inherent to the algorithm that is being implemented and express its OpenStream application using natural, very fine-grained tasks. Ideally for the programmer, the initial OpenStream implementation is completely unaware of the underlying machine architecture, runtime system or operating system settings. In addition, the program only expresses dataflow dependences, is free from data races and enjoys functional determinism, promoting productivity throughout the software development process.

Our approach works by analysing such a program specification and grouping tasks and communication patterns to improve the program’s performance based on the same principles as traditional and well-established loop tiling. We presently lack a full implementation of this algorithm employing state-of-the-art code generation and a comprehensive set of benchmarks to test it on. However, based on previous evidence for manual polyhedral fusion of OpenStream programs [NDRP19] and the preliminary results in section 5.4, we are confident automatically fused programs can also enjoy the typical performance advantages of loop tiling through improved data locality and reduced communication volume.
Chapter 6

Scheduling for Stream Bounding

Streams in CDDF Openstream are assumed unbounded and resource deadlocks impossible. However, every implementation will, in practice, need to consider the memory limitations of the hardware it runs on. We present a strategy to bound streams in OpenStream programs by introducing new WaR, ‘back-pressure’ dependences in the polyhedral representation of the program based on the work in [CDF16]. A few example OpenStream SCoPs are presented which illustrate the procedure but also its limitations. Despite the identified shortcomings, a conservative procedure is delineated which, if the search for a schedule is successful, can certify the non-deadlocking execution of an OpenStream program on realistic, memory bounded hardware.

6.1 Back-Pressure Dependences

If for a task execution schedule \( \theta \) and a stream \( s \) there exists a positive integer \( L_s \), for which all stream indices \( i \leq j - L_s \) are dead (i.e. already both produced and consumed), whenever position \( j \) is written to by some task, then \( s \) is effectively bound by \( L_s \) for \( \theta \). In other words, for an execution prescribed by \( \theta \), stream \( s \) never holds more than \( L_s \) elements, setting an upper bound on its memory requirements.

This suggests that we can first append back-pressure dependences that reproduce this behaviour to the dataflow dependences in (4.1) and only then look for a schedule. To do this, we fix a maximum size \( L_s \) for stream \( s \) and enforce all indices \( i, i \leq j - L_s \), to be consumed before \( j \) is written\(^1\), i.e., we introduce a dependence from a task \( t \in D_S \) to a task \( t' \in D_{S'} \) whenever for some \( c \in R_S \) and \( c' \in W_{S'} \) accessing \( s \), we have some

\(^1\)This assumes, as laid out in section 4.2, that every written position has a designated consumer.
i ∈ [f^S_c(t), l^S_c(t)] and j ∈ [f^{S'}_{c'}(t'), l^{S'}_{c'}(t')], such that i ≤ j − L_s, which is equivalent to:

\[ \exists (c, c') ∈ R_S × W_{S'} : a^S_c(t) = a^{S'}_{c'}(t') = s ∧ h^S_c(t), h^{S'}_{c'}(t') ≥ 1 ∧ f^S_c(t) ≤ l^{S'}_{c'}(t') − L_s. \]  

(6.1)

Clearly, \( L_s \) has to be greater than or equal to the maximum burst of any task accessing \( s \), otherwise there is at least one producer depending on its own consumer, thus a cycle in the task graph.

Similarly to the dataflow dependence sets in (4.2), back-pressure dependence sets between \( S \) and \( S' \) for some \( c ∈ R_S \) and \( c' ∈ W_{S'} \) may also be defined:

\[ \mathcal{B}_{c-c'}^{S,S'} = \left\{ t → t' ∈ D_S × D_{S'} | a^S_c(t) = a^{S'}_{c'}(t') ∧ h^S_c(t), h^{S'}_{c'}(t') ≥ 1 ∧ f^S_c(t) ≤ l^{S'}_{c'}(t') − L_s \right\}. \]  

(6.2)

Under the constraints from section 4.4 defining OpenStream’s polyhedral fragment, any back-pressure dependence relation (6.2) is contained in \( \mathbb{Z}^n × \mathbb{Z}^{n'} \), where \( n \) (resp. \( n' \)) is the number of loops surrounding \( S \) (resp. \( S' \)). However, as for dataflow dependence relations (4.2), these do not necessarily define unions of polyhedra but of (not necessarily, but possibly polyhedral) semi-algebraic sets. Nonetheless, in this chapter, unlike in chapter 5, we do not try to evade polynomial dependence descriptions. Instead, we study a stream bounding technique that can cope with them.

We start by considering the union of all the dependences captured by both (4.2) and (6.2). Then, Feautrier’s approach, as summarised in section 3.4.2, can be leveraged to seek a polynomial schedule with bounded streams. Since we allow polynomial dependences, we need to leverage Schweighofer’s Theorem (see section 3.4.1). Although this theorem’s full statement requires the affine subset of the problem’s constraints to define a compact polyhedron, we do not enforce such restriction on the statement iteration domains. From a pragmatic point of view, if the algorithm produces a schedule, then it fulfils the required positivity conditions, whether the hypotheses of the theorem are satisfied or not.\(^2\)

Moreover, deciding on the existence of a schedule is undecidable. This follows as the converse problem, on determining the existence of deadlocks, is also undecidable, as proven in [CDF16]. Furthermore, even though every schedule is writable as a polynomial\(^3\), Feautrier’s technique, as explained in section 3.4.1, does not cover the whole

\(^2\)Recall compactness is only required by one of the ‘if and only if’ biconditional directions.

\(^3\)Polynomial interpolation through the logical date scheduled for each task is always possible for a polynomial of high enough degree.
6.1. BACK-PRESSURE DEPENDENCES

space of polynomials. However, if the exploration is successful, the mere existence of the retrieved schedule is sufficient to guarantee a runtime execution implementing these dependences will not deadlock [CDF16]. On the contrary, we are forced to conservatively rule out the possibility of executing under the chosen bounds, on the grounds the program might deadlock.

