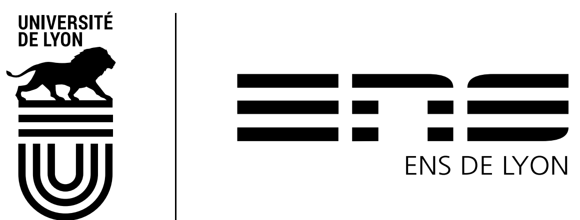


THÈSE
en vue de l'obtention du grade de Docteur, délivré par
l'ÉCOLE NORMALE SUPERIEURE DE LYON

École Doctorale N°512
École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 18/12/2024, par :
par :

Hugo THIEVENAZ

Scalable Trace-based Compile-Time Memory Allocation

Compilation d'allocation mémoire par analyse de trace avec passage à l'échelle

Devant le jury composé de :

ANCOURT, Corinne	Directrice de recherche Mines Paris & Université PSL	Rapporteure
CLAUSS, Philippe	Professeur Université de Strasbourg	Rapporteur
CHARLES, Henri-Pierre	Directeur de recherche CEA	Examinateur
JIMBOREAN, Alexandra	Ramon y Cajal Researcher Université de Murcia	Examinatrice
KETTERLIN, Alain	Maître de conférences Université de Strasbourg	Examinateur
TOUATI, Sid	Professeur Université Côte d'Azur	Examinateur
ALIAS, Christophe	Chargé de recherche HDR Inria Lyon	Directeur de thèse
KIMURA, Keiji	Professeur Université de Waseda	Co-Directeur de thèse

Contents

1 Introduction **7**
- 1.1 The Journey of Compilation . . . 7
- 1.2 Our Approach: the Polytrace Methodology . . . 8
- 1.3 Contributions . . . 9
- 1.4 Outline . . . 10

2 Background **13**
- 2.1 Polyhedral Model . . . 13
  - 2.1.1 Program Model . . . 14
  - 2.1.2 Dependences . . . 16
  - 2.1.3 Intermediate Representation . . . 18
  - 2.1.4 Scheduling . . . 19
  - 2.1.5 Tiling . . . 19
- 2.2 Array Contraction . . . 21
  - 2.2.1 Array Liveness Analysis . . . 23
  - 2.2.2 Successive Modulo . . . 25
- 2.3 Application to High-Level Synthesis . . . 26
  - 2.3.1 High-Level Synthesis (HLS) . . . 26
  - 2.3.2 Data-aware Process Networks (DPN) . . . 27

3 Related Work **33**
- 3.1 Polyhedral Memory Optimization . . . 33
  - 3.1.1 Memory Allocation . . . 33
  - 3.1.2 Array Liveness Analysis . . . 35
  - 3.1.3 Polyhedral Process Networks and Buffer Sizing . . . 36
- 3.2 Scaling the Polyhedral Model . . . 37
  - 3.2.1 Mitigating the Cost of ILP . . . 38
  - 3.2.2 Library-Level Improvements . . . 38
- 3.3 Trace Analysis and Speculation . . . 39
  - 3.3.1 Inference from Execution Traces . . . 39
  - 3.3.2 Speculative Optimizations . . . 39
- 3.4 Conclusion . . . 40

4 Canonical Array Contraction **41**
4.1 Overview . . . 42
4.2 Localizability . . . 42
4.3 θ-uniformity . . . 47
4.4 Trace-based Array Contraction . . . 51
4.4.1 Overview . . . 51
4.4.2 Trace Selection . . . 52
4.4.3 Fast Trace generation . . . 54
4.4.4 Trace buffer allocation . . . 57
4.5 Experimental Validation . . . 58
4.5.1 Setup . . . 58
4.5.2 Applicability . . . 59
4.5.3 Scalability . . . 60
4.6 Conclusion . . . 62

5 Linear Array Contraction **65**
5.1 Outline . . . 66
5.2 Program Model . . . 68
5.3 Correctness . . . 70
5.4 Liveness Extrapolation by Widening . . . 72
5.4.1 Parameter Selection . . . 72
5.4.2 Infer Polyhedral Constraints from Traces . . . 73
5.4.3 Widening Algorithm . . . 75
5.4.4 Narrowing . . . 76
5.5 Linear Allocation . . . 77
5.5.1 Correctness . . . 77
5.5.2 Iterating on the Next Dimension . . . 80
5.5.3 Efficiency . . . 80
5.5.4 Algorithm . . . 81
5.6 Experimental Validation . . . 83
5.6.1 Setup . . . 83
5.6.2 Liveness Analysis . . . 83
5.6.3 Linear Array Contraction . . . 84
5.6.4 Detailed Results . . . 87
5.7 Conclusion . . . 91

6 Conclusion **97**
6.1 Contributions . . . 97
6.2 Publications . . . 98
6.3 Perspectives . . . 99
6.3.1 An Improved Conflict Set Algorithm . . . 99
6.3.2 Global Array Space Optimization . . . 99
6.3.3 Other Compilation Optimizations . . . 99

A Résumé du travail de thèse **101**
A.1 Introduction 101
A.2 Contraction de tableau canonique 102
A.3 Contraction de tableau linéaire 102

Chapter 1

Introduction

High-Level Synthesis (HLS) [23, 11, 21, 7] consists in compiling a *circuit* from a high-level program. With HLS, there is no runtime, every scheduling and allocation decision from high-level task-grain parallelism to low-level operator pipeling must be taken at *compile-time*, which raises numerous challenges. A circuit might be seen as a *huge*, low-level, parallel synchronous program. Hence, precise *scalable* compilation techniques must be designed – in particular *large scale automatic parallelization* is required. Automatic parallelization techniques developped in the context high-performance computing (HPC), in particular those of the polyhedral model [33] meet the right level of precision and expressivity for this purpose. However, they strongly lack of scalability and cannot be used directly in HLS.

The broader goal of our research was to reduce the cost of doing those heavy analyses, by experimenting with a new approach that uses program information obtained at execution time. Our intuition was to exploit the ideas of the domain of dynamic optimizations, which trade their overhead for on-the-fly program transformations. Therefore, the problem we decided to tackle was defined with this in mind. What if this paradigm of execution-time optimization could be instrumented to empower compile-time optimizations, or even subsume them?

1.1 The Journey of Compilation

One starting point for automatic program optimization is the seminal paper of Prosser in 1959 [52]. In the 1960s, data-driven models were used to perform program optimizations. Karp et al. [38] proposed their *Uniform Recurrence Equation*, a foundational work on which the polyhedral model was built. Frances Allen et al. [8] laid the basis of Data-flow analysis by using the Control-flow Graph, an intermediate representation of a program which describes its possible paths of execution. Kildall et al. [40] then proposed the fixpoint method to perform global program analysis. These contributions have, at least in part, been the starting point for automatic compile-time optimizations. This was one of the first of many static analyses, methods that take as input the program's source code or a transformation of it.

By nature, compilers make the interface between *software* and *hardware* and must address the complexity of both. With the end of Dennard scaling, *hardware* has become more specialized and, as a consequence, more heterogeneous. In turn, *software* get more resource-hungry and complex.

Modern-day computation heavy applications are often algorithms processing large amounts of data, to quote a few: rendering engines, video processing tools, but also AI-related processes such as Deep Neural Networks or Large Language Models, who all digest very large data sets. Hence the need to develop *compilation models* specialized for these different domains of tasks. In particular, the *polyhedral model* was specifically designed for automatic parallelization and related program transformations on such compute-intensive HPC and embedded applications. It is one such model focusing on programs with heavy computations, which we want to *parallelize*, i.e. to partition a computation into smaller subcomputations to be run concurrently. Furthermore, *polyhedral HLS* [35, 5, 66, 45, 7] has also been a research subject where optimizations require powerful polyhedral static parallelization, for example to synthesize systolic networks [54].

There exists numerous *specialized compilers* whose their premise is less about which language to compile, but rather what end goal do they target. A general comprehensive compiler like GCC (which its history by itself demonstrates the point, from compiling *C* to supporting many languages and implementing many classical optimizations) cannot be put in the same category alongside verified compilers like CompCert, or High-Level Synthesis tools like VivadoHLS, or frontend frameworks like LLVM for large-scale compilation. Remark that, for example, the latter exploits the *polyhedral model* through Polly, which is a backend implementing several polyhedral optimizations such as automatic parallelization and vectorization.

Finally, the optimizations a compiler can apply have also naturally grown in complexity over time, becoming resource-hungry themselves. Array contraction [4], scheduling [31], tiling [17], are all program transformations of the *polyhedral model* that are usually realised at compile-time using costly geometrical operations, and *parametric integer linear programming* [29] known to be expensive and to cause *major scalability issues.*

Our work is centered around *improving the scalability of the polyhedral optimisations*, especially its biggest offenders cited before, geometrical operations and parametric ILP. On many suitable programs for this model, the optimizations are still very costly. The problem is therefore twofold: how to deal with the scalability of the analyses, while still retaining the necessary correctness?

1.2 Our Approach: the Polytrace Methodology

Our approach is to reproduce the results of an expensive polyhedral optimization by applying a *scalable, lightweight* analysis on a few execution traces and by *interpolating* the results, or at least by making a conservative *extrapolation.*

Polyhedral optimization deals with *Static Control Parts* [12], which basically consists of statements with affine array access functions nested inside regular loop nests. The static nature of the control entails that execution traces only depend on the *data size*, not on the data themselves. Also, polyhedral optimizations manipulate *affine* objects, typically polyhedra (for instance dependence analysis) and affine functions (for instance scheduling). Their affine nature make them *predictable from a finite number of informations*: an affine function f might be deduced from a finite number of points $(x, f(x))$.

For instance, if the program is *such that* the memory footprint is given by some affine mapping f of some program parameter N, we may exploit execution traces to *interpolate* f. From the

traces with $N = 3$ and $N = 4$, some *trace analysis* may deduce that $f(3) = 2$ and $f(4) = 3$. Then, we could *interpolate*, or *retro-engineer* a general expression of f, $f(N) = N-1$ working for any N. The hope is that *trace analysis* will be scalable, unlike pure static polyhedral computations. In the same way, polyhedra might be retro-engineered. For instance, if the set of points $P(1) = \{0\}$ can be produced from the trace obtained by taking $N = 1$ and $P(2) = \{0, 1\}$ from $N = 2$, we might interpolate the general domain $P(N) = \{i \mid 0 \leq i < N\}$. If the interpolation is not possible, we may want to *extrapolate* an overapproximation $P(N) \subseteq \{i \mid 0 \leq i\}$.

In general, precise assumption must be done on the program to ensure the *correctness* of such interpolations. For instance, the footprint might perfectly be *bounded* by some affine form $f(N) \leq a.N + b$ without being an affine form itself. Also, there might be several guesses from a trace. For instance, the polyhedron $P(N)$ might also be interpolated as $Q(N) = \{i \mid 0 \leq 2i \leq N\}$, which is *not* the same as $P(N)$. Hence, the main challenge of our approach is to delimit accurately the program model (assumption) and to show that, under these assumptions, the interpolation leads to a *correct* result.

To summarize, our stategy to deal with the scalability problem of polyhedral methods is based on *processing execution traces* of a program to apply *lightweight* trace-based analysis rather than expensive polyhedral computations. There are *two bets* made here. First, *this approach was faster* than the original polyhedral analysis process. Second, *there are minimum parameter values* for which a trace produced with those values or greater would ensure the approach is correct. The second is required, as we work on HLS, but it is also tied to the first, as parameter values directly impact trace size and therefore analysis runtime.

In this PhD thesis, we will focus on the *memory allocation problem for the purpose of HLS*, which is known to be of poor scalability. While this document focuses on the techniques we devised to realise memory allocation using our trace-based approach, we believe that our Polytrace methodology can be used for other polyhedral compiler optimizations.

1.3 Contributions

With our strategy explained, we can now present the contributions of this work. The following are two techniques to realise the *Array contraction* optimization, which consists in reducing the allocated memory of the arrays of a program as much as possible, as to minimize the program's memory requirement.

Canonical Array Contraction. This contribution's scope is *Data-aware Process Networks* (DPN) [7], an intermediate dataflow representation for HLS. This DPN form consists in partitioning the program into sets of processes that communicate through channels. These channels have to be allocated (and sized), as to minimize the memory consumption. The main problem is that the number of channels to size is *huge* (see Table 4.9). We show how we can replace a correct and accurate, but poorly scalable, buffer allocation algorithm with a scalable trace analysis that retains correctness and accuracy. Compared to state-of-the-art polyhedral techniques on *array contraction*, we present a new method for liveness analysis that operates on execution traces to build the *conflict sets*. We present a technique to select appropriate parameters for the trace execution, and a lightweight trace analysis. Finally, we prove that the underlying theory is correct,

and present the experimental validation of our technique which demonstrate its scalability.

Linear Array Contraction. In turn, this contribution is focused on more general programs, and yielding *linear*, or *affine*, *parametrized mappings*. Similar to our *canonical* method, we use a new method of building the *conflict sets* by realising liveness analysis on execution traces. This time however, we reconstruct parametrized conflict sets from the instances obtained by executing the program by *extrapolation*. We realise this with the following process. First, we present an instrumentalization of the NLR algorithm [39] that takes as input the *conflict information* resulting from the execution trace,and produces the corresponding polyhedral constraints. Next, we get rid of the constraints depending on program parameters using our *widening operator* ∇, which naturally induces over-approximation. It also makes the polyhedra that results from the constraints *open*, meaning the solutions to these constraints can be infinite. We therefore close those constraints by *narrowing* the associated polyhedra. Then, we present an adaptation of the SMO algorithm [14], by reformulating their *conflict set partitionning*, and associated heuristics i.e. *correctness* and *efficiency* constraints to operate on difference sets, which lowers the workload of the method compared to using conflict sets. Subsequently, we yield *linear contraction mappings* that solve the constraints of the difference sets. Finally, we present our experimental results that validate the scalability of our technique, by rephrasing liveness analysis to apply on execution traces, and by making the contraction operate on lighter objects (*difference* rather than *conflict sets*). Those results also show that the footprint overhead resulting from the over-approximation of the conflicts is negligible.

1.4 Outline

This manuscript will present the background notions, related work, and contributions regarding trace-based array contraction. It is structured as follows.

Chapter 2 describes the theorical background needed for the rest of this document. It will introduce the *polyhedral model*, the corresponding program representation and its associated notions. Then, we describe the *array contraction* optimization (memory allocation), which consists in allocating, sizing and potentially reducing the memory requirements of a program. Finally, we present the notions relevant to *High-Level Synthesis*, which is the context in which the programs considered in Chapter 4 evolve.

Chapter 3 describes the research related to our work, in terms of memory optimization techniques, scalability in the polyhedral model, and trace analysis and speculative execution.

Then, Chapters 4 and 5 present our contributions, which are two techniques for memory optimization that both use trace analysis.

Chapter 4 describes our *canonical array contraction* method, which infers mappings of constant sizes for the buffers of a *DPN* program. We show how to analyse the execution trace of a *DPN* program to infer the sizes of the buffers that will communicate data between channels.

Chapter 5 describes our *linear array contraction* method, which in turns infers mappings of parametrized sizes for the arrays of a program. We show how to apply liveness analysis on execution traces for successive parameter values, to deduce the program's patterns for any parameter.

Chapter 6 concludes this document with a summary of our contributions, and discusses several research directions spanning from our work.

Chapter 2

Background

This PhD thesis focuses on improving the scalability of memory allocation in the context of the polyhedral model for the purpose of High-Level Synthesis. This Chapter introduces the framework used to reason about the programs to optimize, and then presents the notions relevant to our work. In particular, Section 2.1 presents the *polyhedral model*, the general framework behind our work. Then, Section 2.2 presents *memory allocation* and discusses the underlying scalability issues. Finally, Section 2.3 presents High-Level Synthesis and outlines the Data-aware Process Networks, the intermediate representation for HLS used in Chapter 4.

2.1 Polyhedral Model

To facilitate both algorithmic and scientific reasoning, program *models* are used to represent the program's features as mathematical properties. They encompass the structure of the program with respect to certain attributes, which usually translates to limitations about the program's representation.

The *polyhedral model* [33] is an intermediate representation of a program as a graph over points of $\mathbb{Z}^n$. The class of programs that can be represented in this model, and therefore subject to polyhedral optimizations, are named Static Control Programs [12], that are comprised of (sequences of possibly nested) `for` loops where all loop bounds and conditions are affine functions of the surrounding loop iterators and program parameters. Polyhedral model make possibles to reason about programs at *iteration-level* and to derive powerful program optimisations.

A *polyhedral compiler* has typically the components outlined on Figure 2.1. A source program, written in a high-level programming langage – typically C – and fitting certain restrictions (Section 2.1.1) is abstracted away to the polyhedral intermediate representation (Section 2.1.3). In turn, the intermediate representation may need some restructuring depending on the program transformation we may want to apply (Section 2.1.5). Then, the polyhedral compiler reorganizes the computation (*Scheduling*, Section 2.1.4) and the data (*Allocation*, Section 2.2). Finally, a polyhedral code generator generates the final, optimized program. Usually, the final program is expressed in the same programming language than the source program. In that case, the polyhedral compiler is said to be a *source-to-source* compiler.

In this section, we will introduce the necessary notions for understanding the *polyhedral model*.

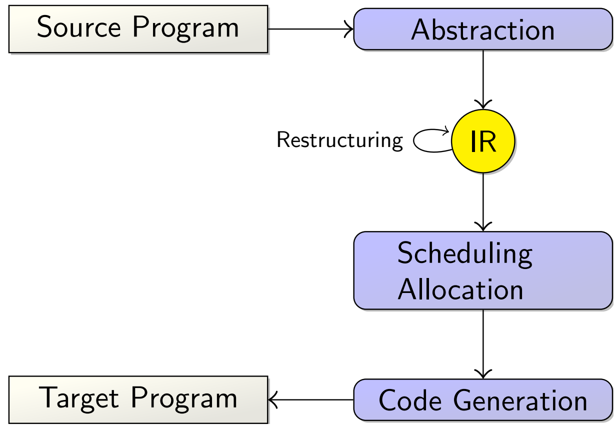


Figure 2.1: Typical polyhedral compilation flow

The next sections will present the compilation problem that we tackled in this context, the *array contraction* problem (Section 2.2) and the application domain, the compilation of circuits, usually refered to as *High-Level Synthesis* (Section 2.3).

2.1.1 Program Model

The polyhedral model focuses on static control programs, essentially *for* loop kernels manipulating arrays with affine indices:

Definition 2.1.1 (Static control program) *A program part is static control (or polyhedral) if and only if:*

- *It only contains assignment statements,* `for` *loops and conditionals.*
- *Loop bounds, conditions and array indices are affine functions of program parameters $\vec{N}$, and if nested, surrounding loop counters.*

Most of linear algebra kernels and signal processing applications fit in this category. Program parameters $\vec{N}$ are usually input and output array size. Typically, dimensions of matrices. The sequence of operations executed by a static control program (its *execution trace*) depends only on $\vec{N}$, *not on the input values.* Also, each operation is uniquely identified as an iteration instance of a statement S:

Definition 2.1.2 (Statement instance, iteration vector and domain) *Each execution of a statement S, nested in a n-depth loop, namely an instance or operation, can be represented by $\langle S, \vec{i}\rangle$ where $\vec{i}$ is a n-dimensional iteration vector of the surrounding loop indices. Its iteration domain D_S, the set of all possible values for the iteration vector of S, forms a graph over points of $\mathbb{Z}^n$.*

Motivating example Figure 2.2 depicts the Blur filter, the motivating example we will use throughout this thesis to illustrate our contributions. It is essentially a 2D producer/consumer with a phase shifting on the i loop, the producer iterations (writing `blurx`) being represented with grey points and the consuming iterations (reading `blurx`) being represented with black points. Red arrow depicts direct dependences held by the array `blurx`. Three arrays are operated on: the `in` array holds the numerical values describing an input image, `blurx` hosts the blurring intermediate computations, and `out` contains the resulting blurred image. More precisely, the computation has two phases.

- *Phase 1.* From $i = 0$ to $i = 1$, the writes into `blurx` will contain the blurring of the input array `in` alongside the vertical axis j.
- *Phase 2.* For $i \geq 2$, the vertical blurring of `in` continues. However, since the values of `blurx` from $i = 0$ to $i = 2$ get available, the blurring along i begins in parallel, by means of summing the values of `blurx` for $i - 2$ to i into output array `out`.

```
1 for(i=0; i<N; i++)
2   for(j=0; j<N; j++) {
3 P:   blurx[i][j] = in[i][j] +
4        in[i][j+1] + in[i][j+2];
5      if(i>=2)
6 C:     out[i][j] = blurx[i-2][j] +
7      blurx[i-1][j] + blurx[i][j];
8   }
```

(a) Kernel

j
4
3
2
1
0
0 1 2 3 4 i

(b) Polyhedral representation

Figure 2.2: Blur filter

The instances of the same statement $\langle C, \vec{i} \rangle$ are executed in the lexicographic order of the iterations $\vec{i}$: $\langle C, \vec{i} \rangle \prec \langle C, \vec{j} \rangle$ whenever $\vec{i} \ll \vec{j}$:

Definition 2.1.3 (Lexicographic order) *$\vec{u}$ is lexicographically less than $\vec{v}$, $\vec{u} \ll \vec{v}$ if there exists an index k such that:*

- *$\forall i < k,\ \vec{u}_i = \vec{v}_i$, and*
- *$\vec{u}_k < \vec{v}_k$.*

When considering instances of *different* statements $\langle S, \vec{i} \rangle$ and $\langle T, \vec{j} \rangle$, we need to restrict the comparison to the iterators of the common loops of S and T. We define $\text{crop}_{ST}(\vec{i})$ to be the parts of $\vec{i}$ pertaining to those common loops:

Definition 2.1.4 (sequential execution order) *$\langle S, \vec{i} \rangle$ is executed before $\langle T, \vec{j} \rangle$ in the sequential execution order, $\langle S, \vec{i} \rangle \prec \langle T, \vec{j} \rangle$, iff:*

- *$S = T$ and $\vec{i} \ll \vec{j}$*
- *$S \neq T$ and:*
  - *Either* $\text{crop}_{ST}(\vec{i}) \ll \text{crop}_{ST}(\vec{j})$
  - *Either* $\text{crop}_{ST}(\vec{i}) = \text{crop}_{ST}(\vec{j})$ *and S is before T in the textual order of the program*

Example (cont'd) Consider $\langle P, i, j \rangle$ and $\langle C, i', j' \rangle$. P and C share the loops i and j, hence $\text{crop}_{PC}(i, j) = (i, j)$. Then $\langle P, i, j \rangle \prec \langle C, i', j' \rangle$ iff $(i, j) \ll (i', j')$ or $(i, j) = (i', j')$, since P is before C in the *textual* order of the program. The full predicate translates to $i < i'$ or ($i = i'$ and $j < j'$) or ($i = i'$ and $j = j'$). Note that each disjunction corresponds to a *loop depth*. This will make possible to structure dependences.