The following example illustrates the concept of back-pressure dependences while demonstrating the consequences it might have on hindering concurrency.

```plaintext
stream s;
for (int k = 0; k < N; ++k) {
    task write (s, win_w(2)) do ; // W
    task read (s, win_r(2)) do ; // R
}
```

Listing 6.1: SCoP containing two task creation statements whose instances, for each statement, are independent. The window first indices are given by $f^w_W(k) = 2k$. The program expresses simple dataflow patterns, $D^W_W \times D^R_R = \{k \rightarrow k' \in D^W_W \times D^R_R \mid k = k'\}$. Hence, it admits a schedule $\theta^a$ with trivially-parallel writes $\theta^a_W(k) = 0$, and, independently, reads $\theta^a_R(k) = 1$. However, when stream $s$ is bounded by $L_s = 2$, the new back-pressure dependences

$$B^R_R \times W^W_W = \{k' \rightarrow k \in D^R_R \times D^W_W \mid \alpha^R_R(k') = \alpha^W_W(k) \land f^R_R(k') \leq l^W_W(k) - L_s\}$$

$$= \{k' \rightarrow k \in D^R_R \times D^W_W \mid 2k' \leq 2(k + 1) - 2\}$$

$$= \{k' \rightarrow k \in D^R_R \times D^W_W \mid k' \leq k - 1\}$$  \text{(after gcd tightening)} \hspace{1cm} (6.3)

constrain the execution of the task graph to a purely sequential schedule $\theta^b$ such that $\theta^b_W(k) = 2k$ and $\theta^b_R(k) = 2k + 1$. Nevertheless, the schedule’s existence warrants execution.

Figure 6.1 illustrates how the back-pressure dependences introduced for $L_s = 2$ augment the existing dataflow dependences and force the execution of the $k^{th}$ instance of W to succeed the $(k - 1)^{th}$ instance of R and precede the $k^{th}$ instance of R.

Since the previous example fits within traditional Farkas’ Lemma techniques, let us, for completion, study a program requiring a polynomial schedule. Consider a slight variation where both tasks are instead enclosed by a triangular loop nest:
CHAPTER 6. SCHEDULING FOR STREAM BOUNDING

![Task creation statement: \( W \) \( R \)
Dependences: \( \text{w} \to \text{r} \) \( \text{r} \to \text{w} \)]

Figure 6.1: Fragment of the dependence task graph for the example in listing 6.1. This depicts the back-pressure dependences (dashed) as defined by (6.2) for \( L_s = 2 \) and the dataflow dependences (solid) given by (4.2). Not all back-pressure dependences are shown since they are subsumed by transitivity of the dependences drawn.

Listing 6.2: Triangular SCoP containing two task creation statements whose instances, for each statement, are independent. The window first indices are now polynomial on the loop iterators, \( f_W(w(i, j)) = f_R(r(i, j)) = i(i - 1) + 2j \).

The dataflow patterns, even though they are still easily seen to encode a polyhedron, must a priori be written as:

\[
\mathcal{D}_{W \times R}^W = \{ (i, j) \to (i', j') \in \mathcal{D}_W \times \mathcal{D}_R \mid i(i - 1) + 2j = i'(i' - 1) + 2j' \}. 
\]

A scheduler based on Schweighofer’s Theorem still finds \( \theta^u \) with trivially-parallel writes \( \theta_W^u(i, j) = 0 \), and, independently, reads \( \theta_R^u(i, j) = 1 \). Meanwhile, when stream \( s \) is bounded by \( L_s = 2 \), the new back-pressure dependences,

\[
\mathcal{D}_{R \times W}^R = \{ (i', j') \to (i, j) \in \mathcal{D}_R \times \mathcal{D}_W \mid i'(i' - 1) + 2j' \leq i(i - 1) + 2j - 1 \},
\]

constrain the execution of the task graph such that a purely sequential schedule \( \theta^b \)
may be found\textsuperscript{4}, $\theta_W^b(i, j) = 2i(i - 1) + 4j$ and $\theta_R^b(i, j) = 2i(i - 1) + 4j + 1$. Note that $\theta^a$ can be found by an instantiation of Feautrier’s scheduling algorithm with products of constraints of order zero and that $\theta^b$ requires order one products, despite being a polynomial of degree two on the loop iterators (see section 3.4.2).

Next, we study a few example programs that illustrate limitations of this procedure. We work with purely polyhedral OpenStream programs, i.e. those tractable under Farkas’ Lemma, demonstrating these limitations materialise even in simple cases.

### 6.2 The Problem with Non-Causal Schedules

A causal schedule $\theta^{cs}$ is such that writes to the same stream occur in the same order as their window first indices, i.e., in the same order as the creation of the corresponding tasks. In other words, if $t \in D_S$ and $t' \in D'_S$ are two producers on the same stream and $t$ precedes $t'$ in the control program then $\theta^{cs}_S(t) < \theta^{cs}_S(t')$:

$$t < t' \land \exists (c, c') \in W_S \times W'_S. a^S_c(t) = a^S_{c'}(t') \land h^S_c(t), h^S_{c'}(t') \geq 1 \rightarrow \theta^{cs}_S(t) < \theta^{cs}_S(t').$$

(6.4)

For instance, the schedule $\theta^a$ above is not causal but $\theta^b$ is. To see why programs not admitting such schedules potentially pose a problem, consider the following example:

```plaintext
stream s;

task read (s, win_r) do ; // SNK

for (k = 1; k < N; ++k)
    task read (s, win_r) write (s, win_w) do ; // WRK

task write (s, win_w) do ; // SRC
```

Listing 6.3: SCoP whose tasks must execute in strict non-causal order. The three task creation statements have unit horizons and first indices $f^{SNK}_r = 0$, $f^{WRK}_r(k) = k$, $f^{WRK}_w(k) = k - 1$ and $f^{SRC}_w = N - 1$.