2.1.2 Dependences

Polyhedral compilers may reorganize the computation to reach various goals. However, this restructuration is constrained by data dependences, that impose a minimal execution order to follow:

Definition 2.1.5 (Data Dependence) *There is a dependence from $\langle S, \vec{i} \rangle$ to $\langle T, \vec{j} \rangle$ iff:*

- *$\langle S, \vec{i} \rangle$ is executed before $\langle T, \vec{j} \rangle$ in the sequential order: $\langle S, \vec{i} \rangle \prec \langle T, \vec{j} \rangle$.*
- *Both operations access the same data (e.g. array cell):*
  - *a* write *to a* read *generates a* flow *dependence:* $\langle S, \vec{i} \rangle \rightarrow^{\text{FLOW}} \langle T, \vec{j} \rangle$
  - *a* read *to a* write *generates an* anti *dependence:* $\langle S, \vec{i} \rangle \rightarrow^{\text{ANTI}} \langle T, \vec{j} \rangle$
  - *a* write *to a* write *generates an* output *dependence:* $\langle S, \vec{i} \rangle \rightarrow^{\text{OUTPUT}} \langle T, \vec{j} \rangle$
  - *a* read *to a* read *generates an* input *dependence:* $\langle S, \vec{i} \rangle \rightarrow^{\text{INPUT}} \langle T, \vec{j} \rangle$

Usually, *input* dependences are ignored as they do not constrain the execution order. However they could be useful in the circuit synthesis context, when a certain input order is expected, for instance when the data are read from a FIFO. The whole dependence relation is usually denoted by $\rightarrow = \rightarrow^{\text{FLOW}} \cup \rightarrow^{\text{ANTI}} \cup \rightarrow^{\text{OUTPUT}}$. Note that *anti* and *output* dependences express resource conflicts, while *flow* dependences express the computation itself. *Anti* and *output* dependences might be removed by turning the program into *dynamic single assignment form* [61] (each array cell is written once at most): since there is only one write per array cell, no output dependence may remain. Also, an *anti* dependence $r \rightarrow^{\text{ANTI}} w$ implies the read r to written before by some operation w_0 – at least as an input – which would entails two writes w_0 and w of the same cell, and then contradicts the dynamic single assignment property.

The dependences might represented by a Polyhedral Reduced Dependence Graph (PRDG):

Definition 2.1.6 (PRDG) *A Polyhedral Reduced Dependence Graph (PRDG) is a graph whose nodes are program statements, and whose edges represent data dependences between statement instances.*

$$S \xrightarrow{\Delta_{ST}} T \iff \Delta_{ST} := \{(\vec{i},\vec{j}) \mid \langle S,\vec{i}\rangle \to \langle T,\vec{j}\rangle\} \neq \emptyset$$

Δ_{ST} *is a* dependence polyhedron. *It might be labeled by the type of dependence:*

$$\Delta^{\ell}_{ST} = \{(\vec{i},\vec{j}) \mid \langle S,\vec{i}\rangle \to^{\ell} \langle T,\vec{j}\rangle\} \qquad \textit{for } \ell \in \{\text{FLOW, ANTI, OUTPUT}\}$$

Example (cont'd) Note that the program is in dynamic single-assignment form: each cell is written once at most. Hence, there are only flow dependences. Figure 2.2.(b) depicts some flow dependences instances (red arrows) between instances of P and C. Let us write the conditions for having a dependence $\langle P,i,j\rangle \to^{\text{FLOW}} \langle C,i,j\rangle$:

- $\langle P,i,j\rangle$ is executed before $\langle C,i,j\rangle$: $(i,j) \ll (i',j')$ or $(i,j) = (i',j')$
- Both operations access the same array cell:
  - From the write `blurx`$[i][j]$ to the read `blurx`$[i'][j']$: $(i,j) = (i',j')$
  - From the write `blurx`$[i][j]$ to the read `blurx`$[i'-1][j']$: $(i,j) = (i'-1,j')$
  - From the write `blurx`$[i][j]$ to the read `blurx`$[i'-2][j']$: $(i,j) = (i'-2,j')$

This way:

$$\Delta_{PC} = \left\{ \begin{array}{r} (i,j,i',j') \mid (i,j) \in D_P \text{ and } (i',j') \in D_C \text{ and} \\ ((i,j) \ll (i',j') \text{ or } (i,j) = (i',j')) \text{ and} \\ \left(\begin{array}{l} (i,j) = (i',j') \text{ or} \\ (i,j) = (i'-1,j') \text{ or} \\ (i,j) = (i'-2,j') \end{array} \right) \end{array} \right\} \tag{2.1}$$

In turn, the lexicographic order is an exclusive disjunction $(i,j) \ll (i',j')$ iff $i < i'$ or ($i = i'$ and $j < j'$), each term of the disjunction corresponding to a loop *depth*. Usually, we turn Eq 2.1 to a disjunctive normal form and we consider each conjunction term as a separate dependence polyhedron − hence a separate edge of the PRDG. Indeed, conjunction of affine constraints are more suitable to perform polyhedral operations. We end up with *one dependence polyhedron per depth d and per read r*, $\Delta^{d,r}_{PC}$. For instance for the read 3 `blurx`$[i'-2][j']$ at depth 1, we have:

$$\Delta^{1,3}_{PC} = \{(i,j,i',j') \mid 0 \leq i < i' < N \text{ and } i = i'-2 \text{ and } j = j'\}$$

The final PRDG will then have 9 edges (3 ordering terms × 3 reads).

Direct dependences A dependence $w \to^{\text{FLOW}} r$ means that w is executed before r and that w write a cell, later accessed by r. However, we may want to retrieve the *last* write w_0 which produces the value read by r. The dependence $w_0 \to^{\text{FLOW}} r$ is called a *direct dependence*, or sometimes a producer/consumer dependence. On the Blur filter example 2.2, the dependence $\langle P,i-2,j\rangle \to^{\text{FLOW}} \langle C,i,j\rangle$ is a direct dependence relating the production of `blurx`$[i][j]$ to its consumption as `blurx`$[i'-2][j']$.

The computation of direct dependences is slightly more complex than the PRDG, as it requires to retrieve the *last* write w_0 executed before r [30]:

$$w_0 = \max_{\prec}\{w \mid w \rightarrow^{\text{FLOW}} r\}$$

On the blur filter example, for the target read `blurx`[i-2][j] of $\langle C, i, j\rangle$, we would have:

$$\begin{aligned} w_0(i,j) &= \max_{\prec}\{\langle P, i', j'\rangle \mid \langle P, i', j'\rangle \rightarrow^{\text{FLOW}} \langle C, i, j\rangle\} \\ &= \begin{cases} \langle P, i-2, j\rangle \text{ if } 0 \leq i-2, j < N \\ \perp \text{ otherwise} \end{cases} \end{aligned}$$

In general, a direct dependence can always be written as $h_C(\vec{i}) \rightarrow^{\text{FLOW}} \langle C, \vec{i}\rangle$, where h is a *piecewise affine mapping* called the *source function* or the *dependence function*. The restriction of h_C to the sources from statement P is denoted by h_{PC}. It is such that

$$h_{PC}(\vec{i}) = \vec{j} \iff h_C(\vec{i}) = \langle P, \vec{j}\rangle$$

Note that h_{PC} yields a *source iteration*.

Direct dependences are very useful in automatic parallelization, for instance to place synchronizations [9]. In High-Level Synthesis, they might be used as a *multiplexers* [62, 7].

2.1.3 Intermediate Representation

A common intermediate representation on polyhedral compilers is the *dynamic single assignment form* [30]:

Definition 2.1.7 (Dynamic single assignment form, DSA) *A program is in dynamic single assignment form if and only if each data location is written once at most during any program execution.*

Dynamic single assignement is more powerful than Static Single Assignment form (SSA) [24], which only require to write different data locations *in the program text*. With dynamic single assignment form, only direct dependences remain. In a way, it is a dataflow representation of the program which abstracts away data organization and keeps only the useful dataflow information about the computation.

As for SSA, the program is rephrased to ensure the DSA property: each operation $\langle T, \vec{i}\rangle$ will write an array $T[\vec{i}]$. As for SSA, the main difficulty is to rephrase the reads in terms of the new locations $T[\vec{i}]$. The solution proposed in [30] is to create for each read r its respective dependence function $h_{T,r}$, and substitute in $h_{T,r}(\vec{i})$ each $\langle S, u(\vec{i})\rangle$ by the data location $S[u(\vec{i})]$. In a way, dependence functions plays for DSA the same role as ϕ-functions for SSA: encoding the source statement of the read.

Example (cont'd) The Blur filter is already in DSA form. Using the algorithm [30], we would obtain the following code:

```
1
2 for(i=0; i<N; i++)
3   for(j=0; j<N; j++) {
4 P:  P[i][j] = in[i][j] + in[i][j+1] + in[i][j+2];
5     if(i>=2)
6 C:    C[i][j] = P[i-2][j] + P[i-1][j] + P[i][j];
7   }
```

Listing 2.1: blur filter C code

Once the program is in DSA form, a typical polyhedral compiler would reorganize the computations of a statement S using a scheduling function θ_S (Section 2.1.4), and the data using a data mapping function σ_S (Section 2.2), and that for all statements. This PhD thesis focuses on the computation of such data mapping functions.

2.1.4 Scheduling

In the polyhedral model, program transformations are specified using a scheduling function:

Definition 2.1.8 (Schedule) *A schedule defines an execution order by assigning to each operation $\langle S, \vec{i} \rangle$ a timestamp $\theta_S(\vec{i})$. Affine schedules rely on affine mappings of the form $\theta_S(\vec{i}) = A_S\vec{i} + B\vec{N} + \vec{c}$, where $\vec{N}$ are the program parameters. The timestamps are ordered with the lexicographic order $\ll$.*

A schedule θ induces an execution order $\prec_\theta$ such that $\langle S, \vec{i} \rangle \prec_\theta \langle T, \vec{j} \rangle$ if and only if $\theta_S(\vec{i}) \ll \theta_T(\vec{j})$. In a way, a schedule maps each operation to a common set of iterations (the set of timestamps), that are then executed according to the lexicographic ordering. When the schedule maps two operations at the same time, they are prescribed to be executed in *parallel*.

A schedule is correct if the dependences are satisfied: whenever $\langle S, \vec{i} \rangle \to \langle T, \vec{j} \rangle$, we must ensure that $\langle S, \vec{i} \rangle$ is executed before $\langle T, \vec{j} \rangle$: $\theta_S(\vec{i}) \ll \theta_T(\vec{j})$.

The affine shape of the schedule makes it possible to express many classical loop transformations (loop fusion, loop distribution, loop skewing). Trickier transformations may require to reorganize the program statements (loop unroll) or the iteration domain (loop tiling), as we will describe in the next section.

2.1.5 Tiling

A *loop tiling* is a partition of iteration domains into *atomic* tiles. Atomicity means that a tile might be abstracted in a function: once the tile inputs are available, the tile computation can be completed without requiring further synchronization. Loop tiling is often used in automatic parallelization to distribute a computation on parallel units, while tuning the granularity level by adjusting the tile size. In a tile, the dependence distance (number of iterations between the source and the target of a dependence) is bounded by the tile volume. Hence, loop tiling also tends to improve data locality.

Rectangular loop tiling associates, to each iteration $\vec{i} \in D_S$ of a statement S, a tile $\text{tile}_S(\vec{i}) = \vec{i}/\vec{b}$, where $\vec{b} \in (\mathbb{N} \setminus \{0\})^n$ is the tile size along each dimension (assuming D_S might be plunged into a space of dimension n), and / is a component-wise euclidian division. For each statement

S, the *tiled iteration domain* is $\hat{D}_S = \{(\vec{T}, \vec{i}) \mid \vec{i} \in D_S, \vec{T} = \text{tile}_S(\vec{i})\}$. Usually, $\vec{T}$ are referred to as *tile counters* or *tile coordinates*. We can then specify a schedule which prescribes an execution per tile, typically $\theta_S(\vec{T}, \vec{i}) = (\vec{T}, \varphi(\vec{i}))$ for any $(\vec{T}, \vec{i}) \in \hat{D}_S$. If $\vec{T} = (T_1, \ldots, T_n)$, the set of iterations obtained by setting the value of $T_1, \ldots, T_{n-1}$ is called a *tile band*.

Affine loop tiling [43, 44, 15] dives the iteration domain D_S of each statement S into a *common iteration space* through an affine mapping ϕ_S before applying rectangular loop tiling: $\text{tile}_S(\vec{i}) = \phi_S(\vec{i})/\vec{b}$. Affine loop tiling might be required when rectangular loop tiling does not satisfy the atomicity condition. Often, the components ϕ_S^k of ϕ_S are referred to as *tiling hyperplanes*.

A sufficient condition for *tiling correctness* is to enforce forward dependences between tiles: whenever $\langle S, \vec{i}\rangle \rightarrow \langle T, \vec{j}\rangle$, $\text{tile}_S(\vec{i}) \leq \text{tile}_T(\vec{j})$, where $\leq$ is componentwise – this rephrases to $\phi_S(\vec{i}) \leq \phi_T(\vec{j})$ whenever $\langle S, \vec{i}\rangle \rightarrow \langle T, \vec{j}\rangle$. This way, there will never be interdependence between tiles, which guarantees the atomicity.

Example (cont'd) The blur filter might be tiled with $\phi_S(i, j) = \phi_T(i, j) = (j, i)$. Figure 2.3 depicts the tiling obtained with tile size $b_1 = b_2 = 4$. For each dependence $s \rightarrow t$, $\phi(s) \leq \phi(t)$ is verified, hence the tiling is correct. Also, the dependence are all held by the last hyperplane (i). The first hyperplane (j) does not hold any dependence, hence it splits the iteration domain into fully independent parts – no communication nor synchronization are required. Thick line delimits tile bands. The first tile band contains the tiles $\vec{T} = (0, 0)$ and $(0, 1)$, the second tile band contains the tiles $\vec{T} = (1, 0)$ and $(1, 1)$.

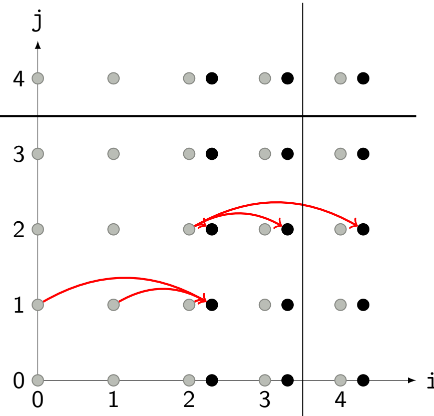


Figure 2.3: Blur filter, tiling

Full tiles A tile is *full* with respect to a statement S if all its vertices belong to the iteration domain of S, D_S.

Recall that a tiling is defined into the common iteration space targeted by each ϕ_S, as a *rectangular tiling*. On that common iteration space, the origin vertice of tile $\vec{T}$ is $\hat{M}_0(\vec{T}) = \vec{b} \times \vec{T}$, where $\times$ is the componentwise multiplication. Then, the remaining vertices $k = 1, \ldots, n$ are obtained as $\hat{M}_k(\vec{T}) = \hat{M}_0(\vec{T}) + (\vec{b} - \vec{1}) \times \vec{e}_k$ where $-$ is the componentwise subtraction, $\vec{1}$ is a vector of 1 and $\mathcal{E} = \{\vec{e}_1, \ldots, \vec{e}_n\}$ is the canonical basis in dimension n ($\vec{e}_1 = (1, 0, \ldots, 0)$,

$\vec{e}_2 = (0, 1, 0, \ldots, 0)$, and so on). The corresponding vertice in the original iteration domain of S is $M_k(\vec{T}) = \phi_S^{-1}(\hat{M}_k(\vec{T}))$. Finally the predicate to check whether or not a tile $\vec{T}$ is full is:

$$\text{fulltile}_S(\vec{T}) = \bigwedge_{k=0}^{n} M_k(\vec{T}) \in D_S$$

Usually, it is better to precompute the domain of full tiles w.r.t. a statement S:

$$\mathcal{F}_S = \{\vec{T} \mid \text{fulltile}_S(\vec{T}) \wedge \exists \vec{i}:\ (\vec{T}, \vec{i}) \in \hat{D}_S\}$$

2.2 Array Contraction

Once the computations specified in the polyhedral intermediate representation have been reorganized, the data must be *compiled*. This means, finding an *allocation* function σ_A for each array A such that each *virtual data location* (single assignment array cell) ℓ of the intermediate representation is mapped to an actual *physical data location* $\sigma(\ell)$. Many approaches exists, essentially classified into *inter-array* allocation and *intra-array* allocation. Inter-array allocation [14] maps any array cell $A[\vec{i}]$ to a location Global$[\sigma_A(\vec{i})]$ in a *common data space shared by all the data*, similarly to a scheduling function for the timestamps. Intra-array allocation maps each array cell $A[\vec{i}]$ into a dedicated private space $A_c[\sigma_A(\vec{i})]$. For instance, this applies to buffer allocation in the context of circuit synthesis, which is the main motivation of this PhD thesis. Hence, we will focus on *intra array allocation*. In the following, array allocation means intra array allocation.

Example (cont'd) On the Blur filter example, with the canonical schedule $\theta_P(i, j) = (i, j, 1)$ and $\theta_C(i, j) = (i, j, 2)$, a valid array allocation (also named *array contraction*) could be:

```
1 for(i=0; i<N; i++)
2   for(j=0; j<N; j++) {
3 P:  Pc[i%3][j] = in[i][j] + in[i][j+1] + in[i][j+2];
4     if(i>=2)
5 C:    Cc[i][j] = Pc[(i-2)%3][j] + Pc[(i-1)%3][j] + Pc[i%3][j];
6   }
```

This problem is analogous to register allocation in a compiler back-end. In particular, the allocation function depends on the *liveness of array cells*, which, in turn, depends on the scheduling function.

Array liveness On a program in DSA form, an array cell $A[\vec{i}]$ is defined once (by an operation $W_A(\vec{i})$) and is alive until its *last read* $R_A(\vec{i})$. In particular, there exists a *direct dependence* $W_A(\vec{i}) \rightarrow^{\text{FLOW}} R_A(\vec{i})$. The time interval $[\theta(W_A(\vec{i})), \theta(R_A(\vec{i}))[$ is said to be the *life interval* of $A[\vec{i}]$. As for register allocation, the last read is excluded to ensure that intervals $[t_w, t_r[$ and $[t'_w, t'_r[$ do not overlap when $t_r = t'_w$, since a write is executed after a read on a timestamp.

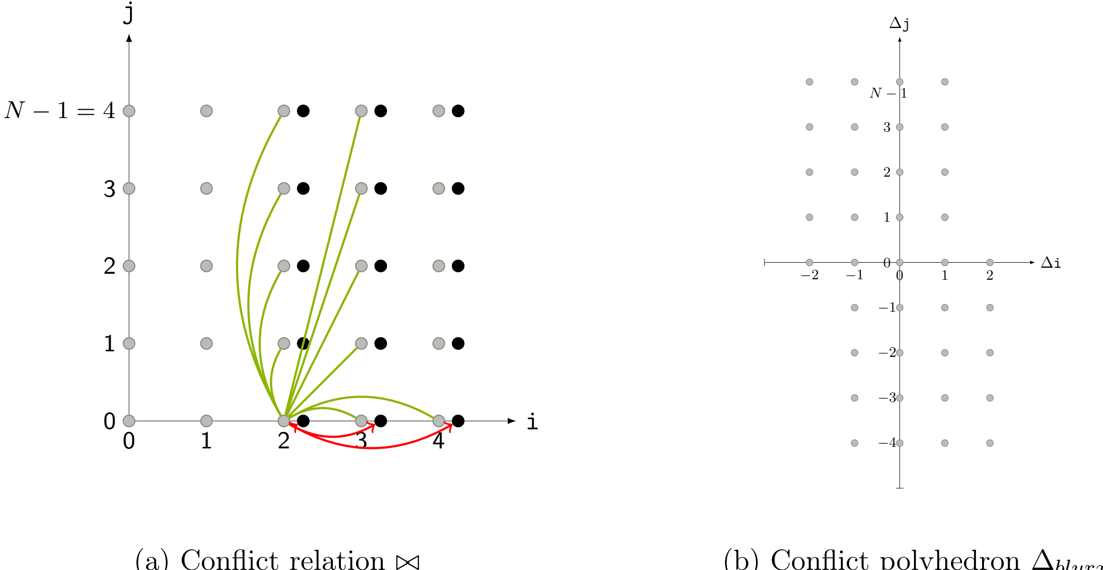
Definition 2.2.1 (Conflict relation) *Array cells $A[\vec{i}]$ and $A[\vec{j}]$ are conflicting, noted as $A[\vec{i}] \bowtie A[\vec{j}]$, if and only if their life intervals intersect. In other words, $A[\vec{i}] \bowtie A[\vec{j}]$ if they are alive on the same execution date.*

Note that conflict relations depend on the organization of life intervals, which in turn depends on the schedule θ. The conflict relation is symmetric and reflexive (by convention). The following is an alternative representation of conflicts, that expresses the *vectors* relating two conficting array cells.

Definition 2.2.2 (Difference set) *The difference set (or Δ-set) for A is:*

$$\Delta_A = \{\vec{i} - \vec{j} \mid A[\vec{i}] \bowtie A[\vec{j}]\}$$

Pursuing the analogy with scheduling, conflict relations are to dependence relations what difference sets are to dependence vectors. Since the conflict relation is symmetric, $\vec{i} - \vec{j} \in \Delta_A$ if and only if $-(\vec{i} - \vec{j}) = \vec{j} - \vec{i} \in \Delta_A$. Hence the difference set is 0-symmetric.



(a) Conflict relation $\bowtie$ (b) Conflict polyhedron Δ_{blurx}

Figure 2.4: Blur filter, liveness analysis

Example (cont'd) Figure 2.4 depicts the liveness of the array `blurx`. Red arrows depict direct dependences through `blurx`, while green edges illustrate the conflict relation. Note that the conflict relation links *written* array cells = (data), not iterations (computations). This representation of the conflict relation is possible only because the program satisfies the *dynamic single assignment* property. For sake of clarity, we do not draw all the conflicts edges. Note that extremal conflict edges also occurs starting from $\langle P, 2, 4\rangle$ to any $\langle P, i, j\rangle$, $3 \leq i \leq 4$ and $0 \leq j \leq 3$. (b) represents the conflict polyhedron, gathering all the vectors corresponding two conflicting `blurx` cells.