\textsuperscript{4}This schedule is valid but not optimal, leaving ‘holes’ in the target space. For $i' = i$, the backpressure constraint boils down to $2j' \leq 2j - 1$, which is not gcd-tight. Thus, $\theta^b_W(i, j) = i(i - 1) + 2j$ and $\theta^b_R(i, j) = \theta^b_W(i, j) + 1$, cannot to be found. The issue of finding optimal schedules is left as future work.
The dataflow dependences,
\[
\begin{align*}
\mathcal{D}_{\text{wrk}}^{\text{src}} &= \{ t_{\text{src}} \rightarrow N - 1 \in \mathcal{D}_{\text{src}} \times \mathcal{D}_{\text{wrk}} \}, \\
\mathcal{D}_{\text{wrk}}^{\text{wrk}} &= \{ k \rightarrow k - 1 \in \mathcal{D}_{\text{wrk}}^2 \}, \\
\mathcal{D}_{\text{wrk}}^{\text{snk}} &= \{ 1 \rightarrow t_{\text{snk}} \in \mathcal{D}_{\text{wrk}} \times \mathcal{D}_{\text{snk}} \},
\end{align*}
\]
depicted in the task graph in fig. 6.2, force the tasks in this program to be sequentially executed in the reverse order of their creation. Thus, this particular program only admits strict non-causal schedules \( \theta^{\text{nc}} \), whose behaviour is such that, for arbitrary tasks \( t \in D_S \) and \( t' \in D_{S'} \):
\[
t < t' \land \exists (c, c') \in W_S \times W_{S'}. \ a^S_c(t) = a^{S'}_{c'}(t') \land h^S_c(t) \geq 1 \rightarrow \theta^{\text{nc}}_{S'}(t') < \theta^{\text{nc}}_S(t).
\]

One such schedule would be: \( \theta^{\text{nc}}_{\text{snk}} = N \), \( \theta^{\text{nc}}_{\text{wrk}}(k) = N - k \) and \( \theta^{\text{nc}}_{\text{src}} = 0 \). However, any stream-bounding dependence of the type of (6.2) requires at least ‘partial causality’. In other words, given a stream \( s \), for all indices \( i \leq j - L_s \) to be consumed before \( j \) is written, the dataflow dependences (4.2) also imply they have to be produced before \( j \) is written.

Let \( (t_{\text{prod}}, c) \in D_S \times W_S \) for some statement \( S \), be the task and its access such that
\[
\begin{align*}
a^S_c(t_{\text{prod}}) &= s \land h^S_c(t_{\text{prod}}) \geq 1 \land f^S_c(t_{\text{prod}}) \leq i \leq l^S_c(t_{\text{prod}}),
\end{align*}
\]
and let \( (t'_{\text{prod}}, c') \in D_{S'} \times W_{S'} \) for some \( S' \), be the task and its access such that
\[
\begin{align*}
a^{S'}_{c'}(t'_{\text{prod}}) &= s \land h^{S'}_{c'}(t'_{\text{prod}}) \geq 1 \land f^{S'}_{c'}(t'_{\text{prod}}) \leq j \leq l^{S'}_{c'}(t'_{\text{prod}}).
\end{align*}
\]

Then, since \( i \leq j \) and streams have the DSA property, we also have \( f^S_c(t_{\text{prod}}) \leq f^{S'}_{c'}(t'_{\text{prod}}) \) which, via item (W2) in chapter 4, implies \( t_{\text{prod}} \preceq t'_{\text{prod}} \). If \( t_{\text{prod}} \neq t'_{\text{prod}} \), we recover the premiss in (6.4) for \( (t_{\text{prod}}, t'_{\text{prod}}) \). Finally, let \( (t_{\text{cons}}, c') \in D_{S''} \times R_{S''} \) for some statement \( S'' \), be a \( \delta \) task and its access such that
\[
\begin{align*}
a^{S''}_{c'}(t_{\text{cons}}) &= s \land h^{S''}_{c'}(t_{\text{cons}}) \geq 1 \land f^{S''}_{c'}(t_{\text{cons}}) \leq i \leq l^{S''}_{c'}(t_{\text{cons}}),
\end{align*}
\]
Then,
\[
(t_{\text{prod}}, t_{\text{cons}}) \in \mathcal{D}^{S \times S'}_{c \rightarrow c'} \land (t_{\text{cons}}, t'_{\text{prod}}) \in \mathcal{D}^{S' \times S'}_{c' \rightarrow c'}.
\]
---

This task is not necessarily unique since multiple tasks might peek at the location.
6.2. THE PROBLEM WITH NON-CAUSAL SCHEDULES

![Diagram of dependence task graph]

This depicts both (a subset of) the back-pressure dependences for each producer task for $L_s \in \{1, 2, 3, N - 1, N\}$ (dashed) and the dataflow dependences (solid). The back-pressure dependences for a given $L_s$ are those depicted for its respective diagram and all those on diagrams to its right, i.e. larger $L_s$.

which requires

$$\theta_S(t_{\text{prod}}) < \theta_{S'}(t'_{\text{prod}})$$

as in (6.4) but only for a subset of all tasks.

This cannot happen in the present program and stream bounding back-pressure dependences will add a cycle to the task graph in fig. 6.2 and introduce a deadlock.

The pitfall is that $s$ can, in fact, be bounded. In particular, for the schedule provided above, the stream is bounded by $L_s = 2$. We note that, although in between the execution of any two tasks, stream $s$ is populated by a single element, we cannot state $L_s = 1$ is a valid bound: when the $k^{th}$ task of WRK is scheduled for execution, element $s(k - 1)$ is due to be written while element $s(k)$, despite due to be read, is still taking memory space. Since tasks execute atomically, storage availability must be assessed before any consumption takes place. Thus, if a producer relies on itself to free the stream, as is the case if $L_s = 1$, the program deadlocks.
The current strategy fails to bound the stream in this program due to its implicit assumption that a read from stream \( s \) in position \( i \) does not have a dataflow dependence, either direct or transitive, on a write on a position \( j \geq i + L_s \). Since \( t_{SNK} \) reads from \( s(0) \) and \( t_{SRC} \) writes to \( s(N-1) \) and the former depends transitively on the latter, satisfying the assumption requires \( L_s \geq N \), i.e. a stream large enough to hold the channeled data in its entirety. Streams must filled in a (partial) causal order which demands writes within \( L_s \) of the current smallest live position\(^6\), before any write beyond such region.

Some OpenStream programs, however, in particular those which necessarily require strict non-causal schedules are unable to comply. ‘Spurious’ deadlocks can be introduced even for programs where streams can be effectively bounded, as a result of artificial back-pressure dependences that, besides bounding streams, implicitly enforce ‘partial’ causality.

### 6.3 Minimising Stream Sizes

Armed with a strategy to check execution feasibility with bounded streams, the programmer could be interested in determining the size \( L_s \) of each stream \( s \) such that the global surface of the streams is minimised. In other words, naming \( u_s \) the size of each element of \( s \), we seek the minimum of \( \sum_s L_s u_s \), such that there is no cycle in the dependence task graph. Since the conclusions of this section do not change if all streams \( s \) share the same element size, \( u_s = u \), we instead consider the minimisation of \( \sum_s L_s \).

Unfortunately, reformulating the question as a multi-objective problem on the size of each individual stream can, at best, give a Pareto optimal solution\(^7\). In other words, the sizes of different streams are not independent, and therefore cannot be optimised separately and successively, one after the other, while still obtaining a solution minimising all stream sizes simultaneously. On the bright side, the solution to the original optimisation problem, due to its particular simple form as a sum of the stream sizes, is always one of the Pareto optimal solutions of the relaxed problem. Consider the following example:

---

\(^6\)By live we mean a stream element not yet consumed even if not yet produced.

\(^7\)A solution whose improvement in one of the problem’s objectives degrades some other, conflicting, objective.
6.3. MINIMISING STREAM SIZES

```plaintext
stream s, t;

task write (s, win_w(2)) do ; // WS

task write (t, win_w(3)) do ; // WT

task write (s, win_w) read (t, win_r(3)); // WSRT

task write (t, win_w(2)) read (s, win_r(2)); // RSWT

task read (s, win_r) do ; // RS

task read (t, win_r(2)) do ; // RT
```

Listing 6.4: SCoP containing task creation statements accessing two streams in an intertwined manner. All horizons are unitary and first indices are as follows: $f_W^{WS} = 0$, $f_W^{WT} = 0$, $f_W^{WSRT} = 2$, $f_T^{WSRT} = 0$, $f_T^{RSWT} = 3$, $f_T^{RS} = 0$, $f_T^{RT} = 3$.

If the size for stream $s$ is minimised first, noticing that $t_{WS}$ requires $L_s \geq 2$ given its window’s horizon, we find the smallest bound $L_s = 2$ is a solution only if $L_t \geq 5$. The bound on stream $s$ introduces a back-pressure dependence which serialises the execution of $t_{WSRT}$ after $t_{RSWT}$, requiring a large enough bound on stream $t$ to prevent the introduction of the reverse dependence, as seen in fig. 6.3 (left). Conversely, if the size of stream $t$ is optimised first, $t_{WT}$ requires $L_t \geq 3$, and we find $L_t = 3$ is feasible for $L_s \geq 3$. Figure 6.3 (centre) shows the corresponding dependence task graph. The back-pressure dependence serialises $t_{RSWT}$ after $t_{WSRT}$.

For this program these are the only two Pareto minimal solutions which can be determined by minimising a stream at a time. In general, even for programs with only two streams, there can be several Pareto minimal solutions, some of which possibly not minimal for any of the stream sizes. As an example, assume a program with two streams, for which we determine there is a schedule if and only if: 1) $L_s \geq 2$ and $L_t \geq 5$, or 2) $L_s \geq 3$ and $L_t \geq 3$, or 3) $L_s \geq 6$ and $L_t \geq 1$. Of the three Pareto minimal solutions, the bounds for 2) minimise $L_s + L_t$, but they are not minimal for either stream.

Let us return to the example in listing 6.4 and fig. 6.3. Next, we show how, in addition, it is possible that none of the identified solutions absolutely minimises the global stream surface. In fact, although within the scope of the back-pressure dependences strategy introduced in section 6.1 the solutions 1) $L_s \geq 2$ and $L_t \geq 5$ or 2) $L_s \geq 3$ and $L_t \geq 3$ are both Pareto minimal and the latter apparently minimises the global stream
surface, we can easily show the back-pressure dependences limit the searchable solution space. In reality, even though a situation where $L_s = 2$ and $L_t = 3$ apparently introduces a deadlock, as seen in fig. 6.3 (right), this is a ‘spurious’ deadlock of the type discussed in section 6.2. For a (non-causal) schedule which executes each of the two (unilaterally) connected components of the dataflow dependence task graph one after the other, these bounds are both respected: sequentially execute $t_{WS}$, followed by $t_{RSWT}$ and $t_{RT}$, and finally $t_{WT}$, followed by $t_{WSRT}$ and $t_{RS}$.