Correctness An array allocation σ for A is *correct* if and only if it satisfies the conflict relation:

$$A[\vec{i}] \bowtie A[\vec{j}],\ \vec{i} \neq \vec{j} \implies \sigma(\vec{i}) \neq \sigma(\vec{j}) \tag{2.2}$$

For intra-array allocation, the litterature focuses on linear mappings $\sigma(\vec{i}) = A\vec{i} \bmod \vec{b}$, where mod is the componentwise modulo function. This includes the particular case of canonical mappings $\sigma(\vec{i}) = \vec{i} \bmod \vec{b}$ (where A is the identity matrix). The mapping *footprint*, size of the compiled target array A_c, is simply the product of the modulo.

Example (cont'd) For `blurx`, a correct canonical mapping is $\sigma(i, j) = (i \bmod 3, j \bmod N)$, while a correct linear mapping may be $\sigma'(i, j) = -i + 2j \bmod 2N + 1$ – this will be detailed in Chapter 5. Note that σ has a *footprint* of $3N$ while σ' has a smaller footprint $2N + 1$. As a rule of thumb, linear mappings usually improve the footprint by a constant factor compared to canonical mappings (i.e., one dimension can be bounded by a constant rather than a parameter).

2.2.1 Array Liveness Analysis

We now discuss *array liveness analysis* on *dynamic single assignment programs*. This is a critical step, which *does not scale well*. The result of the analysis itself is a potentially *large* union of polyhedra, which hinders the scalability of the subsequent array contraction.

Two array cells $S[\vec{i}]$ and $S[\vec{j}]$ are conflicting if and only if their life intervals $[\theta(W_S(\vec{i})), \theta(R_S(\vec{i}))[$ and $[\theta(W_S(\vec{j})), \theta(R_S(\vec{j}))[$ overlap. This translates to the following ordering constraints:

$$(\theta(W_S(\vec{i})) \ll \theta(R_S(\vec{j}))) \wedge (\theta(W_S(\vec{j})) \ll \theta(R_S(\vec{i}))) \tag{2.3}$$

For each write $W_S(\vec{i})$, the last read $R_S(\vec{i})$ might be precomputed with the same algorithm as for computing direct dependences. However, when the program is tiled, this translates to a piecewise affine mapping with many pieces, as all the tiling corner cases must be considered. This incurs to a dramatically heavy overhead. Instead, we observe that $W_S(\vec{i}) \rightarrow^{\text{FLOW}} R_S(\vec{i})$ is a direct dependence. Hence, we enumerate all the direct dependence couples $\langle S, h_{ST}(\vec{i})\rangle \rightarrow^{\text{FLOW}} \langle T, \vec{i}\rangle$ and $\langle S, h_{SU}(\vec{j})\rangle \rightarrow^{\text{FLOW}} \langle U, \vec{j}\rangle$ through an array cell $A[\vec{i}]$ (resp. $A[\vec{j}]$) and we verify the constraints:

$$\theta_S(h_{ST}(\vec{i})) \ll \theta_U(\vec{j}) \text{ and } \theta_S(h_{SU}(\vec{j})) \ll \theta_T(\vec{i}) \tag{2.4}$$

This is summarized on Algorithm 1. Note that the lexicographic ordering in Eq 2.4 must be mutually distributed to obtain a Disjunctive Normal Form. In other words, all the combinations of depth for both $\ll$ must be considered. This is inefficient. If h is the number of dependence edges holding a value of S, and d is the maximum depth of a timestamp, then the number of emptyness testings (line 5) and the number of rational projections (line 7) is in $\mathcal{O}(h^2d^2)$. Usually, h is in $\mathcal{O}(R)$, with R the number of read locations of S in the program. With p the complexity of a rational projection and e the complexity of the emptiness testing, the overall complexity is in $\mathcal{O}(R^2d^2(e + p))$. On tiled programs, d^2, as well as e and p, tend to be higher since the dimensionality of the iteration domains double. *This causes major scalability issues when there are many arrays to allocate.* This typically happens in the HLS context where many buffers need to be allocated.

Example (cont'd) Consider the array P_c. We have 3 direct dependence pieces:

1. $h_{PC}(i, j) = \langle P, i - 2, j\rangle$ when $0 \leq i - 2, j < N, i < N$

Algorithm 1: ArrayLivenessAnalysisDSA

Data: Array S, Schedule θ, direct dependence function h
Result: Conflict relation $\bowtie$, Difference Set Δ_S

```
1  begin
2      foreach direct dependence piece h_ST do
3          foreach direct dependence piece h_SU do
4              ⋈_ℓ := { (i⃗, j⃗) | θ_S(h_ST(i⃗)) ≪ θ_U(j⃗), θ_S(h_SU(j⃗)) ≪ θ_T(i⃗),
                         i⃗ ∈ dom h_ST, j⃗ ∈ dom h_SU }
5              if ⋈_ℓ ≠ ∅ then
6                  ⋈ := ⋈ ∪ ⋈_ℓ
7                  Δ_S := Δ_S ∪ project({(δ⃗, i⃗, j⃗) | δ⃗ = i⃗ − j⃗, i⃗ ⋈ j⃗}, δ⃗)
8              end
9          end
10     end
11     return ⋈, Δ_S
12 end
```

Line 4: $\bowtie_\ell := \left\{ \begin{array}{c} (\vec{i},\vec{j}) \mid \theta_S(h_{ST}(\vec{i})) \ll \theta_U(\vec{j}), \theta_S(h_{SU}(\vec{j})) \ll \theta_T(\vec{i}), \\ \vec{i} \in \operatorname{dom} h_{ST}, \vec{j} \in \operatorname{dom} h_{SU} \end{array} \right\}$

Line 7: $\Delta_S := \Delta_S \cup \operatorname{project}(\{(\vec{\delta},\vec{i},\vec{j}) \mid \vec{\delta} = \vec{i} - \vec{j}, \vec{i} \bowtie \vec{j}\}, \vec{\delta})$

2. $h_{PC}(i,j) = \langle P, i-1, j\rangle$ when $0 \le i-1, j < N, i < N$
3. $h_{PC}(i,j) = \langle P, i, j\rangle$ when $0 \le i, j < N$

The scheduling function is $\theta_P(i,j) = (i,j,0)$ and $\theta_C(i,j) = (i,j,1)$. Then, we consider all the 9 pairs of pieces (h,h'). For instance, piece 1 and 2 leads to the following constraints:

$$\theta_P(i-2,j) \ll \theta_C(i',j'), \theta_P(i'-1,j') \ll \theta_C(i,j)$$

Which correspond to :

$$\left(\begin{array}{l} i-2 < i' \text{ or} \\ i-2 = i', j < j' \text{ or} \\ i-2 = i', j = j', 0 < 1 \end{array}\right), \left(\begin{array}{l} i-1 < i' \text{ or} \\ i-1 = i', j < j' \text{ or} \\ i-1 = i', j = j', 0 < 1 \end{array}\right)$$

Hence for piece 1 and 2, 9 pairs must be considered. For each combination, we apply the emptiness testing, and when it is non-empty, we compute the difference set with a projection on $\vec{\delta} = (i,j) - (i',j')$. All in all, we experimentally count 81 iterations and 39 non-empty cases leading to a projection. Note that for tiled programs, the schedule dimension is doubled which makes the complexity even worse.

A direct attempt of optimization would be to consider only half of the combinations, as (h,h') and (h',h) should lead to symmetric difference sets, and then compute the symmetric closure of the difference set. Experimentally, there is no improvement since the symmetric closure turns out to use up all the saved time.

2.2.2 Successive Modulo

We now describe the successive modulo algorithm [42] to compute canonical allocations $\sigma_a(\vec{i}) = \vec{i} \bmod \vec{b}$. If $\vec{\delta} \in \Delta_a \setminus \{\vec{0}\}$, then some $a[\vec{i}]$ and $a[\vec{i}+\vec{\delta}]$ are conflicting. Hence a correct allocation σ_a should satisfy: $\sigma_a(\vec{i}) \neq \sigma_a(\vec{i}+\vec{\delta})$:

$$\vec{i} \bmod \vec{b} \neq (\vec{i}+\vec{\delta}) \bmod \vec{b}$$

If a is monodimensional, this translates to: $\vec{i}_1 \bmod \vec{b}_1 \neq (\vec{i}_1 + \vec{\delta}_1) \bmod \vec{b}_1$, where $\vec{i}_1$ denotes the first coordinate of $\vec{i}$. A *sufficient condition* to enforce this constraint is $\forall\ \vec{\delta}_1 \in \Delta_a,\ \vec{b}_1 > \vec{\delta}_1$. The smallest one being $\vec{b}_1 = 1 + \max \Delta_a$.

If a is multidimensional, the $\vec{b}_i$ might be computed incrementally from $\vec{b}_1$ to $\vec{b}_n$. With the same reasoning, we have $\vec{b}_1 = 1 + \max\{\vec{\delta}_1 \mid \vec{\delta} \in \Delta_a\}$. This solves all the conflicts $a[\vec{i}] \bowtie a[\vec{j}]$ where $\vec{i}_1 \neq \vec{j}_1$, but lefts unsolved the conflicts with $\vec{i}_1 = \vec{j}_1$ (or $\vec{\delta}_1 = \vec{i}_1 - \vec{j}_1 = 0$). Hence, we focus on unsolved conflicts to compute the remaining $\vec{b}_k$: $\Delta_a := \Delta_a \cap \{\vec{\delta}_1 = 0\}$. This is summarized on Algorithm 2.

Algorithm 2: SUCCESSIVEMODULO

Data: Difference set Δ_a, Array dimension n
Result: Allocation $\sigma : \vec{i} \mapsto \vec{i} \bmod \vec{b}$

```
1 begin
2     for k := 1 to n do
3         b_k := 1 + max{δ_k | δ ∈ Δ_a}
4         Δ_a := Δ_a ∩ {δ | δ_k = 0}
5     end
6     return σ : i ↦ i mod b
7 end
```

Example (cont'd) On Δ_{blurx} depicted on 2.4 (b), we have $\vec{b}_1 = 1 + \max\{\delta i \mid (\delta i, \delta j) \in \Delta_{blurx}\} = 1 + 2 = 3$. We then focus on unsolved conflicts: $\Delta_a := \Delta_a \cap \{(\delta i, \delta j) \mid \delta i = 0\}$, which gives $\{(\delta i, \delta j) \mid \delta i = 0, -N < \delta j < N\}$. Then, $\vec{b}_2 = 1 + \max\{\delta j \mid \delta i = 0, -N < \delta j < N\} = N$. Finally, we get: $\sigma_{blurx}(i,j) = (i \bmod 3, j \bmod N)$.

Scalability In general, Δ_a is a union of convex polyhedra, $\Delta_a = \bigcup_{\ell=1}^{n} \Delta_{a,\ell}$. Hence, the maximum of line 3 is computed as $b_k(\vec{N}) = \max\{b_{k,1}(\vec{N}), \dots, b_{k,n}(\vec{N})\}$ where $b_{k,\ell}(\vec{N}) = \max\{\vec{\delta}_k \mid \vec{\delta} \in \Delta_{a,\ell}\}$. Since $\Delta_{a,\ell}$ is parametrized by the program parameters $\vec{N}$ and the modulo is expected to be an integer, each $b_{k,\ell}(\vec{N})$ is computed with *parametric* integer linear programming [29]. The result is a piece-wise affine mapping depending on program parameters $\vec{N}$:

$$b_{k,\ell}(\vec{N}) = \begin{cases} \vec{N} \in D^1_{k,\ell} : b^1_{k,\ell}(\vec{N}) \\ \dots \end{cases}$$

Then, the global maximum is obtained by combining the piecewise affine mappings $b_{k,\ell}$ in the same way as array dataflow analysis [30]: each n-uple of pieces from $b_{k,1}(\vec{N}) \dots b_{k,n}(\vec{N})$ is considered. Say that $D^{i_1}_{k,1} : b^{i_1}_{k,1}(\vec{N})$ from $b_{k,1}(\vec{N})$, ..., $D^{i_n}_{k,n} : b^{i_n}_{k,n}(\vec{N})$ from $b_{k,n}(\vec{N})$. For each feasible n-uplet $(D^{i_1}_{k,1} \cap \dots \cap D^{i_n}_{k,n} \neq \emptyset)$ we derive the $\max\{b^{i_1}_{k,1}(\vec{N}), \dots, b^{i_n}_{k,n}(\vec{N})\}$. This is very expensive and, with the potentially large number n of $\Delta_{a,\ell}$, leads to the non-scalability of the algorithm.

Relaxed successive modulo When all the parameters are bounded by integer constants $\vec{N}_k \in [\ell_k, u_k]$, the algorithm may be relaxed by adding the constraints $\vec{N}_k \in [\ell_k, u_k]$ to each $\Delta_{a,\ell}$, and computing an *integer* maximum (*non parametrized*) by using a pure ILP algorithm (*non-parametrized*), far more efficient than its parametrized version. This relaxed version is scalable, at the price of a non-parametrized result and a *restricted* range of parameters. This algorithm will be used as a *baseline* in the experiments of Chapter 4.

2.3 Application to High-Level Synthesis

This section outlines the challenges of High-Level Synthesis and the required background for this PhD thesis (Section 2.3.1). In particular, the *data-aware process networks*, which the HLS intermediate representation used in Chapter 4, is defined and discussed (Section 2.3.2).

2.3.1 High-Level Synthesis (HLS)

High-Level Synthesis (HLS) [23] is the process of compiling a circuit from a high-level description, usually a C program with pragmas to guide the synthesis choices. An HLS tool is then a *compiler*, with an intermediate circuit representation summarizing the components of the program, a *front-end* generating said intermediate representation from the source program and a *back-end* generating the circuit description (usually in the VHDL language) from the intermediate representation.

Intermediate representation Most commercial HLS tools use an intermediate representation close to those of a classical compiler. It generates a hierarchical control-flow graph, decorated with statement-level data-flow dependences [11, 21], which could be in SSA-form or one its gated variants [19].

Front-end With this level of information, and without restriction on the input program, the possible front-end level compiler analysis and optimizations are limited. Usually, the tools apply classical dragon-book [2] transformations (code hoisting, constant propagation, loop pipelining, etc) [23, 19] . Most interesting transformations are not possible – or strongly limited – because of data dependence over-approximation.

Back-end *In turn*, the back-end of HLS tools is quite matured and can generate finely optimized finite-state machines and data-path, low-level pipeline scheduling and resource allocation [22].

How to optimize the HLS process? These observations lead to two complementary approaches:

- *Source-to-source optimization for HLS* which tries to overcome the inherent limits of HLS front-end, typically by applying polyhedral optimizations [50, 5, 47].
- *Polyhedral HLS* that attempts to craft a HLS tool using the precepts of polyhedral model, with an appropriate intermediate representation, front- and back-end [48, 7].

What are the challenges? With HLS, *everything must be compiled.* We cannot rely on a runtime library. Even the caching system does not exists and the data transfers with the external memory must be scheduled at compile-time [5]. Speculative optimizations are possible [28], but still very restrictive. Hence, the goal is *correct-by-construction* program transformations.

The circuit must be *massively parallel.* Especially with FPGA synthesis, to exploit the full potential of the chip; we must compile a *massively parallel program.* This causes major *scalability issues*, as we will see in the next section. A pass which lasts tens of second on a simple program can in turn last minutes in the HLS context. This makes Polyhedral HLS a good target to stress and improve the scalability of polyhedral transformations. In this PhD thesis, we focus on *buffer allocation* for DPN using array contraction.

2.3.2 Data-aware Process Networks (DPN)

The intermediate representation must capture the computation and the data-flow at iteration level to enable polyhedral optimization. With *Data-aware Process Networks* (DPN) [7], the program is represented as a set of processes communicating through channels. DPN focuses on *tiled programs* and features data transfers with the external memory at tile level. DPN are a particular case of Regular Process Networks (RPN).

Regular Process Network

A *Regular Process Network* (RPN) [7] is a set of processes communicating through channels. The execution is locally sequential (a process executes sequentially) and globally dataflow, as the processes run in parallel and synchronize through channels. A RPN is obtained from a static control program by specifying:

- A partition $\mathcal{P} = \{P_1, \ldots, P_q\}$ of the computions into processes. Each P_i is a subset of operations to be achieved by the process i.
- A scheduling function θ_i per process i, specifying the *sequential execution order* of the process i. For each operation $o \in P_i$, $\theta_i(o)$ is the execution date of o. θ_i is expected to be injective (sequential order).
- A partition $\mathcal{C} = \{C_1, \ldots, C_r\}$ of the direct dependences into channels. Each direct dependence $w \rightarrow^{\text{FLOW}} r \in C_i$ will be *solved* by the channel i (write to channel i, read from channel i).

Execution The execution of a RPN is *locally sequential* (each process executes its operations sequentially) and *globally parallel*: the processes execute in parallel and synchronize through channels.

Correctness and efficiency Any partitioning $\mathcal{P}$ and $\mathcal{C}$ leads to a correct RPN [7]. The difficulty is to find a partitioning minimizing the overall latency, while fitting the memory limit of the FPGA.

DPN Partitioning

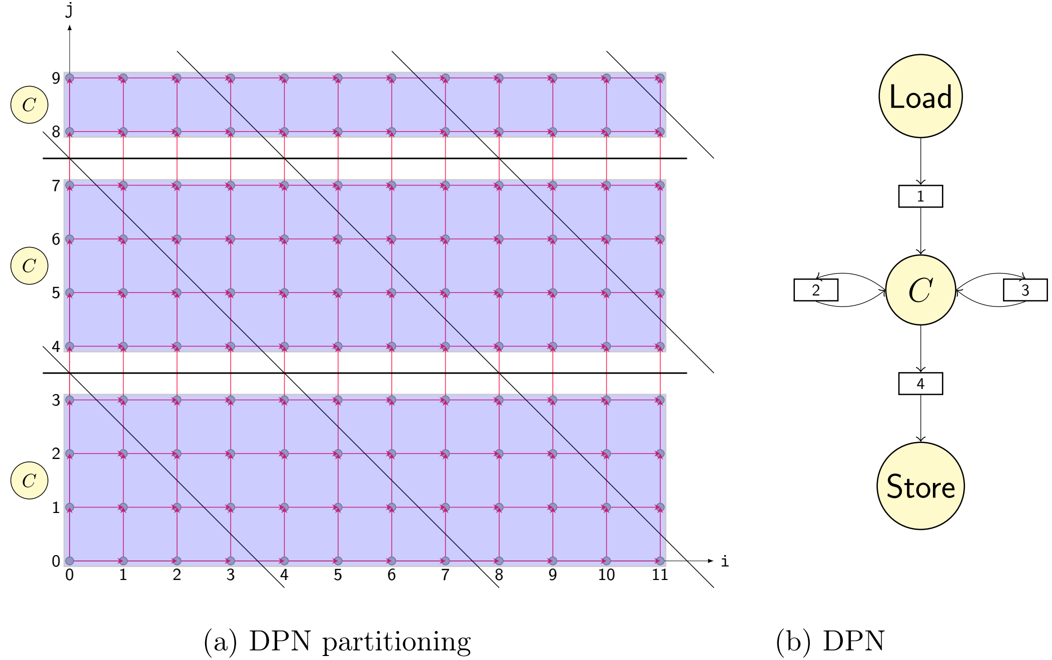
A DPN is an RPN with a partitioning *driven by a loop tiling.* The operations are partitioned by statement: a single process is produced per statement. Special processes $Load(a)$ and $Store(a)$ are produced for each array a and aim at communicating data with the external memory. The schedule assumes a tiled execution: for each tile, the required data is loaded by the tile, which is executed, then the data is stored.

With DPN partitioning, tile bands are considered as *reuse units*: *only* the data coming from a different tile band is loaded. Consequently, only the data going to a different tile band is stored. The data writes and reads are stored into a channel. In other words:

- Direct dependences inside a tile band (source and target inside the same tile band) are solved with a *local buffer*.
- Other dependences are solved through the *external memory*: when a direct dependence $s \rightarrow^{\text{FLOW}} t$ goes outside of the tile band of s, the latter sends the data to the Store process, which, in turn writes it to the external memory. t retrieves the data from a *Load* process, which will read it from the external memory.

Figure 2.5 depicts an example of DPN partitioning. We assume a loop tiling $\phi(i, j) = (j, i + j)$. The direct dependences are $(i - 1, j) \rightarrow^{\text{FLOW}} (i, j)$ and $(i, j - 1) \rightarrow^{\text{FLOW}} (i, j)$. The tile bands are depicted in blue. The dependence inside a tile are solved through channels 2 and 3, while direct dependences between tile bands are solved through the external memory. Assuming such a direct dependence $s \rightarrow^{\text{FLOW}} t$, s will send the data through channel 4 to the Store process, which will spill the data to the external memory. Then, when the target tile is about to be executed, the Load process will load that data from the external memory, and write it to channel 1. Note the two channels 2 and 3: in the DPN-partitioning, there is one channel per read. This is required to enforce a determinist execution [7].

Process scheduling Any correct *sequential* schedule θ prescribes a correct local execution order θ_P for each process P (assuming a process per statement). The schedule must be chosen to reduce the overall latency. In general, a process makes a computation thanks to a pipelined arithmetic operator. If two successive inputs i_1 and i_2 are dependent (i_2 is computed from i_1), the pipeline will *stall* until i_2 is available. This situation may be avoided by scheduling the program so the dependence distance between i_1 and i_2 is at least the operator latency. In particular, iterations between i_1 and i_2 must be *independent.* This is done by tiling the program so the last tiling hyperplane carries all the dependences: $\phi^n_S(\vec{i}) < \phi^n_T(\vec{j})$ whenever $\langle S, \vec{i}\rangle \rightarrow^{\text{FLOW}} \langle T, \vec{j}\rangle$.



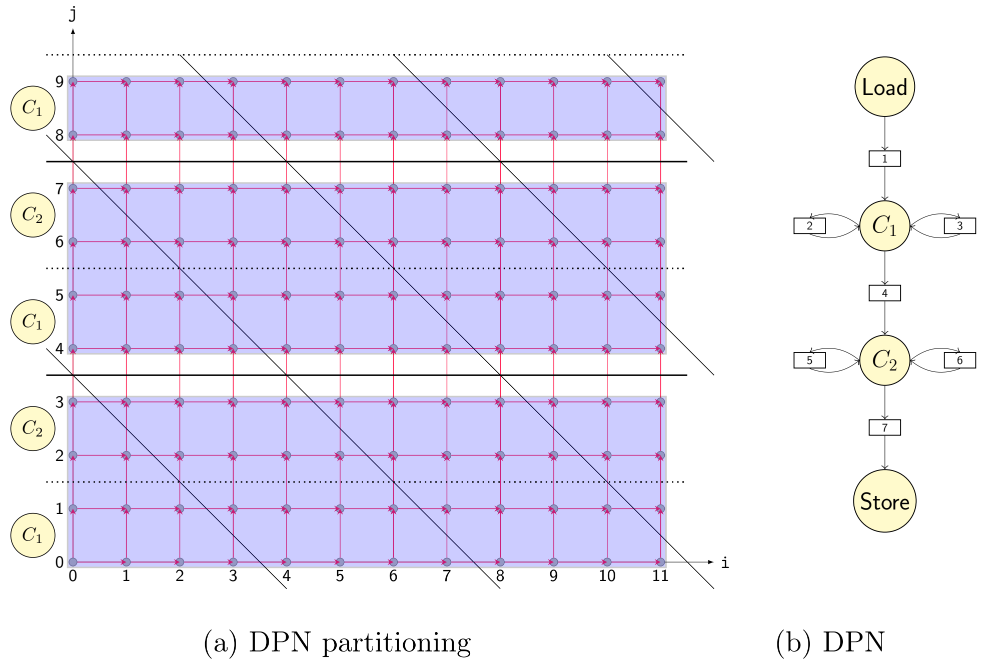
(a) DPN partitioning (b) DPN

Figure 2.5: Data-aware process network

Then, the operations of each process P are executed slice by slice (of ϕ_P^n) [6]. In other words, the first dimension of $\theta_P(\vec{i})$ is set to $\phi_P^n(\vec{i})$ for each process P, then a completion is applied to obtain a correct schedule θ_P. This schedule is called a *slicing schedule*. On Figure 2.5, $\theta(T_1, T_2, i, j) = (T_1, T_2, i + j, j)$ is a correct slicing schedule.

Parallelization factor With this partitioning, parallelism can be added to a DPN by subdividing the tile bands. This is illustrated in Figure 2.6. The *parallelization factor* is the number of sub-tile bands. To saturate FPGA resources, it is common to have a parallelization factor of several tens, typically 64. Note that this implies large tiles, except in the direction of the tile band.

Example (cont'd) Figure 2.7 depicts the DPN assuming the tiling illustrated in Figure 2.3. For each tile, the DPN loads the data (Load(In)), computes the tile (the two compute processes), and then stores the result (Store(Out)). The execution is dataflow: once a data is loaded, the Load process retrieves the data of the next tile, while the computation of the current tile is achieved. Note that the read of `blurx`[i,j] by $\langle C, i, j\rangle$ will be read as `blurx`[i-1,j] by $\langle C, i + 1, j\rangle$ and finally read as `blurx`[i-2,j] by $\langle C, i + 2, j\rangle$. Hence the data read as `blurx`[i,j] from buffer s_0 shall pass through a buffer s_1 to be subsequently read as `blurx`[i-1,j]. Once read as `blurx`[i-1,j], the same data is passed through a buffer s_2 to be read as `blurx`[i-2,j]. The same apparatus applies for the reads `in`[i,j+2], `in`[i,j+1], `in`[i,j]. This apparatus is called *systolization*, it is a feature of DPN. Figure 2.8 illustrates the outcome of a parallelization factor of 2 on that example.



(a) DPN partitioning (b) DPN

Figure 2.6: Data-aware process network: parallelization

Compiling a DPN

DPN compilation involves more than 12 steps, the main ones being:

1. Data-flow analysis and transformation to single-assignment form
2. Buffer partitionning and Load/Store generation
3. Parallelization
4. Buffer FIFOization
5. Buffer systolization
6. Buffer sizing

Step 1 computes the direct dependences and turns the program into a System of Affine Recurrence Equations, which is then post-processed to produce the DPN.

Step 2 adds Load/Store buffers according to DPN partitioning and refines the multiplexing. The set of data to be loaded/stored per tile is also computed following Alias et al.'s method [5].

Step 3 partitions the compute process and the buffers according to the parallelization partitioning. This usually multiplies the number of buffers by the parallelization factor.

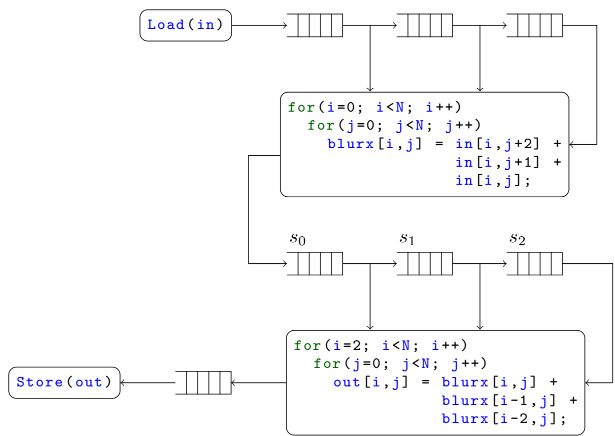


Figure 2.7: Blur filter, DPN

Step 4 restructures the buffer partitioning to enable FIFO communication pattern. This usually multiply by a factor 2 the numbre of buffers.

Step 5 systolizes the buffers as `blurx` in the example, when appropriate. This step generates additional buffers for the initialization of the pipeline.

Step 6 allocates and sizes all the buffers. For each buffer, a program summarizing the production and the consumption of the values is generated, and then fed to an array contraction algorithm. Due to the large number of buffers, *the compilation time is dominated by array contraction*, hence the need for scalable array contraction.

Example (cont'd) For systolization buffers, we have to consider the direct dependences depicted on Figure 2.9(a). For instance, $\langle C, i, j\rangle$ holds the read `blurx`$[i][j]$ and writes it to buffer s_1, so that $\langle C, i+1, j\rangle$ can read it as `blurx`$[i-1][j]$. Hence the direct dependence $\langle C, i-1, j\rangle \rightarrow^{\text{FLOW}} \langle C, i, j\rangle$ for any $2 \leq i-1, 0 \leq i, j < N$. Therefore, s_1 might be allocated by applying array contraction on a program summarizing the writes and the reads associated to that direct dependence. The liveness analysis for buffer s_1 is detailed, leading to the conflict polyhedron in (b). Array contraction leads to the canonical mapping $\sigma_{s_1}(i,j) = (i \bmod 2, j \bmod 4)$, or the smaller linear mapping $\sigma_{s_1}(i,j) = j \bmod 4$. Chapter 4 presents a trace-based method to find such canonical mappings. Then, Chapter 5 will present a trace-based method to find parametrized linear mappings.

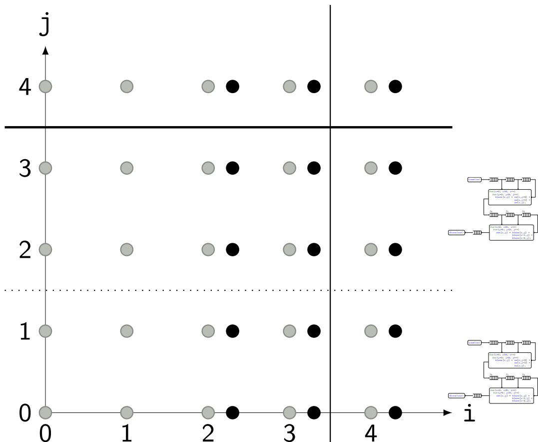
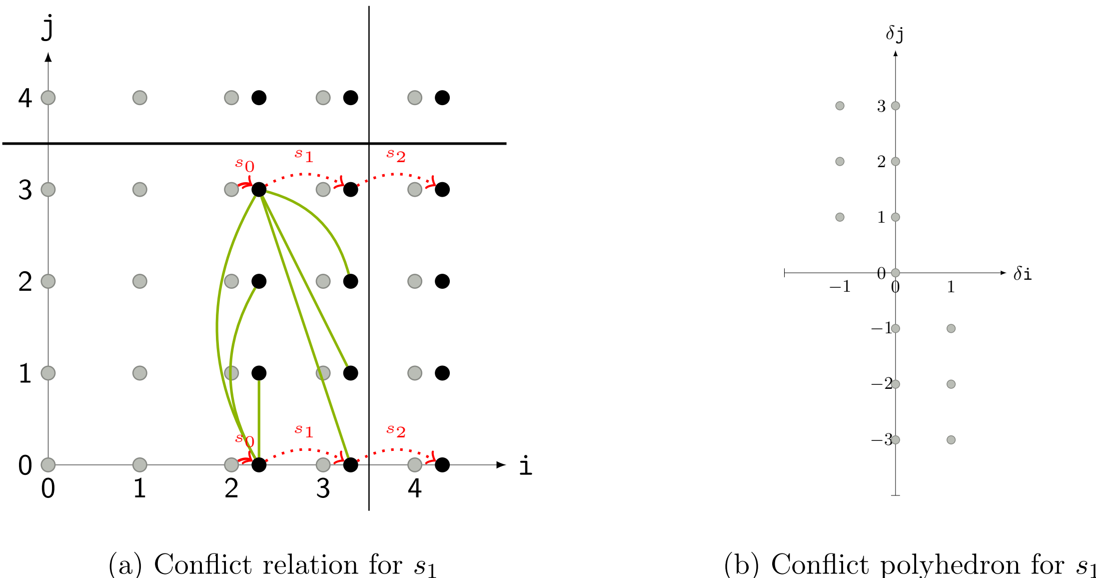


Figure 2.8: Blur filter, DPN with a parallelization factor of 2



(a) Conflict relation for s_1

(b) Conflict polyhedron for s_1

Figure 2.9: Blur filter, liveness analysis for systolization buffers

Chapter 3

Related Work

Our work on *scalable trace-based compile-time memory allocation* has three main characteristics. First, the premise of our work is to explore the use of *execution traces* of a polyhedral program. Second, this choice of trace analysis is motivated by *reducing the cost* of common operations of the polyhedral analysis domain, mainly parametric ILP. Third, our work has focused on the problem of *array contraction* for allocating communicating buffers in HLS. Hence, this related work section will be split in three parts, going in reverse order. To start, we will review memory optimizations in the polyhedral model (Section 3.1). Then, we will glance at work that seek to scale up polyhedral optimizations (Section 3.2). To conclude, we will present work that relates to trace analysis in polyhedral compilation (Section 3.3).

3.1 Polyhedral Memory Optimization

This section reviews state-of-the-art memory optimizations techniques in the polyhedral model. Section 3.1.1 reviews the approaches for allocating arrays. These approaches require array liveness analysis, reviewed in Section 3.1.2. Finally, Section 3.1.3 addresses process networks closed to DPN and their buffer sizing techniques.

3.1.1 Memory Allocation

The approaches may be divided in two categories. *Intra-array allocation* attempts to find a memory allocation *per array* $A[\vec{i}] \mapsto \hat{A}[\sigma_A(\vec{i})]$, while *inter-array allocation* maps arrays elements into a common global array space $A[\vec{i}] \mapsto \text{Global}[\sigma_A(\vec{i})]$.

Intra-array allocation In 1998, Lefebvre et al. [42] show that memory allocation during the automatic parallelization can be realised using Parametric ILP. It directly uses data dependency analysis and lifetime analysis of array elements to expose reuse across parallel tasks. The method proceeds as follows: transform a program using *data expansion* to turn it into DSA, then determine the cells' *utility spans* i.e. lifetimes, clearly exposing data reuse opportunities. Finally, contract the dynamic single assignment arrays with respect to a scheduling function. This paper introduces the *successive modulo* method presented in Section 2.2.2. As discussed, this algorithm is simple, but not scalable.

Quilleré et al. [53] tackle the memory allocation problem for ALPHA programs [41] by looking for projective allocation functions. Their approach selects a projection minimizing the dimension of the target array and is able to reach a proven tight bound on the target dimensionality. They show that it is enough for such mappings to be one-dimensional only, provided the dependencies are *uniform*. Compared to successive modulo, this approach might reduce the the footprint by a constant factor (as for affine mappings described later). However, the algorithm is rather complex and might not be easily rephrased with trace analysis, unlike successive modulo.

Darte et al. [26] rephrases the intra-array allocation problem as finding a *critical lattice* for the conflict polyhedron. The correctness condition 2.2 might be rephrased as $\vec{\delta} = \vec{i} - \vec{j} \in \Delta_a, \vec{\delta} \neq 0$ implies $\vec{\delta} \notin \ker \sigma$. When σ is a linear mapping $\vec{i} \mapsto A\vec{i} \bmod \vec{b}$, $\ker \sigma$ is a *lattice*, and $\sigma \mapsto \ker \sigma$ is a *bijection*: it is then sufficient to find the lattice $L = \ker \sigma$ to have σ. Hence the problem is to find a lattice whose intersection with Δ_a is reduced to $\{0\}$: $L \cap \Delta_a = \{0\}$, such a lattice is called an *admissible lattice* for Δ_a. Furthermore, $\det L$ is the *footprint* of σ: the smaller if $\det L$, the better is σ. Such a lattice is known as a *critical lattice* in number theory, and mathematical apparatus exist to find it, notably Roger's successive minima procedure [34] that Darte rephrases as an effective algorithm, together with two other gauge-based heuristics assuming the matrix A. In this paper, Δ_a is *not parametrized* and is assumed to be *convex*, which is not realistic, as a conflict polyhedron is generally a union of convex 0-symmetric polyhedra (*starred polyhedron*) whose width may involve program parameters. However these limitations are lifted in [25]. Unlike successive modulo, this approach is light and scalable. However, it still assumes the conflict polyhedron Δ_a, *whose computation is not scalable*. Alias et al. [4] used this idea to develop a complete intra-array contraction source-to-source transformation at compile-time, and proposes an array liveness analysis method to do so. Their paper already pointed out the lack of scalability of liveness analysis.

Bhaskaracharya et al. [13] present their *array space partitionning* approach. They propose a method to derive a linear mapping $\sigma : \vec{i} \mapsto A\vec{i} \bmod \vec{b}$ dimension per dimension as the successive modulo method. Each dimension is called a *storage hyperplane*, similarly to affine tiling. For each dimension, a storage hyperplane is found to *satisfy is much conflicts as possible* while *minimizing the footprint*. Farkas lemma is used to formulate the constraints. Then, unsolved conflicts are kept to compute the subsequent dimensions of σ. This is the same methodology than Feautrier's greedy scheduling algorithm [31], with the analogy conflict $\leftrightarrow$ dependence, allocation $\sigma \leftrightarrow$ schedule θ, footprint $\leftrightarrow$ latency. However, this is just an analogy. Unlike dependences, the conflict relation does not have a natural partition with the depth notion that allowed to find an optimal greedy algorithm [64]. So far, there is no *a priori* partitioning of conflicts into subsets whose *complete* resolution would lead to a provably optimal allocation. This is the main drawback of this approach. Also, the conflict relation tends to be huge, as discussed in Section 2.2.2 which may cause inefficiencies. In chapter 5, we will propose a similar rephrasing starting from the conflict polyhedron Δ_a instead of the conflict relation $\bowtie$ as they did, and we will show that the constraints are lighter.

Inter-array memory allocation De Greef et al. [27] presented a data placement technique, with the focus on embedded systems. This work considers the notion of local and global array space, showing how arrays can be made to fit into a single global array space that can then be allocated. They present a method to check if arrays are *compatible*, that is, no element of both

arrays are alive at the same time. This technique can be intuitively done by projecting the array space over the relative logical time dimension, and check for overlap. But it can also be done by analysing the *lifetimes* of the two arrays being compared, i.e., the time span between the last read of an array and the last write of another, or also, by checking if the arrays have periodical alternating lifetimes. A second method checks if two arrays can be *merged*, and a special case arises when one array A_1 reads from a second array A_2, and the cells of A_2 die and can be reused. This case is not uncommon and is a reuse opportunity. Finally, they present a greedy algorithm to piece mergeable array spaces together, by sorting the arrays and placing them in the global space, shifting it until there is no conflict left. They note that splitting the local arrays can further improve this memory reduction.

Bhaskaracharya et al. [14] extend their approach to *inter-array allocation* with a similar goal than De Greef's *compatible* and *mergeable* arrays paper [27]. However, the approach of De Greef only contracts along the original basis and therefore misses reduction opportunities. For each array cell $a[i]$, an allocation to a unified global array space is seeked $A[\vec{i}] \mapsto \text{Global}[\sigma_a(\vec{i})]$ with a mapping $\sigma_a : \vec{i} \mapsto A\vec{i} + \vec{b} \bmod \vec{c}$ *affine per array*. The same analogy might be drawn with Feautrier's greedy scheduling algorithm, now with *affine-per-statement schedules*. Actually, *intra-array allocation* is analogous to the *scheduling of perfect loop nests*, while *inter-array allocation* is analogous to *scheduling of non-perfect loop nests*. Inter-array allocation requires an *affine mapping*, which makes possible shifts into the unified global array space. Once an array a is completely allocated, additional dimensions may be required in the global space to allocate some other array b. If the global indices of a are not prefixing some global indices of b, then space is wasted for each additional dimension. This is mitigated by *decoalescing*: the algorithm is applied to subsets of arrays, separately. Then, the results are merged into the global array space. However, as for the intra-array algorithm, the algorithm is not optimal because of the conflict relation structure. Also, the linear constraints complexity are dictated by the complexity of the conflict relation $\bowtie$, which again, may be huge. Furthermore, the decoalescing is not provably optimal.

3.1.2 Array Liveness Analysis

As discussed in Section 2.2.1, liveness analysis for arrays is a critical step for building scalable memory allocation algorithms. However, most papers simply assume the conflict relation to be given and do not explain how to compute it, or do not compute the liveness explicitely and rely on partial liveness informations (which is not sufficient for our purposes) such as De Greef's *binary occupied address time domain* (BOATD) [27] or Strout's *universal occupancy vectors* (UOV) [58] and its variants [59, 65]. As far as we know, the first in-depth discussion of *array liveness analysis* was done in the PhD thesis of Isoard [36] which presents several techniques to compute the conflict relation from a static control program (SCoP). Unlike the technique presented in Section 2.2.1, the program is not expected to be in dynamic single assignment form (DSA). Each of these techniques, namely the *butterfly*, *cross-product* and *triangle* are depicted on Figure 3.1 and discussed below. The pictures of liveness patterns for *butterfly*, *cross-product* and *triangle* are borrowed from [36].

The *butterfly* technique considers pairs $W_x \rightarrow^{\text{FLOW}} R_x$, $W_y \rightarrow^{\text{FLOW}} R_y$ accessing distinct array elements $A[x]$ and $A[y]$ so that $\theta(W_x) \ll \theta(R_y)$ and $\theta(W_y) \ll \theta(R_x)$. For each pair, the

Liveness analysis	Direct (Section 2.2.1)	Butterfly	Cross-product	Triangle
Strategy	$h(R_x)$, R_x, $h(R_y)$, R_y	W_x, R_x, W_y, R_y	W_x, R_x, t, W_y, R_y	W_x, R_x, t, W_y
Program model	SCoP & DSA	SCoP	SCoP	SCoP
Complexity	$\mathcal{O}(R^2d^2)$	$\mathcal{O}(W^2R^2d^4)$	$\mathcal{O}(W^2R^2d^4)$	$\mathcal{O}(W^2Rd^2)$
Scalable	No	No	No	No

Figure 3.1: Array liveness analysis approaches and their complexity (worst case number of polyhedral emptiness testings and projections).

array cells $A[x]$ and $A[y]$ are conflicting. Hence, the conflict relation is obtained by projecting the constraints on (x, y). The complexity is not analyzed, but we can estimate the number of projections and emptiness testings as $\mathcal{O}(W^2R^2d^4)$ where W (resp. R) is the number of writes (resp. reads) to the array in *the program text* and d the maximum loop depth. Indeed, for each quadruplet (W_x, R_x, W_y, R_y) (term W^2R^2), we need to express 4 order relations: $\theta(W_x) \ll \theta(R_x)$, $\theta(W_y) \ll \theta(R_y)$, $\theta(W_x) \ll \theta(R_y)$, $\theta(W_y) \ll \theta(R_x)$, hence $\mathcal{O}(d^4)$ pieces to examinate (emptiness testing) and to exploit (projection to retrieve the conflict relation). Note that the term d tends to be larger in our case since the loop depth doubles on tiled programs.

Also, they propose a *cross-product* method, which computes the set Live(t) of array cells alive at that timestamp t as an affine relation $t \mapsto \text{Live}(t)$ and finally the conflict relation as a composition $\bowtie := \text{Live} \circ \text{Live}^{-1}$. As mentioned in their discussion, that composition incurs the same cross product than the butterfly method with the *same complexity*. Indeed, Live(t) involves ordering constraints $\theta(W_x) \ll t \ll \theta(R_x)$ with $\mathcal{O}(WRd^2)$ pieces. The composition $\text{Live} \circ \text{Live}^{-1}$ will process (emptiness testing and projection) the cross product of these pieces, hence $\mathcal{O}(W^2R^2d^4)$ pieces at worst.

Finally, their *triangle* method mitigates this problem and considers only a write executed in some liveness interval ($W_x \to^{\text{FLOW}} R_x$ and $\theta(W_y) \in [\theta(W_x), \theta(R_x)[$). Again, we can estimate the complexity as $\mathcal{O}(W^2Rd^2)$ projections and emptiness testings. Indeed, for each triplet (W_x, R_x, W_y), we just need to express the ordering constraints $\theta(W_x) \ll \theta(W_y)$ and $\theta(W_y) \ll \theta(R_x)$ (hence the term d^2).

All these techniques have a *worst* complexity than than the method presented in Section 2.2.1 and which is *experimentally demonstrated to be unscalable* in Chapter 4, Table 4.8. In the next two chapters, we show how to use lightweight trace analysis to produce a *scalable* array liveness analysis.

3.1.3 Polyhedral Process Networks and Buffer Sizing

The closest work to DPN are the Polyhedral Process Networks (PPN) introduced by Rijpkema [55] as a dataflow model of computation for programming heterogeneous multiprocessor platforms. PPN improves on *Kahn Process Networks* [37], as their process networks have bounded memory sizes, and support more storage types than FIFOs. PPN might be viewed as a particular

case of RPN, where each program statement is mapped to a process and with a single communicating buffer per couple (producer, read). In his PhD thesis, Turjan [60] proposed a complete compilation chain to derive automatically a PPN from a program specification. In particular, he addressed buffer sizing using a bounding box technique: the buffer size is exactly the volume of data read. This does not exploit data reuse and this leads to oversized buffers.