Implementing this behaviour, however, would present further problems. Notice that simply restricting streams $s$ and $t$ to a maximum of $L_s = 2$ and $L_t = 3$ elements at runtime would not be enough: since $t_{WS}$ and $t_{WT}$ are free to execute straightaway, as soon as they do, there is a resource deadlock.

Furthermore, one should note that the determined global stream surface does not correspond to the actual memory usage of the program. Notice that, even if our strategy were able to identify $L_s = 2$ and $L_t = 3$ as the global stream surface minimiser, the sequential schedule described above provides an example of an execution where streams are never simultaneously fully utilised. In other words, at no point during the execution do stream $s$ and stream $t$ simultaneously hold two and three elements, respectively. This shows that, even if a minimum global stream surface were to be found, this would still be a conservative approximation of the actual memory requirements of the program.
6.4 Guidelines for Practical Applicability

Despite the undecidability of the scheduling problem and the ‘spurious’ constraints introduced by the stream bounding strategy, an approach can still be devised through which, given the resource constraints of a particular target architecture, one can, conservatively, decide on the execution feasibility of a particular OpenStream dataflow program.

Consider a memory constrained device, e.g., an FPGA, with some memory of total size $M$ and an OpenStream program with $n$ streams. We first build the space of candidate tuples $(L_s)_{1 \leq s \leq n}$ such that $\sum_{s=1}^{n} L_s u_s \leq M$ where $u_s$, the size of each element of $s$, is assumed statically known and such that each stream can hold at least as many items as its greatest burst, i.e., $L_s \geq \max_b b^S(t)$ for all $(S,t,c)$ accessing $s$. If these burst lower bounds on $L_s$ are statically known, we get a polyhedron in $\mathbb{N}^n$. Otherwise, in general, since the bursts may be polynomial, the lower bounds for $L_s$ can also be polynomial on the structure parameters of the program. However, we do not concern ourselves with this problem here since, given fixed $M$, we are usually interested in determining if a program of a predefined problem size is able to execute.

Assuming viable solutions exist, there must always be some on or neighbouring the limiting hyperplane $\sum_{s=1}^{n} L_s u_s = M$, where the componentwise maximal integer solutions lie. Accordingly, a traversal of the polyhedron should start from this region, on the grounds that if no solutions exist here, no solution can exist for this $M$. Next, we build the back-pressure dependence relations for one of these combinations by leveraging (6.2) and take the union with the dataflow dependences (4.2). Finally, we apply Feautrier’s approach [Fea15], section 3.4.2, in order to find a schedule.

Once a schedule is constructed, we know that if we enforce the same dependences at runtime, by keeping track of the current smallest live index $i_s$ for every stream $s$ and forbidding writes from $i_s + L_s$ onwards, the streams are implemented as bounded streams in accordance to the combination for which the schedule was built, thereby guaranteeing successful execution on said device. If a schedule is not found, we can either try another combination of stream bounds or try to increase the order of the polynomial schedule (see section 3.4.2). If the search for a polynomial schedule is unsuccessful after a predefined time limit has elapsed or after exhausting, up to a certain order, all the stream bound possibilities, we conservatively conclude on no execution. In fact, as far as we were able to determine, the program might deadlock.

If the tasks’ working data and communicated data share hardware resources, it is possible to take into account the requirements of tasks’ work functions, if these are
known statically, as it is likely the case for FPGA kernels. In that case, the method just outlined is still applicable by replacing $M$ with $M' = M - M_{wf}$ where $M_{wf}$ are the collective memory requirements of the tasks’ work functions.

Constraining a dataflow program to run on such devices can, however, diminish the amount of parallelism available. Let us reconsider the example in section 6.3. If $M = 8$, we may set $L_s = 3$ and $L_t = 5$, essentially making the program free from back-pressure dependences. The program can then, for example, execute in three waves: first $t_{WS}$ and $t_{WT}$ simultaneously; then $t_{RSWT}$ and $t_{WSRT}$; and lastly $t_{RS}$ and $t_{RT}$. If, however, $M = 6$, the only possibility is, within the proposed back-pressure dependence strategy, $L_s = L_t = 3$ which, as we have seen, serialises the execution of $t_{RSWT}$ after $t_{WSRT}$.

On the other hand, these memory constrained devices may be massively parallel. Thus, even if some parallelism is lost due to memory constraints, the parallelism opportunities still available might greatly surpass the capabilities of many-core CPUs. In addition, the specialised hardware they expose to the programmer might also allow specially designed, faster implementations of tasks’ work functions, thereby compensating for the restrictions imposed by the constrained memory resources.

### 6.5 Summary

We showed how augmenting polyhedral OpenStream programs with back-pressure dependences can be used to bound streams and statically, albeit conservatively, decide if said programs can be executed on devices with limited memory. We described how the current method is limited in its power due to its undecidable nature, the introduction of ‘spurious’ deadlocks, the difficulty of finding a global stream surface minimiser and the overestimation of actual memory usage.

In the literature, a few studies exist suggesting similar trial and error approaches to that of section 6.4. For instance, in the context of Kahn Process Networks (KPNs), the authors in [Hom01] arbitrarily bound the capacities of the buffers and simulate the execution to verify if no deadlocks appears. In [Par95], the author developed a greedy heuristic that increased the capacity of a buffer every time a deadlock appears.

In general, exactly solving the problem is hard. For example, minimising the global stream surface has been proven NP-complete for a subset of KPNs with restricted functionality, namely finite tasks with a statically known sequence of read and write operations, and as few as three tasks [MKN05].

Likewise, buffer minimisation for Synchronous DataFlow (SDF) and Cyclo-Static
DataFlow (CSDF) graphs has also been proven NP-complete [BLM96]. Furthermore, under real-time throughput constraints, it is known NP-hard [MMK10]. The work for SDF in [BMMU10] and CSDF in [BMK13] proposed approximations to find sufficient, close to optimal buffer sizes, in polynomial time. The bounded capacity of buffers is modelled, for every FIFO arc in the original graph, using a backward arc whose tokens represent the buffer’s free spaces.

Unfortunately, it is not clear how to transfer the techniques devised in these efforts to OpenStream. All the mentioned frameworks work on single-producer, single-consumer buffers, which is not the case for OpenStream. In a sense, the back-pressure dependences used here are an attempt to bring the backward FIFO arc construction to the realm of OpenStream, by forcing the eviction of data in lower indices before writing higher indices. As shown, in the case of OpenStream, this is an over-approximation of the problem: despite sufficient to guarantee the memory footprint of a stream is bounded to a fixed capacity, it is not a necessary requirement. Indeed, we studied programs for which the smallest valid stream bounds cannot be modelled by the proposed strategy due to the introduction of ‘spurious’ deadlocks.

This leaves a few options to the programmer willing to exploit data-centric programming models: (1) resort to models with restricted expressive power such as SDF and CSDF that statically provide deadlock-absence and bounded memory guarantees [PPL95]; (2) resort to approaches such as the one described here to, in practice, albeit conservatively, certify the execution of more expressive dataflow programs on devices with constrained memory resources; or (3) rely on automated process network generation from static control loop nests initially written in an imperative language, such as Verdoolaege’s Polyhedral Process Networks [Ver10b] which allow conservative estimation of the required buffer sizes for a given schedule, but are limited to polyhedral-amenable code.

This choice will probably be ultimately guided by the application domain, but we believe that current and upcoming architectures require dataflow models that are more expressive — even than the polyhedral subset of OpenStream defined in chapter 4 and used here — allowing irregular, dynamic communication patterns and the execution of non-polyhedral code, e.g., as part of tasks’ work functions. In this sense, the presented work builds on this insight and motivates future work in this direction.
Chapter 7

Conclusion and Perspectives

This chapter briefly outlines the work presented in this thesis and provides a discussion of directions and opportunities for future research.

7.1 Summary of the Thesis and its Contributions

Chipmakers show no sign of slowing down and the trend is for the industry to continue to provide a growing number of diverse hardware resources on a single semiconductor die, once the home of the lone uniprocessor. Hence, the software industry is under unprecedented pressure to provide better programming models, libraries and compilers to aid programmers efficiently exploit the impressive computational power at their disposal.

This work is motivated by two main observations. On the one hand, task-parallel programming models are a rising solution to express concurrent computational activity with potentially good performance and portability across architectures, albeit often times at the expense of the programmer’s productivity. On the other hand, high-level compilation techniques based on the polyhedral model allow the programmer to benefit from their analysis power and transformation expressiveness while, simultaneously, sparing them from learning yet another language or considering the details of the target architecture. An automatic tool is responsible for the algebraic heavy lifting, directly returning optimised code or the result of some analysis exercise.

In particular, we targeted streaming dataflow programs written in the CDDF Open-Stream language, since they differ from more prevalent shared-memory models in a key productivity feature, determinism, which also greatly simplifies static analysis. Unfortunately, the same mechanism that deterministically attributes stream indices to
each task, also obscures data access patterns. Compared to the traditional scope of the polyhedral model, where dependence analysis is a problem of colliding array accesses explicitly stated in the program’s text, dependence analysis in OpenStream requires obtaining closed forms for the access indices. Incidentally, even for polyhedral control flow, these indices are commonly non-affine but rather higher degree polynomials on the loop iterators and the structure parameters of the program, hampering traditional techniques.

The two major contributions of this thesis take a step towards adopting polyhedral techniques in the race for task-parallel programming productivity.

By restricting applicability to single, perfectly nested task creation statements fitting the polyhedral subset of OpenStream, disallowing polynomial dependences and imposing uniform dependence vectors, we devised an algorithm for:

**Task Fusion** with the ability to generate coarser-grained tasks from an initial finer-grained and programmer-friendly OpenStream specification. By leveraging well-known tiling heuristics from PLuTo \([\text{BBK}^+08, \text{BHRS}08]\), the output is a program with improved data reuse and reduced communication volume and runtime synchronisation overhead.

In addition, we devised a strategy which, despite only suitable for polyhedral and ‘sufficiently causal’ OpenStream programs can, even in the presence of polynomial dependences, conservatively provide:

**Bounded Scheduling** guarantees by issuing certificates for execution on memory constrained hardware. By leveraging Schweighofer’s Theorem \([\text{Sch}02]\) and Feautrier’s scheduling algorithm \([\text{Fea}15]\), we search for a (possibly polynomial) schedule, whose existence certifies the absence of deadlocks when the capacity of streams is bounded.

### 7.2 Future Work

We envision three main lines for future endeavours building on the work laid out in this thesis. The first is simply the engineering effort of:
Implementing the task fusion algorithm and a battery of benchmarks The presented technique lacks thorough experimental evaluation and, therefore, a framework to fully validate the benefits of tuning task granularity. A complete toolset could, for instance, comprise:

**lexer and parser** to extract the iteration domains and stream access information for each statement in a static control program (for a starting point see the compiler for OpenStream [PP20]);

**isl/barvinok** [VSB+07, Ver10a] to obtain the first indices of the stream access windows and other required polyhedral projections and cardinals;

**PLuTo** [BBK+08, BHRS08] to tile the control program;

**code generator** to re-orchestrate the stream accesses and the work function of each new fused task creation statement.