Clauss et al. [20] proposed a method to estimate the memory requirements, later used by Verdoolaege [62] to size FIFOs in the PPN context. They build a polynomial $L(t)$ which estimates the memory used at each execution point t of the program. Then, the maximization $M = \max_t L(t)$ provides an upper bound on the memory used. Since they consider FIFO communication patterns, each write is read once. Hence $L(t) = W(t) - R(t)$ where $W(t)$ is the number of writes before t and $R(t)$ is the number of reads before t. In turn, $W(t) = \text{card}\{\vec{i} \in D_W \mid \theta(\vec{i}) \ll t\}$ and $R(t) = \text{card}\{\vec{i} \in D_R \mid \theta(\vec{i}) \ll t\}$ where D_W is the iteration domain of writes, D_R those of reads and θ is a global schedule. Since $W(t)$ and $R(t)$ are counting polynomials, so is $L(t)$. Finally, the maximum M is obtained from the Bernstein expansion of L. However, the approach is inherently limited to dynamic single assignment programs whose arrays cells are read once, which limits its applicability for buffer sizing. Extending this technique to general access patterns would require to consider the *last read* for each write, whose computation might be expensive for tiled programs.

Later, Verdoolaege [62] built on the technique developped with Clauss et al [20] to size FIFO buffers of PPNs. Also, they proposed a way to size non-FIFO buffers without computing a last read. However, their formulation relies on multiple ordering constraints, which leads to a large union of convex polyhedra as the cross product must be computed. The obtained set of polyhedra is then subtracted from the consumer iteration, which is even worse in terms of complexity. Also, this work is only concerned with buffer sizing, it does not compute an allocation function.

All these approaches size a buffer by assuming its producer and consumer processes to follow some global affine schedule – the original program sequential schedule in [60], an affine schedule closed to the dataflow execution in [62]. Although correct – at worst, the process network will follow that schedule – it might causes buffer oversizing. Turjan [60] proposed preliminary ideas to estimate the memory requirements of a PPN by considering the global *dataflow execution* of the processes. Turjan derives a bound ℓ so that if the PPN executes without deadlock, then the total buffer size is $\geq \ell$. The bound is not proven to be tight, it is only a *necessary condition*: the minimal correct size could perfectly be bigger than ℓ. Also the PPN is assumed to be acyclic, which is not realistic. For instance, all the PPNs obtained from the Polybenchs are cyclic.

This concludes our overview of memory optimization techniques in the polyhedral model. We will now take a look at papers that worked on scaling different aspects of the polyhedral model, and while the works listed above naturally sought to improve the memory reduction, analysis runtimes and applicability of their techniques, the following papers were directly driven with the idea to polish parts of the overall polyhedral computation.

3.2 Scaling the Polyhedral Model

Polyhedral compilation often relies on parametric ILP and geometric operations over polyhedra, whose complexity is often exponential in the number of variables and constraints. Unfortunately,

this leads to non-scalable analysis, which may even freeze on complex programs. Hence, the research on speeding up polyhedral optimizations has mostly revolved around alleviating this pressure. Section 3.2.1 reviews state-of-the-arts approaches for mitigating the cost of specific polyhedral compilation steps, while Section 3.2.2 presents the most recent library-level approach.

3.2.1 Mitigating the Cost of ILP

In 2006, Feautrier [32] sought to speed-up automatic parallelization by improving the construction of an affine schedule. The paper presents a dataflow model of computation, the *communicating regular processes* (CRP) and presents a scalable modular scheduling algorithm to derive a global schedule for a CRP. A CRP is a set of processes communicating through infinite arrays. Each process executes a static control program in DSA form. In particular, each channel is single assignment, and each channel cell has a unique date of production given by a *channel schedule*. This structure suggests a modular scheduling algorithm. Each process is scheduled independently. The space of valid schedules Θ_P for a process P involves the coefficients $\mathcal{C}_P$ of its input and output channel schedules. Each Θ_P is projected out on $\mathcal{C}_P$, the constraints are concatenated and a global channel schedule is found. In turn, the values obtained for each $\mathcal{C}_P$ are injected into Θ_P and a local schedule is found for each process P. Note that each scheduling step involves a smaller set of constraints/variables than the global greedy affine scheduling algorithm [31]. This approach allows to effectively scale the scheduling algorithm and might be applied beyond the CRP context providing a decomposition into independent sub-problems. In general, divide-and-conquer approaches are worth to investigate for scaling polyhedral compilation.

A different approach was taken by Acharya et al. [1], who sought to get rid of ILP in the Pluto algorithm [15] by relaxing the ILP as a rational LP problem, and scaling the result to obtain integer solutions. Then, they scale the coefficients up to integrals using Mixed Integer Programming, but argue that other polynomial time heuristics could be used so their algorithm does not use ILP. However, for the Polybench kernels, the speed-ups are not noticeable, or sometimes create additional overhead.

3.2.2 Library-Level Improvements

Pitchanathan et al. [49] propose the *fast polyhedral library* (FPL), optimizing the usual operations on Presburger sets required in polyhedral compilers (subtraction, emptiness testing, coalesce, etc) by tuning the integer representation (quoted as integer precision). The operations are carefully implemented, avoiding pointer indirections and limiting memory size. The authors claims that their implementation supercedes the state-of-the-art ISL in performance and ease of use, and the performance improvement (notably by number of memory allocations) is due to using more suitable precision for the data and by exploiting vector instructions. However, FPL does not provide *parametric* integer linear programming (for instance a lexicographic maximum), which is mandatory in most polyhedral analysis. In particular, the *successive modulo* algorithm presented in Section 3.1.1 and used as baseline throughout this PhD thesis cannot be realised. Instead, the current version of FPL proposes ILP and LP algorithms (not discussed in the paper). The gain compared to state-of-the-art libraries like GLPK [46] remains to be established.

This concludes our quick overview of work on the scaling of polyhedral methods. In the last

part, we will be looking at inference in a wide sense, that is, reconstruction of program properties from traces, but also prediction of beneficial program transformations.

3.3 Trace Analysis and Speculation

The vast majority of work in the polyhedral model focuses on static optimizations, as Static Control Parts are meant to be easily analysable at compile-time, and the information required for optimizing these affine loop nests are stored in the Intermediate Representation of the program. This is why dynamic approaches, including trace-based approaches, are much less common to our knowledge. Section 3.3 presents analysis to retrieve a polyhedral representation from an execution trace – this is a fundamental building block in our approach. Then, Section 3.3.2 reviews state-of-art dynamic analysis in the polyhedral model.

3.3.1 Inference from Execution Traces

The *Nested Loop Recognition* (NLR) algorithm [39] by Ketterlin and Clauss is able to retrieve, from a trace of memory accesses, an equivalent affine loop nest. The goals are to predict the possible data accesses by simply extending the domains of the iteration vectors, and as a byproduct, to compress the input as a loop nest representation. The algorithm takes as input a sequence of (multidimensional) scalar variables or memory addresses, and output loop nests that reproduce the input. It is a *greedy algorithm* that stacks the terms of the sequence, checking if a triplet of values can be grouped into a loop, or if the new value can already belong to an existing loop created earlier in the process. The algorithm has a time complexity linear in the execution trace and demonstrates to be very efficient in practice. This work is important to ours, as we directly use their method to reconstruct a conflict polyhedron from an execution trace in Chapter 5.

Similarly, the work of Rodriguez et al. [56] is focused on recognizing loop nests from memory address traces. For each trace entry, either it is recognized as a *next iteration*, either a *new dimension* (loop) is added, either it is not recognized at all. Hence, many cases must be consider. For instance, the next iteration of (i, j) might be $(i, j + 1)$ or $(i + 1, 0)$. Also, the new loop k could be inserted in several positions: (k, i, j), (i, k, j) or (i, j, k). Their algorithm explores that solution space, guaranteeing to obtain the minimal solution should it exist. However, the complexity increases exponentially with the number of irregularities, dispite the pruning techniques proposed by the authors. This makes this work less useful to us, as our primary focus is to reduce the runtime of the array contraction optimization.

3.3.2 Speculative Optimizations

Apollo [18] is a polyhedral optimizer which, from an irregular code, speculates an equivalent affine loop nest and applies a polyhedral optimization. The code is divided into *slices* (e.g. the iterations of the first enclosing loop). On the first slice, the first iterations are collected and passed to NLR to predict an affine loop nest (e.g. the enclosed loop nest). In turn, that loop nest is optimized thanks to a state-the-art polyhedral optimizer (e.g. Pluto [16]) and the optimized code is efficiently obtained by instanciating *code bones*, which also check the correctness of the speculation. The same speculated loop nest is used for the next slices. If it no longer works,

a new affine loop nest is speculated. If the speculation is wrong, the system backtracks to the original code, executed as a background task at program start-up. This system is able to mine hidden regular computations in non-static control programs without complex static analysis. However, the speculation verification and the parallel execution of the original code causes an overhead. In particular, the memory should be allocated twice: for the original code and for the optimized code. That speculation scheme is very useful for *software* optimization. However, this would not apply for our purpose as we target *hardware* memory optimization. Indeed, once a memory is synthesized, it cannot be "deallocated" if the speculation was wrong. Instead, we exploit execution traces to produce a *correct by construction* memory analysis.

In the same vein, PolyJIT [57] is a polyhedral loop optimizer built on top of LLVM. Being *Just-In-Time*, it applies its transformations during execution, i.e. at runtime. It works by detecting *hot* regions, that is, compute-intensive program parts. These are the parts for which, if polyhedral optimizations can be applied, the potential speedup outweight the dynamic overhead. It is important to note that these *hot* regions are profiled at compile-time, and then the transformations are applied at runtime. This is the exact opposite of our approach: we use runtime informations collected from very small execution traces to infer compile-time optimizations.

3.4 Conclusion

In this chapter, we reviewed state-of-the-art approaches for memory optimization in the polyhedral domain, array liveness analysis and application to HLS context. We show that all the techniques developed so far *do not scale*. Either because they rely on non-scalable liveness analysis, either because they are themselves non-scalable. Also, we review attempts to scale polyhedral compilation – none of them are specifically concerned with array contraction. In particular, divide-and-conquer techniques seem to give effective results when the problem is modularizable, which is not the case for array contraction. Finally, we review state-of-art trace-based techniques in the polyhedral model. So far, traces are used for *speculation*. There is no need to have a correct result, it will be checked at runtime. In our case, we seek for optimizing memory for *hardware* and speculation does not apply. Hence, *correct-by-construction techniques* are required.

In the next two chapters, we will present our analysis to obtain *canonical* and *linear* memory mappings using *correct-by-construction scalable trace-based* analysis.

Chapter 4

Canonical Array Contraction

High-Level Synthesis requires *scalable* automatic parallelization. Usually, the parallelization factors applied to cover an FPGA lead to a huge circuit with hundreds of communicating buffers. However, in the polyhedral model, buffer allocation inherits from the *high unscalability* of array liveness analysis, as discussed in Chapter 3. For instance, a direct application of the successive modulo algorithm [42] would *not finish after hours*.

In this chapter, we propose a scalable algorithm to allocate buffers of a data-aware process network (DPN), the HLS intermediate representation introduced in Chapter 2. Our method relies on lightweight trace analysis and produces provably correct results in *seconds* with the *same precision than state-of-the-art array allocation algorithms*. This chapter focuses on canonical mappings $\vec{i} \mapsto \vec{i} \bmod \vec{b}$ of constant size ($\vec{b}$ is a constant vector) and shows how to reproduce the results of the successive modulo algorithm [42] with a lightweight trace analysis on a carefully selected small execution trace. Note that constant mappings are sufficient for our purpose, as we seek to synthesize each buffer (a constant size is required).

The main difficulty of this work is to produce a *correct by construction* allocation. The buffer allocation should not only work on the selected execution trace but on *any possible execution trace*. Sections 4.2 and 4.3 present the underlying hypothesis and program restrictions, experimentally shown to fit most Polybench kernels and demonstrate the underlying theoretical results required to prove the correctness of our approach. In particular, we introduce the notion of *localizability* (Section 4.2), which implies that a finite execution trace is sufficient to derive an allocation. This is completed with the notion of θ-uniformity (Section 4.3) which eases the selection of that execution trace. Then, Section 4.4 presents the steps of our algorithm (trace selection, trace generation, trace allocation) and discusses all the technical details. Finally, Section 4.5 presents the results of our methods on benchmarks of the Polybench/C suite [51] and demonstrate the *applicability* and the *scalability* of our method on real-life HLS problems.

4.1 Overview

The goal of this chapter is to exploit trace analysis to build an efficient and scalable procedure for DPN buffer allocation. Consider the example of the buffer s_1 from the Blur filter DPN introduced in the previous chapter and reproduced in Figure 4.1. The data produced at iteration $\langle C, 2, 0\rangle$ is

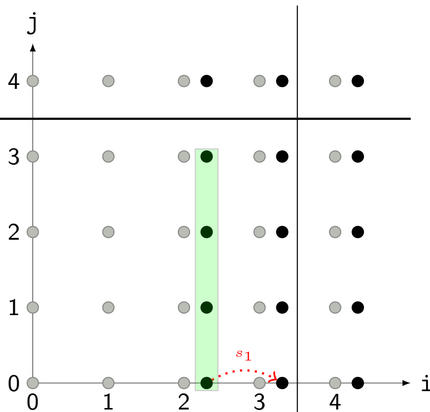


Figure 4.1: Systolization buffer s_1

consumed at iteration $\langle C, 3, 0\rangle$ and never after. Hence, it conflicts exactly with the data written by the green iterations. Since the associated direct dependence function is $h_{CC}(i, j) = (i - 1, j)$ for any $3 \le i < N, 0 \le j < N$, the *same liveness pattern will repeat.* Hence, *it is sufficient to consider the first one to infer the mapping.* The main challenge addressed in this chapter is to find the smallest execution trace which contains such a liveness pattern (Section 4.4.2). We demonstrate that it is possible provided the following conditions on the program:

- The buffers should be *localizable* (Section 4.2). Under this condition, there exists a *finite execution trace* which makes possible to find a valid mapping (Theorem 4.2.2).
- The buffers should be θ-uniform (Section 4.3). Under this condition, we can characterize and retrieve a correct execution trace (Theorem 4.4.1).

Then, we propose a complete approach to *select the execution trace* (Section 4.4.2), to *generate the trace* (Section 4.4.3) and finally to *infer the mapping from the trace* (Section 4.4.4).

4.2 Localizability

The notion of *localizability* plays a critical role in our contributions. Localizability ensures that mappings will have a constant modulo and will be computable from a single trace.

Definition 4.2.1 (Localizability) *A direct dependence edge from a source statement S to a target statement T, denoted by h_{ST}, is localizable w.r.t. an affine loop tiling ϕ (of depth n) if and only if there exists some constant c such that*

$$\forall \vec{i} \in \operatorname{dom} h_{ST},\ \phi^n_T(\vec{i}) - \phi^n_S(h_{ST}(\vec{i})) \le c$$

If $\forall \vec{i} \in \operatorname{dom} h_{ST},\ \phi^n_T(\vec{i}) - \phi^n_S(h_{ST}(\vec{i})) = c$, *then* h_{ST} *is said to be tightly localizable.*

Pluto's algorithm [14] computes an affine mapping of the form $\vec{N} \mapsto U\vec{N} + \vec{v}$ with $\phi_T(\vec{j}) - \phi_S(\vec{i}) \leq U\vec{N} + \vec{v}$ for any dependence edge $(\vec{i}, \vec{j}) \in \Delta_{ST}$. However, dependence edges Δ_{ST} *contains* direct dependence edges $(\langle S, h_{ST}(\vec{i})\rangle, \langle T, \vec{i}\rangle)$, for $\vec{i} \in \operatorname{dom} h_{ST}$. Therefore, the mapping $U_n\vec{N} + v_n$ is an upper approximation of c. When the linear part of the mapping is non-zero ($U_n \neq 0$), then the mapping cannot be expressed as a constant function for any parameter. Therefore, we cannot conclude for non-localizability. We can only conclude when $U_n = \vec{0}$ that all the dependences are *localizable*, with a constant c upper bounded by v_n.

Definition 4.2.2 (Array localizability) *An array is localizable if and only if all the dependences that use this array are localizable. In that case, their maximum localizability constant is the localizability constant of the array.*

Definition 4.2.3 (Array tight localizability) *An array is tightly localizable if and only if all the dependences that use this array are tightly localizable with the same constant c. In that case, c is the localizability constant of the array.*

Example (cont'd) Consider the buffer s_1 from the Blur filter example 2.2 introduced in Chapter 2. The tiling is $\phi_P(i,j) = \phi_C(i,j) = (j,i)$. Its only direct dependence is $h_{CC}(i,j) = (i-1,j)$ whenever $3 \leq i < N, 0 \leq j < N$. Since $\phi^2_C(i,j) - \phi^2_C(h_{CC}(i,j)) = i - (i-1) = 1$, h_{CC} is tighly localizable, so is s_1. This property is depicted in Figure 4.2. In the remainder of this section, we show that there is only the need for a constant set of operations to compute the modulo mappings. In general, this set of operations is exactly the yellow area, since DPN have a *slicing schedule* (here $\theta_P(i,j) = (i,j,0)$, $\theta_C(i,j) = (i,j,1)$).

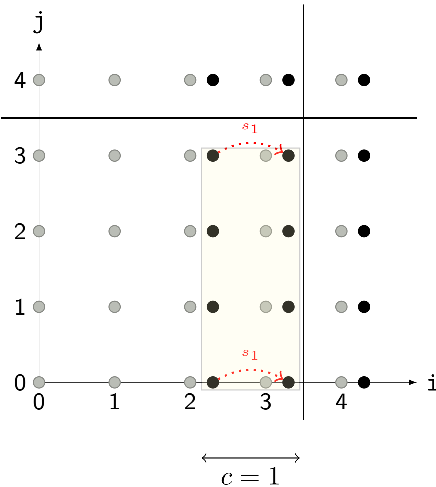


Figure 4.2: Localizability analysis: systolization buffer s_1

First, we prove that, on single assignment programs, the smallest sequence of operations required to compute a conflict set is executed between the source and the target of a *critical dependence.*

Lemma 4.2.1 (Critical dependence) *Consider a single assignment program with a single producer statement P and a single consumer statement C, and a schedule θ. There exists a direct dependence $\langle P, \vec{i}\rangle \rightarrow^{\text{FLOW}} \langle C, \vec{j}\rangle$ such that:*

$$\text{footprint}(A) \leq \text{card}\{\langle P, \vec{k}\rangle \mid \langle P, \vec{i}\rangle \preceq_\theta \langle P, \vec{k}\rangle \prec_\theta \langle C, \vec{j}\rangle\}$$

This direct dependence is called a critical dependence.

Proof. Since the program is single assignement, each array cell is written once. Hence, to simplify the presentation, a write w can be bound to the array cell written. w is alive until its last read r – note the direct dependence $w \rightarrow^{\text{FLOW}} r$. Meanwhile, all array cells w' written in the time interval $[\theta(w), \theta(r)]$ are conflicting with w: $w' \bowtie w$.

The footprint(A) is reached at some time step t_0. Consider the maximum live interval $[\theta(w), \theta(r)]$ (in terms of cardinality) such that $t_0 \in [\theta(w), \theta(r)]$. Let W be the set of array cells conflicting at t_0: by hypothesis $\text{card}\, W = \text{footprint}(A)$ and $w \in W$.

- Any $w' \in W$ is executed before r. Otherwise it would not conflict with w.
- Any $w' \in W$ is executed after w. Otherwise, that would contradicts the maximality of the interval $[\theta(w), \theta(r)]$.

Hence, any write of W is executed in the time interval $[\theta(w), \theta(r)]$.
Therefore, footprint$(A) = \text{card}\, W$ is smaller that the volume of writes in the interval $[\theta(w), \theta(r)]$.
□

Example (cont'd) On our example, footprint$(s_1) = 4$ – it is realised with the mapping $(i, j) \mapsto j \bmod 4$. A critical dependence could be $\langle C, 2, 0\rangle \rightarrow^{\text{FLOW}} \langle C, 3, 0\rangle$, and the corresponding set of writes $\{\langle C, 2, 0\rangle, \langle C, 2, 1\rangle, \langle C, 2, 2\rangle, \langle C, 2, 3\rangle\}$, whose cardinal is exactly footprint(s_1).

The next step is to prove that, on localizable buffers under the DPN partitioning, a critical dependence crosses a finite number of tiles. Since the tiles have a constant volume, they can be used as the minimal set of operations to compute the conflicts and the mapping.

Theorem 4.2.2 (Constant mapping) *If A is localizable, then* footprint(A) *is constant on the DPN partitioning scheme with constant tile size $\vec{b}$. Furthermore, the maximum number of tiles reached by a critical dependence is $\lfloor \frac{c}{b_n} \rfloor + 2$, with c the localizability constant of A and b_n the tile size in the last tiling direction.*

Proof.

- By the program being in DPN form, there exists an unique producer P and a consumer C such that the array A is entirely written by P and entirely read by C.

- Since DPN has the dynamic single assignment property, by the previous lemma, there exists some direct dependence $\langle P, h_{PC}(\vec{i})\rangle \rightarrow \langle C, \vec{i}\rangle$ such that

$$\text{footprint}(A) \leq \text{card}\{\langle P, \vec{j}\rangle \mid \langle P, h_{PC}(\vec{i})\rangle \preceq \langle P, \vec{j}\rangle \prec \langle C, \vec{i}\rangle\}$$

  Since A is localizable, there exists a constant c such that $\phi_C^n(\vec{i}) - \phi_P^n(h_{PC}(\vec{i})) \leq c$ for any $\vec{i} \in \text{dom}\, h_{PC}$ where n is the number of tiling hyperplanes. Hence $\phi_C^n(\vec{i}) \leq \phi_P^n(h_{PC}(\vec{i})) + c$. With b_n the tile size along the last hyperplane, T_P^n the *last* tile counter of $\langle P, h_{PC}(\vec{i})\rangle$ and T_C^n the *last* tile counter of $\langle C, \vec{i}\rangle$, we have:

$$\begin{aligned} T_C^n = \lfloor \tfrac{\phi_C^n(\vec{i})}{b_n} \rfloor \quad & \leq \lfloor \tfrac{\phi_P^n(h_{PC}(\vec{i}))+c}{b_n} \rfloor \\ & = \lfloor \tfrac{\phi_P^n(h_{PC}(\vec{i})}{b_n} + \tfrac{c}{b_n} \rfloor \\ & \leq \lfloor \tfrac{\phi_P^n(h_{PC}(\vec{i})}{b_n} \rfloor + \lfloor \tfrac{c}{b_n} \rfloor + 1 \\ & = T_P^n + \lfloor \tfrac{c}{b_n} \rfloor + 1 \end{aligned}$$

  The DPN ensures that $\langle P, h_{PC}(\vec{i})\rangle$ and $\langle C, \vec{i}\rangle$ belong to the same tile band. Hence the *number of tiles* reached by the critical dependence $\langle P, h_{PC}(\vec{i})\rangle \rightarrow^{\text{FLOW}} \langle C, \vec{i}\rangle$ is bounded by $\lfloor \frac{c}{b_n} \rfloor + 2$. Since the tile size is constant, there is a constant number of iterations between $\langle P, h_{PC}(\vec{i})\rangle$ and $\langle C, \vec{i}\rangle$. Hence $\text{card}\{\langle P, \vec{j}\rangle \mid \langle P, h_{PC}(\vec{i})\rangle \preceq \langle P, \vec{j}\rangle \prec \langle C, \vec{i}\rangle\}$ is constant, and therefore footprint(A) is constant. □

Example (cont'd) Consider the critical dependence instance $\langle C, 2, 0\rangle \rightarrow^{\text{FLOW}} \langle C, 3, 0\rangle$. The iterations conflicting with $\langle C, 2, 0\rangle$ are *included* in the yellow region, whose volume $(c+1) \times 4 = 8$ is constant. Hence, the footprint is constant. Note that, depending on the schedule, conflicting iterations may go outside of that region. For instance, that would be the case for an intra-tile schedule $(i, j) \mapsto (i + j, j)$. However, conflicting iterations are confined into $2 + \lfloor c/b_n \rfloor = 2 + \lfloor 1/4 \rfloor = 2$ consecutive tiles (in a band) at most, whose volume is constant by hypothesis.

The direct consequence of this theorem is that a constant mapping is possible and correct for a program in DPN partitioning with constant tile size.

Corollary 4.2.3 *Let A a localizable array. Then on the DPN partitioning scheme, there exists a correct σ_A with constant moduli.*

Checking array localizability

To check the localizability of a buffer, we propose the algorithm 3, which essentially verifies the constraints of definition 4.2.1. The algorithm is able to check the tight localizability (resulting in (TIGHLY_LOCALIZABLE,c)) and, otherwise, the localizability in the general meaning (resulting in (LOCALIZABLE,c)).

For each direct dependence, Φ describes the set of differences δ between the last tile of the destination and the source (line 4). The actual set of δ is extracted thanks to a projection (5). Then

we check the constraints of definition 4.2.1 to conclude. When all the dependences are tightly localizable with the *same* constant c (line 20), the algorithm concludes (TIGHLY_LOCALIZABLE,c). Otherwise (lines 8 and 12) the algorithm expects localizable dependences. When the set of δ is unbounded (line 15), the algorithm can directly conclude that the buffer is not localizable.

Algorithm 3: IsLocalizable

Data: Buffer to allocate A, Tiling ϕ
Result: (LOCALIZABLE,c) or (TIGHLY_LOCALIZABLE,c) with the localizability constant c, NO if A is not localizable

```
1  begin
2    c := 0
3    foreach direct dependence h_ST held by A do
4      Ψ := (δ = φ_T^last(i⃗) − φ_S^last(h_ST(i⃗)) ∧ i⃗ ∈ dom h_ST)
5      Φ := project(Ψ, {δ})
6      if Φ has only constraints δ = c' for some constant c' then
7        c := max{c, c'}
8        if not first iteration and c ≠ c' then
9          tight := false
10       end
11     else
12       if Φ has a constraint δ ≤ c' for some constant c' then
13         c := max {c, min{c' | δ ≤ c' is a constraint of Φ}}
14         tight := false
15       else
16         return NO
17       end
18     end
19   end
20   if tight then
21     return TIGHTLY_LOCALIZABLE(c)
22   else
23     return LOCALIZABLE(c)
24   end
25 end
```

Example (cont'd) Consider the buffer s_1 for the DPN implementing Blur filter with the tiling $\phi_P(i,j) = \phi_C(i,j) = (j,i)$. Its direct dependence function is $h_{CC}(i,j) = (i-1,j)$ for any $3 \le i < N, 0 \le j < N$. Applying our algorithm, we have:

$$\Psi := (\delta = \phi_C(i,j) - \phi_C(h_{CC}(i,j))) \wedge 3 \le i < N \wedge 0 \le j < N$$

Since $\phi_C^2(i,j) - \phi_C^2(h_{CC}(i,j)) = i - (i-1) = 1$, this simplifies to:

$$\Psi := (\delta = 1) \wedge 3 \leq i < N \wedge 0 \leq j < N$$

And then: $\Phi = \{\delta \mid \delta = 1\}$, hence we deduce that the buffer s_1 is tightly localizable with localizability constant $c = 1$.

As it will discussed later in Section 4.5.2, most polybench kernels lead to DPN with *tightly localizable* buffers with a small localizability constant. Hence, localizability is not an actual limitation.

4.3 θ-uniformity

When a buffer is localizable, it is sufficient to consider the execution trace between the source tile and the target tile of a critical dependence $s \rightarrow^{\text{FLOW}} t$ to compute the mapping. In this section, we present an additional constraint on the dependences to ease the location of a critical dependence, and thereby find the right trace.

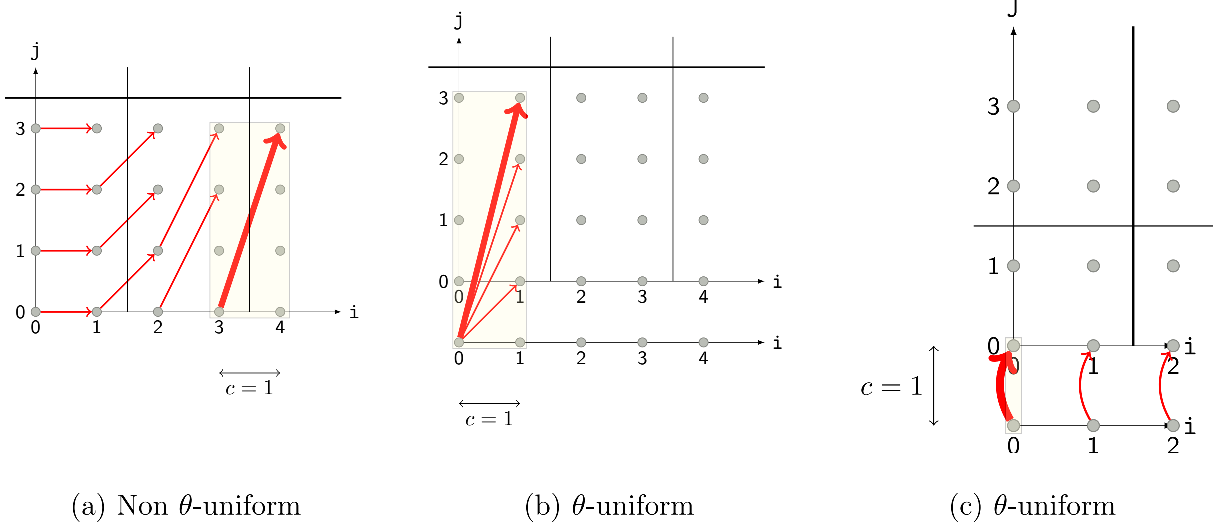
Consider the direct dependence depicted on Figure 4.3.(a). Producer and consumer are instances of the same statement, with tiling $\phi(i,j) = (j,i)$ and intra tile schedule $\theta(i,j) = (i,j)$ and direct dependences $(i,j) \rightarrow^{\text{FLOW}} (i+1, j+i)$ for any valid iterations (i,j) and $(i+1, j+i)$. The dependence is tighly localizable with $c = \phi^2(i+1, j+i) - \phi^2(i,j) = i+1-i = 1$. In that case, a critical dependence is only reached from the second to the third tile (denoted by the bold arrow). Hence one can consider an execution trace with $N \geq 4$ to compute a correct mapping. Since the dependence distance changes while moving along the direction of the tile band, one should evaluate and maximize the dependence distance to find the critical dependence. To avoid that situation, one may consider a simplified program model where *dependences are invariant by translation* along the direction of the tile band. This way, focusing only on the first full tiles only will be correct for parameter selection.

First, we need to define the *tile band direction*. On the previous example, it is the vector $\vec{\beta} = (1,0)$. In general, one has to deal with non-perfect loop nests where some statements may have a dimension lower than the tiling itself. Hence, we rely on the following result:

Theorem 4.3.1 (Tile band direction) *Consider a tiling $\phi_S(\vec{i}) = A_S\vec{i} + B_S\vec{N} + \vec{c}$ of depth n and A'_S the matrix with the $n-1$ first lines of A_S. Then, there exists $\vec{\beta}_S$ such that the vector right $\mathbb{R}\vec{\beta}_S$ is the solution of $A'_S\vec{i} = 0$. $\vec{\beta}_S$ is called a tile band direction of ϕ_S.*

Proof. Note that by construction of the tiling ϕ_S, A_S has exactly $\dim D_S$ linearly independant line vectors. When removing the last line of A_S, this either removes one of those vectors (case 1: $\operatorname{rg} A'_S = \dim D_S - 1$) or not (case 2: $\operatorname{rg} A'_S = \dim D_S$). Applying the rank theorem to $\vec{i} \mapsto A'_S\vec{i}$, we have: $\dim D_S = \operatorname{rg} A'_S + \dim\ker A'_S$ (*).

- Case 1: (*) leads to $\dim D_S = \dim D_S - 1 + \dim\ker A'_S$, hence $\dim\ker A'_S = 1$. Let $\vec{\beta}_S$ be a basis of $\ker A'_S$, then: $\ker A'_S = \mathbb{R}\vec{\beta}_S$.
- Case 2: (*) leads to $\dim D_S = \dim D_S + \dim\ker A'_S$, hence $\dim\ker A'_S = 0$. Hence $\ker A'_S = \{\vec{0}\}$ and then: $\vec{\beta}_S = \vec{0}$. □



(a) Non θ-uniform (b) θ-uniform (c) θ-uniform

Figure 4.3: Locating the critical dependence

Note that case 2 can only arise on statements S whose iteration dimension $\dim D_S$ is strictly less than the tiling dimension n. Typically, those are the initilization statements before a reduction, for instance $C[i][j] = 0$ in the matrix product.

Example (cont'd)

- Consider Figure 4.3.(a). The unique statement has a fully dimensional tiling $\phi(i, j) = (j, i)$. Hence its tile band direction satisfies $j = 0$, generated by $\vec{\beta} = (1, 0)$.
- *Example (b).* We have two statements P (the one-dimension producer) and C (the two-dimensional consumer) with $\phi_P(i) = (0, i)$ and $\phi_C(i, j) = (j, i)$. The tile band direction for ϕ_P satisfies $0 = 0$, hence $\vec{\beta}_P = (1)$. Note that we are on a degenerate version of case 1, the “linearly independent” dimension i was removed. The tile band direction for ϕ_C satisfies $j = 0$, hence $\vec{\beta}_C = (1, 0)$.
- *Example (c).* Again, we have two statements P (the one-dimension producer) and C (the two-dimensional consumer) with $\phi_P(i) = (i, 0)$ and $\phi_C(i, j) = (i, j)$. The tile band direction for ϕ_P satisfies $i = 0$, hence $\vec{\beta}_P = (0)$. Now, it is a degenerate version of case 2: the “linearly independant” dimension i was kept. The tile band direction for ϕ_C satisfies $i = 0$, hence $\vec{\beta}_C = (0, 1)$.

Since we consider non-perfect loop nests, there is no straightforward notion of dependence vector. However, all the operations are plunged into the *same set* of execution dates throught the scheduling function θ. Hence the idea to consider the difference of execution dates between the target and the source of a direct dependence.

Definition 4.3.1 (θ-dependence function) *Consider a scheduling function θ and a direct dependence $\langle P, h_{PC}(\vec{i})\rangle \rightarrow^{\text{FLOW}} \langle C, \vec{i}\rangle$. The associated θ-dependence function is defined as $\Delta\theta_{PC}(\vec{i}) = \theta_C(\vec{i}) - \theta_P(h_{PC}(\vec{i}))$.*

The execution dates are assumed to have the same dimension. In practice, we can always enforce this property by padding execution dates with 0s.

Example (cont'd)

- *Example (a)* has the intra tile schedule $\theta(i,j) = (i,j)$ and direct dependence $(i,j) \rightarrow^{\text{FLOW}} (i+1, j+i)$. Hence $\Delta\theta(i,j) = (i+1, j+i) - (i,j) = (1,i)$: when going forward in the band direction $\vec{\beta}_S = (1,0)$, i increases, and so does $\Delta\theta(i,j)$. Our goal is to avoid that situation by *enforcing a constant vector* when moving along the tile band direction.

- *Example (b)* has the intra-tile schedule $\theta_P(i) = (i,0)$ and $\theta_C(i,j) = (i, j+1)$ and the dependence $\langle P, i-1\rangle \rightarrow^{\text{FLOW}} \langle C, i, j\rangle$. Hence $\Delta\theta_{PC}(i,j) = (i, j+1) - (i-1, 0) = (1, j+1)$. When moving in the tile band direction $\beta_C = (1,0)$, $\Delta\theta_{PC}$ stays constant: we have the same dependence pattern along the tile band.

- *Example (c)* has the intra-tile schedule $\theta_P(i) = (0,i)$ and $\theta_C(i,j) = (j, i+1)$ and the direct dependence $\langle P, i\rangle \rightarrow^{\text{FLOW}} \langle C, i, 0\rangle$. Hence $\Delta\theta_{PC}(i,j) = (0, i+1) - (0, i) = (0,1)$. This is a constant vector, which will then stays constant along the tile band.

A dependence is θ-uniform if it does not change when moving on the tile band direction:

Definition 4.3.2 (θ-uniform dependence) *Consider a direct dependence d, its θ-dependence function $\Delta\theta_{PC}$, and the tile band direction $\vec{\beta}_C$. The dependence is θ-uniform if for any valid target iteration $\vec{i}$,*

$$\Delta\theta_{PC}(\vec{i}) = \Delta\theta_{PC}(\vec{i} + \vec{\beta}_C)$$

Note that *uniform dependences are θ-uniform.* Indeed, a uniform dependence has a constant $\Delta\theta_{PC}$, in particular $\Delta\theta_{PC}(\vec{i}) = \Delta\theta_{PC}(\vec{i} + \vec{\beta}_C)$. On our main motivating example, the direct dependence for the buffer s_1 is uniform (with vector $(1,0)$), hence it is θ-uniform.

Finally, we extend θ-uniformity to arrays, as for localizability:

Definition 4.3.3 (θ-uniform buffer) *A buffer A is θ-uniform if and only if all the direct dependences held by A are θ-uniform.*

When a buffer is θ-uniform, there is a *finite dependence pattern* repeating along the tile band direction. Hence, the trace only needs to contain the *first instance* of that dependence pattern. To enforce critical dependences, this first instance should be made of *full tiles.* This will be the purpose of the parameter selection algorithm presented in section 4.4.2.

Note that θ-uniformity does *not* imply localizability. Consider for instance the degenerate case of a single dependence $d = \langle P, h_{PC}(\vec{i}_0)\rangle \rightarrow^{\text{FLOW}} \langle C, \vec{i}_0\rangle$ from the first tile to the last tile of tile band. $\operatorname{dom} h_{PC}$ is restricted to a single point $\vec{i}_0$. Hence $\operatorname{aff}\operatorname{dom} h_{PC} = \{\vec{i}_0\}$ and $\vec{\beta}_C \not\parallel \operatorname{dom} h_{PC}$ which makes the dependence θ-uniform. However, it is not localizable as $\phi^n_C(\vec{i}_0) - \phi^n_P(h_{PC}(\vec{i}_0)$

is a width of the iteration domain which may be parametrized, hence it is not bounded by a constant c.

However localizability and θ-uniformity seem to imply tight-localizability. In the following, ***we will consider tightly localizable and θ-uniform buffers***. Again, as it will be discussed in Section 4.5.2, most Polybench kernels are compiled to DPNs whose buffers have those properties. Hence, it is not an actual limitation.

Checking θ-uniformity

We give an easy-to-check sufficient condition for the θ-uniformity of a dependence, which is used in Algorithm 4.

First, if *no valid iterations $\vec{i}$ and $\vec{i}+\vec{\beta}_C$ exists* in $\operatorname{dom} h_{PC}$, *we may immediatly conclude that the dependence is θ-uniform* – indeed, the dependence is not used along the tile band. For having $\vec{i}$ and $\vec{i}+\vec{\beta}_C$ in $\operatorname{dom} h_{PC}$, it is sufficient to have $\vec{\beta}_C$ as a *line* of $\operatorname{aff} \operatorname{dom} h_{PC}$, the *affine hull* of $\operatorname{dom} h_{PC}$ (*obtained by keeping only the equalities of* $\operatorname{dom} h_{PC}$). Let us write $\{\vec{i} \mid A\vec{i}+B\vec{N}+\vec{c}=0\}$ the *equalities of* $\operatorname{dom} h_{PC}$. It is sufficient to have $A\vec{\beta}_C = 0$. In our algorithm, this test is written as $\vec{\beta}_C \uparrow \operatorname{dom} h_{PC}$ (line 3).

If the test fails, we can conclude that the dependence is θ-uniform. Otherwise, we still need to check that $\Delta\theta_{PC}(\vec{i}) = \Delta\theta_{PC}(\vec{i}+\vec{\beta}_C)$ for any $\vec{i}, \vec{i}+\vec{\beta}_C \in \operatorname{dom} h_{PC}$. Generally, $\Delta\theta_{PC}$ is an affine mapping $\vec{i} \mapsto A_{PC}\vec{i}+B_{PC}\vec{N}+\vec{c}_{PC}$. Hence, we need to check that $A_{PC}\vec{i}+B_{PC}\vec{N}+\vec{c}_{PC} = A_{PC}(\vec{i}+\vec{\beta}_C)+B_{PC}\vec{N}+\vec{c}_{PC}$, which simplifies to $A_{PC}\vec{\beta}_C = 0$ or $\operatorname{lin}\Delta\theta_{PC}(\vec{\beta}_C) = 0$ (line 5).

Algorithm 4: IsThetaUniform

Data: Buffer to allocate A, Tiling ϕ, Tile band direction (per statement) $\vec{\beta}$
Result: Is A θ-uniform? (YES or NO)

```
1  begin
2      foreach direct dependence h_PC held by A do
3          if β_C ↑ dom h_PC then
4              Set Δθ_PC(i) := θ_C(i) − θ_P(h_PC(i))
5              if lin Δθ_PC(β_C) ≠ 0 then
6                  return NO
7              end
8          end
9      end
10     return YES
11 end
```

Example (cont'd)

- *Example (a).* We have $\operatorname{dom} h_{PC} = \{(i,j) \mid 0 \le i,j < N\}$, hence $\operatorname{aff} \operatorname{dom} h_{PC} = \{(i,j) \mid \text{true}\}$ and $\vec{\beta}_C = (1,0) \uparrow \operatorname{dom} h_{PC}$. Also, $\operatorname{lin}\Delta\theta_{PC}(i,j) = (0,i)$.

Hence $A_{PC}\vec{\beta}_C = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \neq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. Hence the dependence is *not* θ-uniform.

- *Example (b).* Again, $\operatorname{dom} h_{PC} = \{(i,j) \mid 0 \leq i,j < N\}$, hence $\operatorname{aff} \operatorname{dom} h_{PC} = \{(i,j) \mid \text{true}\}$ and $\vec{\beta}_C = (1,0) \uparrow \operatorname{dom} h_{PC}$. Also, $\operatorname{lin} \Delta\theta_{PC}(i,j) = (0,j)$.
  Hence $A_{PC}\vec{\beta}_C = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. Hence the dependence is θ-uniform.

- *Example (c).* $\operatorname{dom} h_{PC} = \{(i,j) \mid 0 \leq i < N, j = 0\}$, hence $\operatorname{aff} \operatorname{dom} h_{PC} = \{(i,j) \mid j = 0\}$ and $\vec{\beta}_C = (0,1) \not\uparrow \operatorname{dom} h_{PC}$ – the check yields $1 = 0$; the dependence does not go along the tile band. Hence, we can conclude that the dependence is θ-uniform.

4.4 Trace-based Array Contraction

In this section, we present our algorithm to infer a correct buffer allocation using execution traces. We first outline our algorithm, then we detail the steps.

4.4.1 Overview

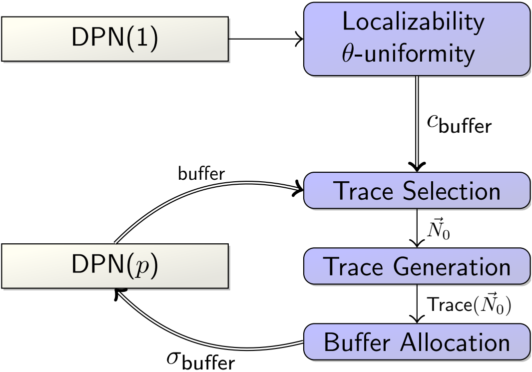


Figure 4.4: Overview of our approach

Our approach is summarized on Figure 4.4. First, localizability analysis (Algorithm 3) and θ-uniformity analysis (Algorithm 4) are applied to the *non-parallelized* version of the DPN (DPN(1)) to reduce the cost of our approach. Indeed, each buffer b_k of DPN(p) (DPN(1) parallelized with a factor p) solves a *subset of direct dependences* solved by a buffer b of DPN(1). If a set of dependence instances is localizable (resp. θ-uniform), then any subset will share this property. Hence, the localizability and the θ-uniformity of b with ensure those of b_k.

Then, *for each buffer of DPN(p)*, we compute the minimal parameters required to produce an execution trace of sufficient size to infer a correct buffer allocation (*Trace selection*, Section 4.4.2). Then, we generate the trace. In particular, we will show the refinements to operate solely

on the relevant parts of the trace (Section 4.4.3). Finally, we will present our array contraction algorithm that analyses this trace (Section 4.4.4).

4.4.2 Trace Selection

For each buffer, we select an execution trace (with parameters value $\vec{N}$) which makes possible to infer a correct mapping, while being as small as possible to ensure the efficiency of the whole process. To ensure the correctness, this execution trace must contain a *critical dependence* (Lemma 4.2.1).