We leave two further suggestions. First, code generation for the work function of the fused tasks may be improved by leveraging the developments summarised in section 3.3.2. Second, we remark that the PolyBench/C benchmark suite [PY16] of numerical computations is probably the most adequate source from which the initial, fine-grained OpenStream versions of amenable programs may be written. Since OpenStream is currently limited to one-dimensional stream arrays, parametric versions of classical routines, e.g. the matrix-matrix product kernel, which lend themselves naturally to two-dimensional data structures, might quickly lead to non-amenable programs. In these circumstances, a non-parametric version might still be valuable, particularly once a compilation flow which can automatically regenerate code is created.

A second line of exploration could aim at incremental improvements to the current contributions. Particularly, in the context of our proposed task fusion strategy.

**Polyhedral transformations of tasks’ work functions** Since the body of each OpenStream task is an imperative (typically compound) statement, direct application of the well-known polyhedral techniques from chapter 3 may improve the performance of individual tasks. Although the application of such transformations does not require previous fusion, its relevance increases with the amount of work sequentialised inside each task. For example, for sufficiently sized problems, hierarchical blocking might be advantageous. Indeed, a fused task could work on data fitting in a higher level
7.2. FUTURE WORK

cache, while non-parallel loop tiling internal to the task could work on a volume of data suitable for a lower level cache.

**Multiple and imperfectly nested task creation statements** Similarly to loop tiling, which is possible even in the presence of multiple, imperfectly nested statements [BHRS08], it would be interesting to consider fusion across tasks created by distinct statements. This would unlock data locality optimisations across distinct computational kernels, typically residing on different tasks. A rework of the dependence handling steps of the procedure in chapter 5 would be necessary since considering inter-statement dependences breaks assumptions in step 8 of the algorithm. The following generalisation provides a starting point by removing the prerequisite for uniform dependences.

**Breaking free from uniform dependences** The presented task fusion algorithm relies on uniform dependence vectors to exhaustively enumerate and create dependence polyhedra for each inter-tile dependence direction. This is stricter than necessary. In reality, once the program’s dataflow dependences are written in terms of the new iterators, a projection onto the source and target tile iterators would reveal the inter-tile dependences. The range of the resulting relation contains, for each source tile, the iteration vectors of each target tile, for each of which a separate stream may be built. There is no requirement for uniform dependences.

**Avoiding communication of duplicated data** It is possible that after task fusion, dependences previously targeting peek and read accesses accessing the same stream indices from different tasks now fall on the same fused task. As our task fusion method designates a new stream per cross-tile dependence polyhedron, this may result in (partial) duplication of the data flowing in these new streams. Instead, a unique stream per tile pair could potentially transport the union of the data pertaining to any stream currently associated to such pair. Besides reducing the volume of communicated data, this optimisation also decreases the number of new streams, potentially reducing runtime synchronisation overhead. Note that horizons should now reflect the cardinal of the union and the work function of the fused task would have to be adjusted to make sure each iteration accesses its rightful data.

The third line of study is perhaps more research-driven and explores uncharted territory both in the context of OpenStream high-level transformations and bounded scheduling.
**Transformations exclusive to the control program** The aim is to improve performance by reordering task creations in such a way that dependent tasks are revealed to the runtime system in close time proximity, thereby also inciting close execution schedules. The challenge is to morph the control program whilst preserving the task dependences and, thus, the task graph. Fortunately, achieving this does not imply conserving stream indexing. As a matter of fact, it suffices that all matching producer and consumer access windows are equally shifted along the stream, so they still match. For instance, pure loop tiling of the Gauss-Seidel kernel of chapter 5 is perfectly valid for the given affine hyperplanes since the use of a stream array means stream indexing remains unaltered. Moreover, for the same reason, independent tiles may be concurrently scanned by the control program, exposing wavefront task creation parallelism. Tasks keep their granularity but are created in blocks of dependence paths in the task graph.

**Task placement mappings** For each dependence between task instances assigned to distinct processors, a data communication is generated. In the best case scenario, this only involves a cache transfer but, in the case of distributed NUMA machines, this may involve an exchange between different nodes. Thus, it is desirable to keep these costly operations to a minimum with a good task placement mapping. The advantage of this approach is that, since a runtime system is responsible for dynamically allocating each task to a processor, this mapping can be a polynomial in the control program loop iterators and parameters. The evaluation of this polynomial can then be attached to each task creation and presented to the runtime as a placement hint for that task. This optimisation could be implemented as a drop-in alternative for the runtime NUMA placement heuristics developed in [Dre15].

**Computing memory footprints for feasible schedules** Bounding streams using back-pressure dependences is an over-constraining strategy. In fact, it leads to ‘spurious’ deadlocks and incomplete characterisations of programs whose streams are effectively bounded. Alternatively, given a space of schedules satisfying the dataflow dependences of the program, one could seek to invert these schedules. The result is a relation over which the cumulative difference of write and read bursts, i.e. the stream’s current memory usage, could be computed as a function of the runtime timeline. The global maximum of this function, if finite, is the effective stream bound. For affine and low
7.2. FUTURE WORK

degree schedules this seems practical but, based on previous work [CDF16], impossible in the general case.
Index of Notation

Program Structure

\( S(\alpha) \)  \( \alpha \)-th iteration of statement \( S \)

\( t, p \)  iterators for sequential (time) and parallel (placement) loops

\( i_t, i_T, B_i \)  intra-tile iterator, inter-tile iterator and tile size along the same direction

\( N_{ST} \)  number of common loops surrounding both statements \( S \) and \( T \)

\( \prec \)  lexicographic order, task creation order

\( t_S \)  if statement \( S \) creates a single task, this is its sole task

\( L_s \)  upper bound on the number of elements that stream \( s \) can hold at any time

\( u_s \)  size of each element of stream \( s \)

Iteration Domains

\( D_S \)  \textbf{iteration domain} for statement \( S \): set of all iterations or tasks created by \( S \)