Since the buffer is localizable, Theorem 4.2.2 ensures the existence of such a trace and predicts that it will cross at most $\lfloor \frac{c}{b_n} \rfloor + 2$ consecutive tiles of a tile band, with c the localizability constant of the buffer and b_n the tile size in the last tiling direction. *We enforce that number of tiles to 2 by assuming a tile size $b_n \geq c + 1$. Since DPN have a slicing schedule, the mapping obtained this way will be correct for any greater b_n.*

We assume θ-uniform buffers solving direct dependences generated by a single dependence function h_{PC}, and that dependence instances from full tiles to full tiles exists when the program parameters are big enough. In practice, most buffers satisfy this property. Under these conditions, there exists a critical dependence starting from the first full tile reached by a producer instance:

Theorem 4.4.1 (Parameter selection) *Consider a θ-uniform DPN buffer resolving only dependences $d(\vec{i}) = \langle P, h_{PC}(\vec{i})\rangle \rightarrow^{\text{FLOW}} \langle C, \vec{i}\rangle$, for any $\vec{i} \in \operatorname{dom} h_{PC}$. If d has an instance from a full tile to a full tile for program parameters big enough, Then, there exists a critical dependence instance $\langle P, h_{PC}(\vec{i}_0)\rangle \rightarrow^{\text{FLOW}} \langle C, \vec{i}_0\rangle$ such that $\langle P, h_{PC}(\vec{i}_0)\rangle$ belongs to the first full tile reached by the producer operations $\mathcal{P} = \{\langle P, h_{PC}(\vec{i})\rangle \mid \vec{i} \in \operatorname{dom} h_{PC}\}$.*

Proof. Since the critical dependence $d_0 = \langle P, h_{PC}(\vec{i}_0)\rangle \rightarrow^{\text{FLOW}} \langle C, \vec{i}_0\rangle$ is θ-uniform, we can translate it to the dependence $d_0' = \langle P, h_{PC}(\vec{i}_0 + k.\vec{\beta}_C)\rangle \rightarrow^{\text{FLOW}} \langle C, \vec{i}_0 + k.\vec{\beta}_C\rangle$, i.e. into any full tile covered by $\operatorname{dom} h_{PC}$ that shares the same schedule difference. Hence, the volume of writes between the source and the target of d_0' is at least the same as for d_0, and d_0' is also critical. In particular, we can choose a translation so that $\langle P, h_{PC}(\vec{i}_0 + k.\vec{\beta}_C)\rangle$ belongs to the first full tile reached by the producer operations $\mathcal{P} = \{\langle P, h_{PC}(\vec{i})\rangle \mid \vec{i} \in \operatorname{dom} h_{PC}\}$. □

Hence, our algorithm 5 selects the minimum parameter value so that any direct dependence has at least one instance from a full tile to a full tile. Since inter-tile dependences have a dependence distance greater or equal than intra-tile dependence, we impose that these dependences cross a tile whenever possible (line 3). If not, we consider intra-tile dependence instances (line 4). Since critical dependences cover at most 2 tiles ($b_n \geq c + 1$, see above), we are sure to reach a critical dependence. If not, there is no dependence instance from a full tile to a full tile. Hence the hypothesis of Theorem 4.4.1 are not fulfilled and we conservatively fail (line 6). In that case, the buffer will not be contracted with our method. Finally, we keep the minimum feasible parameters $\vec{N}$ (line 10). Note that the set of feasible parameters Φ is usually a *quadrant* (constraints $N_1 \geq v_1, \dots N_k \geq, v_k$). In that case, $\min_{\ll}$ might be extracted syntactically from the constraints. Otherwise, we rely on integer linear programming.

Algorithm 5: GetParameters

Data: Buffer to contract A, dependence function h_{PC}, Tiling ϕ

Result: Parameter instance $\vec{N}_0$, FAIL if the hypothesis of Theorem 4.4.1 are not satisfied

```
1  begin
2      Ψ := i⃗ ∈ dom h_PC ∧ T_P = tile_P(h_PC(i⃗)) ∧ T_C = tile_C(i⃗) ∧ fulltile_P(T_P) ∧ fulltile_C(T_C)
3      Φ := project(Ψ ∧ T_P^n < T_C^n, N⃗)
4      if Φ = ∅ then
5          Φ := project(Ψ ∧ T_P^n = T_C^n, N⃗)
6          if Φ = ∅ then
7              return FAIL
8          end
9      end
10     N⃗_0 := min_≪ Φ
11     return N⃗_0
12 end
```

Example (cont'd) In the remainder, we get back to the example of the buffer s_1, illustrated on Figure 4.2. The direct dependence function is $h_{CC}(i,j) = (i-1, j)$ for any $3 \le i < N, 0 \le j < N$. Writing $T_C = (T_{C_1}, T_{C_2})$ the source tile (with $\langle C, i-1, j\rangle$) and $T'_C = (T'_{C_1}, T'_{C_2})$ the target tile (with $\langle C, i, j\rangle$), the constraints $T_C = \text{tile}_C(h_{CC}(\vec{i})) \wedge T'_C = \text{tile}_C(\vec{i})$ are:

$$\begin{array}{l} 4T_{C_1} \le j < 4(T_{C_1}+1), 4T_{C_2} \le i-1 < 4(T_{C_2}+1), \\ 4T'_{C_1} \le j < 4(T'_{C_1}+1), 4T'_{C_2} \le i < 4(T'_{C_2}+1) \end{array}$$

The full tile constraints $\text{fulltile}_C(T_C) \wedge \text{fulltile}_C(T'_C)$ are:

$$\begin{array}{l} 4T_{C_1} \le N-4, T_{C_1} \ge 0, 4T_{C_2} \le N-4, T_{C_2} \ge 1, \\ 4T'_{C_1} \le N-4, T'_{C_1} \ge 0, 4T'_{C_2} \le N-4, T'_{C_2} \ge 1 \end{array}$$

In particular, the first tile $(0,0)$ is *not* full. Hence:

$$\Psi := \left\{ \begin{array}{ll} (T_{C_1}, T_{C_2}, T'_{C_1}, T'_{C_2}, i, j) \mid & 3 \le i < N, 0 \le j < N, \\ & 4T_{C_1} \le j < 4(T_{C_1}+1), 4T_{C_2} \le i-1 < 4(T_{C_2}+1), \\ & 4T'_{C_1} \le j < 4(T'_{C_1}+1), 4T'_{C_2} \le i < 4(T'_{C_2}+1), \\ & 4T_{C_1} \le N-4, T_{C_1} \ge 0, 4T_{C_2} \le N-4, T_{C_2} \ge 1, \\ & 4T'_{C_1} \le N-4, T'_{C_1} \ge 0, 4T'_{C_2} \le N-4, T'_{C_2} \ge 1 \end{array} \right\}$$

Then, we get: $\Phi = \text{project}(\Psi \wedge T_{C_2} < T'_{C_2}, N) = \{N \mid N \ge 12\}$ and finally: $N_0 = \min_{\ll} \Phi = 12$, which covers 3×3 tiles. Since the critical dependence is already in the tile $(1,0)$, it would have been sufficient to choose $N = 8$. However, our algorithm conservatively assumes that the critical dependence could hold between two consecutive tiles and choose $N = 12$.

4.4.3 Fast Trace generation

A trace is an application $T : t \mapsto (W(t), R(t))$ mapping each execution date t (obtained from the scheduling function) to a set of array cells written $W(t)$ and a set of array cells reads $R(t)$ such that, at date t, we execute the reads $R(t)$ in parallel, *and then* the writes $W(t)$ in parallel. The trace will be used in section 4.4.4 to compute the liveness information of arrays cells, and then the mapping.

We generate the trace using the algorithm 7. Given a program P, with a schedule affine per statement θ and a parameter binding σ, mapping each program parameter to a constant value, the trace is generated separately for each statement (line 4). In turn, Algorithm 6 generates the trace recursively for each statement. For each iteration of the enclosing loop (line 9), the iterations of the inner loops are recursively generated. The current enclosing iteration vector is stored in the binding σ, mapping loop counters to their current values. Whenever the inner statement is reached (line 2), there is an operation $\langle S, \vec{i} \rangle$, where $\vec{i}$ is obtained from σ. In that case, the algorithm returns an *elementary trace* $t \mapsto (W(t), R(t))$ where $t = \theta_S(\vec{i})$, $W(t)$ is the singleton with the array cell written by $\langle S, \vec{i} \rangle$ and $R(t)$ is the set of array cells read by $\langle S, \vec{i} \rangle$. For each recursive call, the traces obtained recursively are fused using an union operator $\sqcup$ such that $T \sqcup T'(t) = (W(t) \cup W'(t), R(t) \cup R'(t))$ if $T(t) = (W(t), R(t))$ and $T'(t) = (W'(t), R'(t))$. If some $T(t)$ is not defined, $T \sqcup T'(t) = T'(t)$. If some $T'(t)$ is not defined, $T \sqcup T'(t) = T(t)$. Thanks to the trace structure, the actual interpretation order of the operations does not matter, as they would be reordered anyway.

Algorithm 6: TRACEGEN_STMT

Data: Statement S, Iteration domain D_S, Scheduling function θ_S, counter and parameter binding σ_S

Result: Trace T_S

```
1  begin
2      if σ_S binds all counters of D_S then
3          return ENTRY(T_S, S, D_S, σ_S, θ_S)
4      else
5          T_S := ∅
6          Let i be the next loop counter enclosing S, not assigned in σ_S
7          ℓ := max{c ∈ ℤ | i ≥ c is a constraint of D'_S}
8          u := min{c ∈ ℤ | i ≤ c is a constraint of D'_S}
9          for i_value = ℓ to u do
10             σ'_S := σ_S[i ↦ i_value]
11             T_S := T_S ⊔ TRACEGEN_STMT(S, σ_S(D_S), θ_S, σ_S)
12         end
13         return T_S
14     end
15 end
```

Note that this algorithm works providing the iteration domains D_S are *row-echelon*: the constraints involving a counter c must only use *counters from enclosing loops*. Otherwise, the

Algorithm 7: TRACEGEN

```
  Data: Program P, Scheduling function θ, parameter binding σ
  Result: Trace T_P
1 begin
2   T := ∅
3   foreach Statement S of P do
4     T := T ⊔ TRACEGEN_STMT(S, σ(D_S), θ_S, σ)
5   end
6   return T
7 end
```

loop bounds ℓ and u (line 9) would not evaluate to integers. Usually, this happens when the constraints are directly extracted from a loop nest.

Handling loop tiling

In general, the iteration domains for tiled programs are *not* row echelon, as tile counters are usually defined with additional constraints.

On the example of the s_1 buffer, the tiled iteration domain is $D = \{(T_1, T_2, i, j) \mid 2 \leq i < N, 0 \leq j < N, 4T_1 \leq j < 4(T_1 + 1), 4T_2 \leq i < 4(T_2 + 1)\}$, and the enclosing counters are T_1, T_2, i, j, in that order. Hence T_1 is expected to not depend on any counter, T_2 to depend on T_1 at most, i to depend on T_1, T_2 at most and j to depend on T_1, T_2, i; which is not the case with the constraints of D.

Hence, we apply the algorithm 8, inspired from the Ancourt & Irigoin code generation method [10]. For each counter, the algorithm uses a projection to keep only constraints with enclosing counters (line 5), ensuring a row-echelon form.

Algorithm 8: TOROWECHELON

```
  Data: Iteration domain D_S, list of nesting counters ℓ_C
  Result: Row-echelon domain R_S
1 begin
2   foreach counter i of ℓ_C in reverse order do
3     R_S := R_S ∪ {constraints of D_S involving i}
4     ℓ_C := ℓ_C \ i
5     D_S := proj(D_S, ℓ_c)
6   end
7   return R_S
8 end
```

Example (cont'd) On the example, we obtain the following constraints. At the first iteration (j counter), we filter the constraints $4T_1 \leq j < 4(T_1 + 1), 0 \leq j < N$.

- At next iteration (counter i), projecting on (T_1, T_2, i) and filtering the constraints with i yields: $4T_2 \leq i < 4(T_2 + 1), 2 \leq i < N$.
- At next iteration (counter T_2), projecting on (T_1, T_2) and filtering the constraints with T_2 yields: $T_2 \geq 0, 4T_2 < N$.
- At next iteration (counter T_1), projecting on T_1 and filtering the constraints with T_1 yields: $T_1 \geq 0, 4T_1 < N$.

Finally, the equivalent row-echelon domain is:

$$\left\{ \begin{array}{r} (T_1, T_2, i, j) \mid T_1 \geq 0, 4T_1 < N, T_2 \geq 0, 4T_2 < N, 4T_2 \leq i < 4(T_2 + 1), 2 \leq i < N, \\ 4T_1 \leq j < 4(T_1 + 1), 0 \leq j < N \end{array} \right\}$$

Generally, the original non-tile domain (i, j) is already in row-echelon. Hence, we just need to process the tile counters.

Prune useless operations

Finally, the algorithm is slightly modified to *stop once* the first two consecutive full tiles are reached, which is the relevant execution trace according to Theorem 4.4.1. We just need to play with the full tile predicate $\text{fulltile}_S(T)$ (examplified in Section 4.4.2) for each statement S (producer and consumer) to detect that situation. Operations prior to these full tiles are kept in the trace.

Example (cont'd) For sizing s_1, we build a producer/consumer program summarizing the writes and the reads to s_1 with the same schedule $(\theta_C(T_1, T_2, i, j) = (T_1, T_2, i, j))$. With $N = 12$, the two first consecutive full tiles are $(T_1, T_2) \in \{(0, 1), (0, 2)\}$. Hence the trace is made of read and write access for the operations $\{\langle C, i, j\rangle \mid 2 \leq i < 12, 0 \leq j < 4\}$. The first trace entries are:

Timestamp t	$W(t)$	$R(t)$
(0,1,2,0)	$\{s_1[2, 0]\}$	$\emptyset$
(0,1,2,1)	$\{s_1[2, 1]\}$	$\emptyset$
(0,1,2,2)	$\{s_1[2, 2]\}$	$\emptyset$
(0,1,2,3)	$\{s_1[2, 3]\}$	$\emptyset$
(0,1,3,0)	$\{s_1[3, 0]\}$	$\{s_1[2, 0]\}$
(0,1,3,1)	$\{s_1[3, 1]\}$	$\{s_1[2, 1]\}$
(0,1,3,2)	$\{s_1[3, 2]\}$	$\{s_1[2, 2]\}$
(0,1,3,3)	$\{s_1[3, 3]\}$	$\{s_1[2, 3]\}$
...	...	...

Note that these first trace iterations already reach a critical dependence through $s_1[2, 0]$, for instance.

4.4.4 Trace buffer allocation

Conflict set Array contraction relies on *conflict set* Δ_a, obtained from a *conflict relation* $\bowtie$: $\Delta_a = \{\vec{i} - \vec{j} \mid a[\vec{i}] \bowtie a[\vec{j}]\}$. On a trace, we can simply apply the classical liveness analysis, by considering each array access as a separate location. For each trace entry $t \mapsto (W(t), R(t))$, we have:

$$\begin{cases} \text{In}(t) & = & (\text{Out}(t) \cup R(t)) \setminus W(t) \\ \text{Out}(t) & = & \text{In}(t+1) \quad \text{if } t+1 \text{ is a trace entry} \end{cases}$$

We implemented set operations with *bit sets* to accelerate the computation. Then, $a[\vec{i}] \bowtie a[\vec{j}] \iff a[\vec{i}], a[\vec{j}] \in \text{In}(t)$ for some t. Finally, we deduce the conflict set Δ_a by computing $\vec{i} - \vec{j}$ for each pair of conflicting array cells.

On our example, we have the following liveness. To simplify the presentation, the array cell $s_1[i, j]$ is denoted ij.

Timestamp t	$W(t)$	$R(t)$	$\text{In}(t)$
(0,1,2,0)	$\{s_1[2,0]\}$	$\emptyset$	$\emptyset$
(0,1,2,1)	$\{s_1[2,1]\}$	$\emptyset$	$\{20\}$
(0,1,2,2)	$\{s_1[2,2]\}$	$\emptyset$	$\{20, 21\}$
(0,1,2,3)	$\{s_1[2,3]\}$	$\emptyset$	$\{20, 21, 22\}$
(0,1,3,0)	$\{s_1[3,0]\}$	$\{s_1[2,0]\}$	$\{20, 21, 22, 23\}$
(0,1,3,1)	$\{s_1[3,1]\}$	$\{s_1[2,1]\}$	$\{21, 22, 23, 30\}$
(0,1,3,2)	$\{s_1[3,2]\}$	$\{s_1[2,2]\}$	$\{22, 23, 30, 31\}$
(0,1,3,3)	$\{s_1[3,3]\}$	$\{s_1[2,3]\}$	$\{23, 30, 31, 32\}$
...	...	...	...

This leads to the conflict set depicted in Figure 2.9.(b). For instance, the verticals points with $\delta i = 0$ might be obtained from the differences of conflicting cells of $\text{In}(0, 1, 3, 0) = \{20, 21, 22, 23\}$. The vertical points with $\delta i = 1$ might be obtained from the differences of conflicting cells of $\text{In}(0, 1, 3, 3) = \{23, 30, 31, 32\}$.

Array contraction Once Δ_a is obtained, we can derive a mapping by applying a trace version of the successive modulo algorithm. Algorithm 9 presents the whole method. After computing Δ_A as a finite set of points (line 4), we apply our trace version of the successive modulo algorithm (line 5). Since Δ_A is no longer a polyhedron, the modulo $\vec{b_k}$ are simply computed by iterating on the points of Δ_A. In our example, we obtain the mapping $(i, j) \mapsto (i \bmod 2, j \bmod 4)$. As stated in the background chapter, the footprint (maximum number of conflicting cells at any timestamp, which is 4 here) shows this result is not optimal $(2 \times 4 > 4)$ and might be improved by extending our approach to linear mapping. This will be the purpose of the next chapter. All the steps of this algorithm are *linear* in the trace size $|T|$, *except* the computation of Δ_A (line 4), which is in $\mathcal{O}(|T|^2)$. As we will see in the experiments, this will dominate the execution time of our approach.

Algorithm 9: GetConstantMapping

Data: Array to contract A, execution trace T

Result: Constant mapping $\vec{i} \mapsto \vec{i} \bmod \vec{b}$

1 **begin**
2 $\quad (\text{In}, \text{Out}) := \text{LIVENESS}(T)$
3 $\quad \bowtie := \bigcup_t \{(A[\vec{i}], A[\vec{j}]) \mid A[\vec{i}], A[\vec{j}] \in \text{In}(t)\}$
4 $\quad \Delta_A := \{\vec{i} - \vec{j} \mid A[\vec{i}] \bowtie A[\vec{j}]\}$
5 $\quad$ **for** *each array dimension k in increasing order* **do**
6 $\quad\quad \vec{b}_k \leftarrow 1 + \max\{\vec{\delta}_k \mid \vec{\delta} \in \Delta_A, \vec{\delta}_\ell = 0 \text{ for } 0 \leq \ell < k\}$
7 $\quad$ **end**
8 $\quad$ **return** $\vec{i} \mapsto \vec{i} \bmod \vec{b}$
9 **end**

4.5 Experimental Validation

This section presents the experimental validation of our method as a buffer allocator into an HLS tool. We first show that most of benchmark kernels fit our program restrictions (*localizability* and *θ-uniformity*), (Section 4.5.2). Then, we demonstrate the scalability of our approach on a real-life HLS process (Section 4.5.3).

4.5.1 Setup

We have applied our algorithm to allocate the buffers of the DPNs produced from the kernels of the Polybench benchmarks [51]. The setup is summarized in Figure 4.5. We have implemented our algorithm as a tool named PoLa, which represents 5637 lines of C++ code. Each DPN is produced using the Dcc compiler [7] *with a parallelization factor of 64*. Then, for each DPN buffer, Dcc feeds PoLa with a producer/consumer program summarizing the writes and reads to the buffer. In turn, PoLa analyses this program and yields the allocation (depicted as σ_{buffer}).

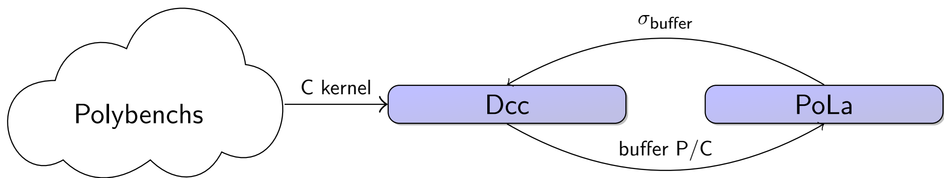


Figure 4.5: Experimental setup

Baseline The baseline is the original parametrized successive modulo algorithm (*Baseline*) and the non-parametrized relaxed successive modulo (*Relaxed baseline*) presented in Section 2.2.2. *Relaxed baseline* assumes parameters in the interval $I = [4, 4096]$. When parameters are outside of this interval, the relaxed algorithm does not come with any guarantee, which is quite limitating.

On all the examples, I happens to contain the minimum parameter value ensuring, thanks to our study, correct constant mappings for any program parameters.

The timings were measured on a PC with an Intel core i9-10885H CPU 2.40GHz with 64GB of DDR memory. All the timings are measured in seconds.

4.5.2 Applicability

Our method expects each DPN buffer to be *tightly localizable* and *θ-uniform*. Using Algorithms 3 and 4, we checked these properties on the DPN obtained from the Polybenchs kernels. As mentioned in Section 4.4.1, we consider DPNs with a parallelization factor of 1 for these checks. The results are summarized on Table 4.7. For each kernel, we check if it is fully tileable (*Tileable*). In other words, if there exists an affine tiling *without cuts*. This is a *prerequisite* to produce a DPN. If a kernel is not fully tileable, it is not considered. Then, we provide the results for tight localizability (*Tightly Localizable*) with the maximum localizability constant obtained among all the buffers (*Localizability constant*). Finally, we check for θ-uniformity of all the buffers (*θ-uniform*). We added an entry for the Blur filter discussed throughout this chapter (blur).

We observe that most Polybench kernels are tightly-localizable. Among the non-tightly localizable kernels, none are localizable. Hence, there wasn't any non-tightly localizable but localizable kernel. Also, the localizability constant is usually 1. This is because most dependences are either reduction dependences (from an iteration to the next, following the last hyperplane) or initialization dependences (as on Figure 4.3.(c)). Note the exception of stencils, where the last tiling hyperplane is slanted and leads to a localizability constant ≥ 2. Finally, we note that most of the tightly localizable kernels are also θ-uniform, with the exception of *lu* and *trisolv*. In both cases, this is due to a broadcast dependence, spreading a value across the computation. For instance, Figure 4.6 depicts the trisolve kernel (a) and the faulty dependences (b). The tiling is

```
1     for (i = 0; i < N; i++)
2       {
3  R:     x[i] = b[i];
4         for (j = 0; j <i; j++)
5  S:       x[i] = x[i] - L[i][j] * x[j];
6  T:     x[i] = x[i] / L[i][i];
7       }
```

(a) Kernel

(b) Broadcast Dependences

Figure 4.6: Trisolve is not θ-uniform

$\phi_R(i) = (i, 0), \phi_S(i, j) = (i, j), \phi_T(i) = (i, i)$, the tile band direction is $\beta_S = (0, 1)$ and the intra-tile schedule is $\theta_R(i) = (i, 0, 0), \theta_S(i, j) = (i, j, 1), \theta_T(i) = (i, i, 1)$. The dependence broadcasting the value of $x[i]$ computed by $\langle T, i\rangle$ is $\langle T, i\rangle \rightarrow^{\text{FLOW}} \langle S, i', i\rangle$ whenever $i < i' < N$. Hence

$\Delta\theta_{TS}(i', i) = \theta_S(i', i) - \theta_T(i) = (i' - i, 0, 0)$. In particular, $\text{lin}\,\theta_{TS}(\beta_S) = (-1, 0, 0) \neq (0, 0, 0)$, hence the dependence is *not* θ-uniform.