\( D_{S\circ} \)  target, transformed (possibly tiled) iteration domain for statement \( S \)

\( D_{T,S\circ} \)  inter-tile projection of a tiled iteration domain \( D_{S\circ} \)

\( D_{i,S\circ} \)  intra-tile projection of a tiled iteration domain \( D_{S\circ} \)

Memory References

\( C, R, W \)  sets of all, read and write array references or stream accesses in the program

\( C_S, R_S, W_S \)  sets of all, read and write array references or stream accesses for statement \( S \)

\( A \)  set of all arrays in the program

\( r, w \)  symbols for read and write array references
\( a^S_c \) array access function for array reference \( c \), statement \( S \): 0-ary function giving the array for array reference \( c \), statement \( S \)

\( i^S_c \) index access function for array reference \( c \), statement \( S \): function from iteration to array indices for array reference \( c \), statement \( S \)

\( S \) set of all streams in the program

\( p, r, w \) symbols for peek, read and write stream accesses

\( r_s, w_s \) read and write positions on stream \( s \)

\( a^S_c \) stream access function for stream access \( c \), statement \( S \): function from task to stream for access \( c \), statement \( S \)

\( a^{\text{st}}_{t,c} \) stream access function from intra-tile iteration to stream for access \( c \), statement \( S \)

\( A^{\text{st}}_{T,c} \) stream access relation from tiles to their streams for access \( c \), statement \( S \)

\( A^{\text{st} \times T}^{\text{st}}_{T,c,d} \) stream access relation from the tiles in ran \( P^{\text{st} \times T} c,d \) to those streams in access \( d \), statement \( T \) accessed by intra-tile iterations also in ran \( P^{\text{st} \times T} c,d \)

\( w^S_c \) access window for stream access \( c \), statement \( S \): function from task to first index, burst and horizon for access \( c \), statement \( S \)

\( f^S_c, l^S_c \) first and last indices for stream access \( c \), statement \( S \): functions from task to the positions of the first and last accessed stream elements, respectively, for access \( c \), statement \( S \)

\( f^{\text{st}}_{c,t}, l^{\text{st}}_{c,t} \) first and last index functions from tiled-space iterators to first and last accessed stream elements, respectively, for access \( c \), statement \( S \)

\( f^{\text{st}}_{t,c}, l^{\text{st}}_{t,c} \) first and last index functions from intra-tile iteration to first and last accessed stream elements, respectively, for access \( c \), statement \( S \)

\( F^{\text{st}}_{T,c}, L^{\text{st}}_{T,c} \) first and last index relations from tiles to their first accessed stream elements and last accessed stream elements, respectively, for access \( c \), statement \( S \)

\( b^S_c \) burst for stream access \( c \), statement \( S \): function from task to the shift in the accessed stream’s read or write position at task creation for access \( c \), statement \( S \)
### Index of Notation

$b^{\otimes}_{c-d \times (n,m)}$ \text{ lump burst for each tile in } \text{ ran } P^{\otimes}_{c-d \times (n,m)} \text{ for access } d, \text{ statement } S$

$h^S_c$ \text{ horizon} for stream access $c$, statement $S$: function from task to size of the access window for access $c$, statement $S$

$h^{\otimes}_{c-d \times (n,m)}$ \text{ lump horizon for each tile in } \text{ ran } P^{\otimes}_{c-d \times (n,m)} \text{ for access } d, \text{ statement } S$

### Dependences

$S(a) \delta T(b)$ \text{ asserts iteration } T(b) \text{ depends on iteration } S(a)$

$P^{S \times T}_{c-d}$ \text{ dependence relation} for array references or stream accesses $c$ in statement $S$ and $d$ in statement $T$: a subset of all those tasks in the Cartesian product $D_S \times D_T$

$P^{S \times T|l}_{c-d}$ dependence relation for array references or stream accesses $c$ in statement $S$ and $d$ in statement $T$ at loop level $l$

$P^{S \otimes T}_{c-d}$ dependence relation for access $c$ in tiled statement $S$ and access $d$ in statement $T$

$P^{S\otimes \times T}_{c-d}$ dependence relation across pairs of tiles distancing $(n,m)$ in a bidimensional space for accesses $c$ and $d$ in statement $S$

$B^{S \times T}_{c-d}$ back-pressure dependence relation for accesses $c$ in statement $S$ and $d$ in statement $T$

### Schedules

$\theta$ \text{ schedule} for some program: function from iteration or task to a (possibly multidimensional) logical date

$\theta_S$ schedule for statement $S$: function from iteration or task in $S$ to a (possibly multidimensional) logical date

$\theta^i_S$ $i$-th component of the schedule for statement $S$

$\theta^{\text{orig}}$ schedule of the original program

$\theta^u, \theta^b$ schedule with unbounded and bounded streams

$\theta^{cs}, \theta^{nc}$ causal and strict non-causal schedule
INDEX OF NOTATION

δ_{ST,cd,l}  tentative schedule bound for Farkas’ lemma application associated with the dependence described by relation \( P_{c\rightarrow d}^{S\times T/l} \)

**Algebra and Analysis**

\([x], \lceil x \rceil\)  floor and ceiling of real number \(x\)

\(a[i..j]\)  subvector of vector \(a\) built from components \(i\) to \(j\)

\(a[i]\)  shorthand for \(a[i..i]\)

\(A^\top, A^{-1}, A^+\)  transpose, inverse and pseudoinverse of matrix \(A\)

\(A^+\)  orthogonal projector onto the null space of matrix \(A\)

\(\text{Card } U\)  cardinality of set \(U\)

\(\text{dom } Q\)  domain of binary relation \(Q\)

\(\text{ran } Q\)  range of binary relation \(Q\)

\(Q^+\)  transitive closure of relation \(Q\)
Personal Bibliography


Bibliography