Out of eligible kernels (both tightly-localizable and θ-uniform), dcc failed to compile a parallelized DPN for *fdtd-2d*, *gesummv*, *heat-3d*, *symm* and *trmm*. Hence, we will focus on the remaining kernels: *bicg*, *gemm*, *jacobi-1d*, *jacobi-2d*, *mvt*, *seidel-2d*, *syr2k* and *syrk*. We also add the blur filter *blur* to our benchmarks.

Kernel	Tileable	Localizable	Localizability constant	θ-uniform
2mmYES	NO			
3mm	NO			
adi	YES	FAIL		
atax	NO			
bicg	YES	PASSED	1	PASSED
cholesky	YES	FAIL		
doitgen	NO			
durbin	NO	FAIL		
fdtd-2d	YES	PASSED	1	PASSED
gemm	YES	PASSED	1	PASSED
gemver	NO			
gesummv	YES	PASSED	1	PASSED
gramschmidt	NO			
heat-3d	YES	PASSED	2	PASSED
jacobi-1d	YES	PASSED	2	PASSED
jacobi-2d	YES	PASSED	2	PASSED
lu	YES	PASSED	1	FAIL
ludcmp	NO			
mvt	YES	PASSED	1	PASSED
seidel-2d	YES	PASSED	2	PASSED
symm	YES	PASSED	1	PASSED
syr2k	YES	PASSED	1	PASSED
syrk	YES	PASSED	1	PASSED
trisolv	YES	PASSED	1	FAIL
trmm	YES	PASSED	1	PASSED

Figure 4.7: Kernel eligibility statistics

4.5.3 Scalability

Figure 4.8 (a) and (b) depicts the scalability analysis of our approach. For each kernel, we compare the cumulative runtime (in seconds) of buffer allocation using our approach (blue bar in

Kernel	Tile size
bicg	$(64b, 2)$
blur	$(64b, 2)$
gemm	$(8b, 8b, 2)$
jacobi-1d	$(64b, 3)$
jacobi-2d	$(8b, 8b, 3)$
mvt	$(64b, 2)$
seidel-2d	$(8b, 8b, 3)$
syr2k	$(8b, 8b, 2)$
syrk	$(8b, 8b, 2)$

Table 4.1: Tile sizes for the scalability analysis

(a), Column *Pola* in (b)), *relaxed successive modulo* (orange bar in (a), Column *Relaxed baseline* in (b)) and the original *successive modulo* (Column *Baseline* in (b)). We provide the speedups *Relaxed baseline / Pola* (first Column *Speed-up* in (b)) and *Baseline / Pola* (second Column *Speed-up* in (b)). Different tile sizes were tested, as specified in Table 4.1. For instance, on the *bicg* kernel, we tested tile sizes $(64b, 2)$ for $b = 4$ and $b = 8$. On the *gemm* kernel, we tested tile sizes $(8b, 8b, 2)$ for $b = 4, 8, 16, 32$. Since the parallelization factor is 64, the kernel is split into processes operating on sub tiles of size $(b, 2)$ for *bicg* and $(b, b, 2)$ for *gemm*. Note that the inner tile size is set to $c + 1$ with c the maximum localizability constant, as discussed in Section 4.4.2. For instance, on blur the maximum localizability constant is $c = 1$ (Table 4.7), hence the inner tile size is $1 + 1 = 2$. As discussed, the mappings obtained this way are correct for any larger inner tile size, since DPN have a slicing schedule.

We select b as a power of 2 and we report all the results such that speed-up is observed compared to the baseline. For instance, for *bicg*, the maximum tile size resulting in a speedup is $(64 \times b, 2)$ with $b = 8$, hence $(512, 2)$.

As anticipated, the successive modulo method does not scale at all. In particular for 2D stencils *jacobi-2d* and *seidel-2d*, where the time limit of 2h30m was reached. This simply means that, by itself, this method is not suitable for large scale buffer allocation. Compared to relaxed sucessive modulo, we achieve an $\mathcal{O}(10)$ speedup in most cases. This demonstrates that, on Polybench kernels, not only our method scales, but it is better than the relaxed scalable version of successive modulo.

Figure 4.9 depicts the timings step-by-step (b) and the normalized timings (a). Column *Parameters* gives the parameters selected to generate the execution trace, Column *Trace* provides the cumulative trace size, Column *Buffers* gives the number of buffers to allocate, Columns *Pola* and *Baseline* give the cumulative execution time of our approach and the baseline and Column *Speed-up* gives the ratio *Baseline / Pola*.

Our method significantly reduces the time of buffer allocation, the most spectacular speed-ups being obtained on the *gemm* kernel. For tile size with $b = 4$, the execution time is reduced by one order of magnitude. Unsurprisingly, the execution time increases with the number of buffers to allocate (e.g. *jacobi-2d* and *seidel-2d*) and the tile size. Indeed, the bigger the tile size is, the

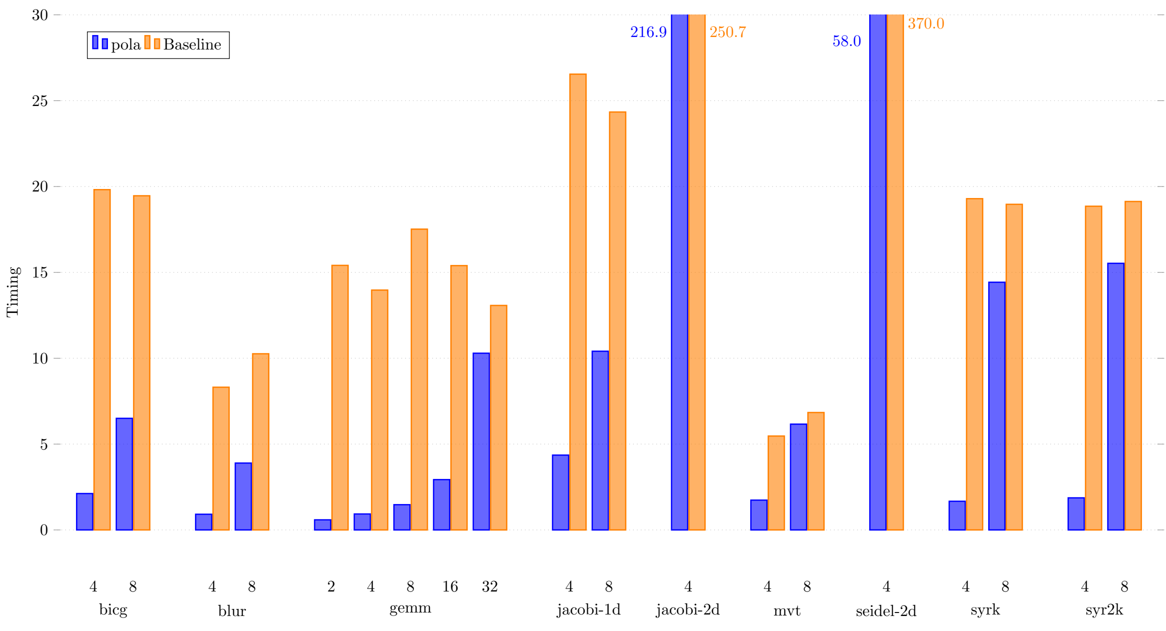
bigger the execution trace will be, hence the resulting execution time.

When the tile size increases, the execution time is dominated by the liveness analysis, followed by trace generation. For instance, on *gemm*, for smaller tile size, the trace generation (green) dominates. Then, the bigger is the tile size, the more liveness analysis (blue) dominates. This is explained by the complexity of trace generation and liveness analysis. Trace generation is linear in the trace size. Liveness analysis by itself (Algorithm 9, line 2) is linear in the trace size. However, the difference set computation (line 4) is linear in the size of the conflict relation, whose size is *quadratic in the trace size*, as we deal with dynamic single assignment programs.

4.6 Conclusion

In this chapter, we have shown how a non-scalable buffer allocation algorithm might be rephrased with a *scalable* trace analysis producing the *same result*. Specifically, we proposed a complete *correct-by-construction* trace-based algorithm to allocate DPN buffers. We prove the correctness of our algorithm under specific program hypothesis, for the buffers shall be both localizable and θ-uniform. Finally, we validate our approach on the Polybench benchmark suite. In particular, we show that our program hypothesis cover most of the polybench kernels. We demonstrate that the scalability of our algorithm for allocating channels of large-scale parallel circuits. This shows the effectiveness of the Polytrace methodology for producing scalable polyhedral compilers.

Our work may be improved in many directions. First, the experimental results show that the execution time is dominated by the computation of the difference set Δ_a, whose complexity is quadratic in the trace size. The scalability might be further improved by optimizing that part. The current version iterates on all the couples $(\vec{i}, \vec{j})$ such that $a[\vec{i}] \bowtie a[\vec{j}]$ to compute the set of $\vec{i} - \vec{j}$. This leads to build *all the points* of Δ_a, while we just need the vertices. Also, Δ_a is 0-symmetric, hence, only half of its vertices need to be computed. Focusing on the computation of those vertices would reduce drastically the overall complexity. The time spent for trace generation is the second hot spot after liveness analysis. So far, we generate the trace for two consecutive tiles to be sure that a critical dependence is reached (Theorem 4.4.1). Refining the location of a critical dependence would reduce further the execution trace, hence the trace generation time.

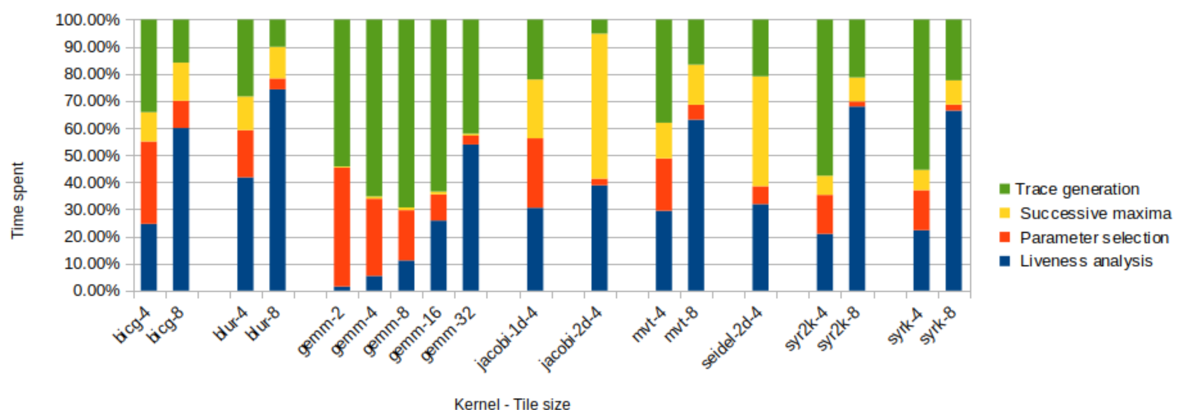


(a) Comparison with relaxed successive modulo

Kernel	b	Pola	Relaxed Baseline	**Speed-up**	Baseline	**Speed-up**
bicg	4	2.11	19.81	**9.3**	48	**22**
	8	6.49	19.46	**2.9**	51	**7.8**
blur	4	0.91	8.31	**9**	54.9	**60**
	8	3.89	10.26	**2.6**	58.7	**15**
gemm	4	0.92	13.97	**15**	227	**244**
	8	1.47	17.51	**12**	237	**161**
	16	2.93	15.39	**5.2**	224	**76**
	32	10.29	13.07	**1.2**	253	**24**
jacobi-1d	4	4.35	26.54	**6**	23m51	**329**
jacobi-2d	4	216.94	250.68	**1.15**	>2h30m	**>43**
mvt	4	1.73	5.46	**3.1**	63.8	**36**
	8	6.16	6.83	**1.1**	64.8	**10**
seidel-2d	4	58.07	370.02	**6.3**	>2h30m	**>155**
syrk	4	1.66	19.29	**11.5**	358	**214**
	8	14.42	18.96	**1.3**	460	**31**
syr2k	4	1.86	18.85	**10**	358	**191**
	8	15.52	19.12	**1.2**	460	**29**

(b) Detailed timings (seconds)

Figure 4.8: Scalability analysis. *Baseline:* Successive modulo, *Relaxed baseline:* relaxed successive modulo (Section 2.2.2)



(a) Normalized

Kernel	b	Parameters	Trace	Buffers	Parameter selection	Trace generation	Liveness analysis	Successive modulo	Total
bicg	4	$(M, N) = (256, 256)$	5120	391	0.64	0.72	0.52	0.22	2.11
	8	$(M, N) = (512, 512)$	10240	391	0.65	1.03	3.89	0.91	6.49
blur	4	$N = 256$	6144	260	0.15	0.25	0.37	0.11	0.91
	8	$N = 512$	12288	260	0.15	0.39	2.88	0.45	3.89
gemm	4	$(NI, NJ, NK) = (32, 32, 4)$	8192	198	0.26	0.60	0.05	0.00	0.92
	8	$(NI, NJ, NK) = (64, 32, 4)$	16384	198	0.27	1.01	0.16	0.01	1.47
	16	$(NI, NJ, NK) = (128, 32, 4)$	32768	198	0.28	1.86	0.75	0.02	2.93
	32	$(NI, NJ, NK) = (256, 32, 4)$	65536	198	0.33	4.33	5.54	0.07	10.29
jacobi-1d	4	$(T, N) = (256, 516)$	9972	958	1.02	0.88	1.21	0.86	4.35
jacobi-2d	4	$(T, N) = (32, 97)$	59910	2786	5.25	11.08	83.40	114.99	216.94
mvt	4	$N = 256$	4096	262	0.33	0.66	0.51	0.22	1.73
	8	$N = 512$	8192	262	0.33	1.02	3.88	0.91	6.16
seidel-2d	4	$(T, N) = (32, 67)$	51781	2230	3.54	11.31	17.23	21.89	58.07
syrk	4	$(M, N) = (4, 64)$	12224	197	0.24	0.92	0.37	0.12	1.66
	8	$(M, N) = (4, 128)$	40320	197	0.32	3.23	9.57	1.29	14.42
syr2k	4	$(M, N) = (4, 64)$	12224	198	0.26	1.07	0.39	0.13	1.86
	8	$(M, N) = (4, 128)$	40320	198	0.28	3.31	10.54	1.38	15.52

(b) Detailed timings (seconds)

Figure 4.9: Step-by-step analysis of our approach

Chapter 5

Linear Array Contraction

In this chapter, we take a different approach to memory sizing by focusing on *linear mappings* $\sigma : \vec{i} \mapsto A\vec{i} + \vec{b}$ with *parametrized modulos* $\vec{b}$. Compared to canonical mappings, linear mappings may reduce the footprint by a constant factor. Also *parametrized modulo* makes possible to apply the contribution of this chapter beyond the DPN context. As discussed in Chapter 3, there are many approaches to compute a linear mapping. However, unlike the successive modulos, specializing these algorithms on an execution trace and generalizing each trace result to a mapping working for any execution trace is quite challenging. Hence, we will not attempt to provide a trace specialization. Instead, we will exploit traces to derive the array liveness and we improve [13] to derive a *linear allocation* using ILP problems *with less constraints and variables.*

We start by presenting the program model on which our algorithm operates (Section 5.3). This class of programs is *totally unimodular programs*, with *quasi-uniform* dependences and schedule. This is not that restrictive, as most Polybench kernels fit in the category. However, tiled programs do not fit directly and would need an additional postprocessing left to future work. For instance, they might be considered if a finite subset of tiles could be processed separetely to allocate the buffers.

Then, we show how a *correct* array liveness analysis might be derived from a small number of execution traces. For each trace, a conflict polyhedron is retrieved. Then we rely on a *widening operator* ∇, to extrapolate trace-level conflict polyhedra to a generalized conflict polyhedron (Section 5.4). Because our goal is to reconstruct a generalized mapping for any parameter $\vec{N} \geq \vec{N}_0$, we want to infer a general pattern from a sequence of traces. This widening operator ∇ realises just that, by keeping only the relevant constraints. The key point is the correctness of the widening operator, whose proofs are presented in Section 5.3. Also, we present a pure polyhedral algorithm to derive a linear allocation (Section 5.5). Mainly, we reformulate the ideas of [13] to reduce the complexity of the ILP problems incurred. Finally, we present the experimental validation of our contributions (Section 5.6).

5.1 Outline

Consider again the Blur filter depicted on Figure 5.1 and presented in Chapter 2.

```
1 for(i=0; i<N; i++)
2   for(j=0; j<N; j++) {
3     blurx[i][j] = in[i][j] +
4       in[i][j+1] + in[i][j+2];
5     if(i>=2)
6       out[i][j] = blurx[i-2][j] +
7     blurx[i-1][j] + blurx[i][j];
8   }
```

(a) Kernel

(b) Liveness

Figure 5.1: Blur filter

Liveness analysis Using the methodology described in Chapter 4, we run several instances of Blur filter, e.g. for $N = 3$ and $N = 4$, and we end up with the conflict sets depicted on Figure 5.2. Observe that constraints depending on a parameter $-N < \delta j < N$ are *changing* from $N = 3$ to $N = 4$ while the others constraints are kept. This way, constraints depending on a parameter might be *detected*. Hence, we may *extrapolate* the conflict sets using a *widening operator* ∇ removing all the constraints depending on a parameter. This way, we obtain a conflict set *correct for any program parameter* consisting of three conflict polyhedra: $\nabla(\Delta_{blurx}(3), \Delta_{blurx}(4)) = \hat{\Delta}_{blurx,1} \cup \hat{\Delta}_{blurx,2} \cup \hat{\Delta}_{blurx,3}$ where:

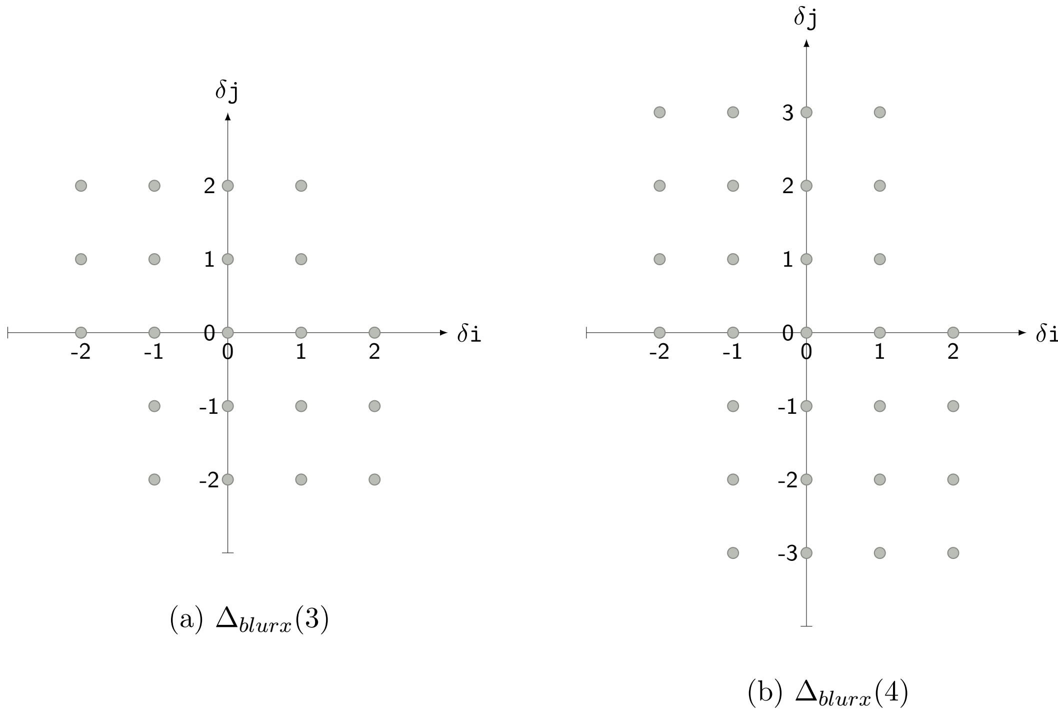
$$\begin{aligned}
\hat{\Delta}_{blurx,1} &= \{(\delta i, \delta j) \mid \delta i = -2, 0 \le \delta j\} \\
\hat{\Delta}_{blurx,2} &= \{(\delta i, \delta j) \mid -1 \le \delta i \le 1\} \\
\hat{\Delta}_{blurx,3} &= \{(\delta i, \delta j) \mid \delta i = 2, \delta j \le 0\}
\end{aligned}$$

Note that the conflict set *does not have to be monotonic* ($\vec{N} \le \vec{N}'$ implies $\Delta_a(\vec{N}) \subseteq \Delta(\vec{N}')$) to ensure the correctness of the widening. The only important point is to be able to detect and remove parametrized constraints from a few traces. This will be discussed in Section 5.2 and Section 5.3.

Finally, we apply a *narrowing* on the conflict set to enforce closed polyhedra (and then finite modulos):

$$\begin{aligned}
\Delta_{a,1} &= \{(\delta i, \delta j) \mid \delta i = -2, 0 \le \delta j < N\} \\
\Delta_{a,2} &= \{(\delta i, \delta j) \mid -1 \le \delta i \le 1, -N < \delta j < N\} \\
\Delta_{a,3} &= \{(\delta i, \delta j) \mid \delta i = 2, -N < \delta j \le 0\}
\end{aligned}$$

Our widening operator *detects* and *removes* constraints $c(\vec{N}) \ge 0$ depending on program parameters. The challenge is to detect them from several non-parametrized conflict sets, e.g.



(a) $\Delta_{blurx}(3)$

(b) $\Delta_{blurx}(4)$

Figure 5.2: Blur filter: Conflict polyhedron instances for `blurx`

$\Delta_{blurx}(3)$ and $\Delta_{blurx}(4)$. At first glance, the parametric constraints are the *moving* constraints from $N = 3$ to $N = 4$. The next problem is, how to make sure that non-moving constraints do not depend on a parameter? In general, this strongly depends on the selection of parameter values. For instance, when $\Delta(\vec{N}) = \{i \mid 0 \leq 2i \leq N\}$, $N = 0$ and $N = 1$ will give the same polyhedron $\{0, 1\}$. This choice of parameter does not succeed to *detect* the constraint $2i \leq N$. Multiple parameters might also be an issue: when $\Delta(M, N) = \{i \mid 0 \leq i \leq M - N\}$, $(M, N) = (1, 1)$ and $(M, N) = (2, 2)$ would give the same constraints.

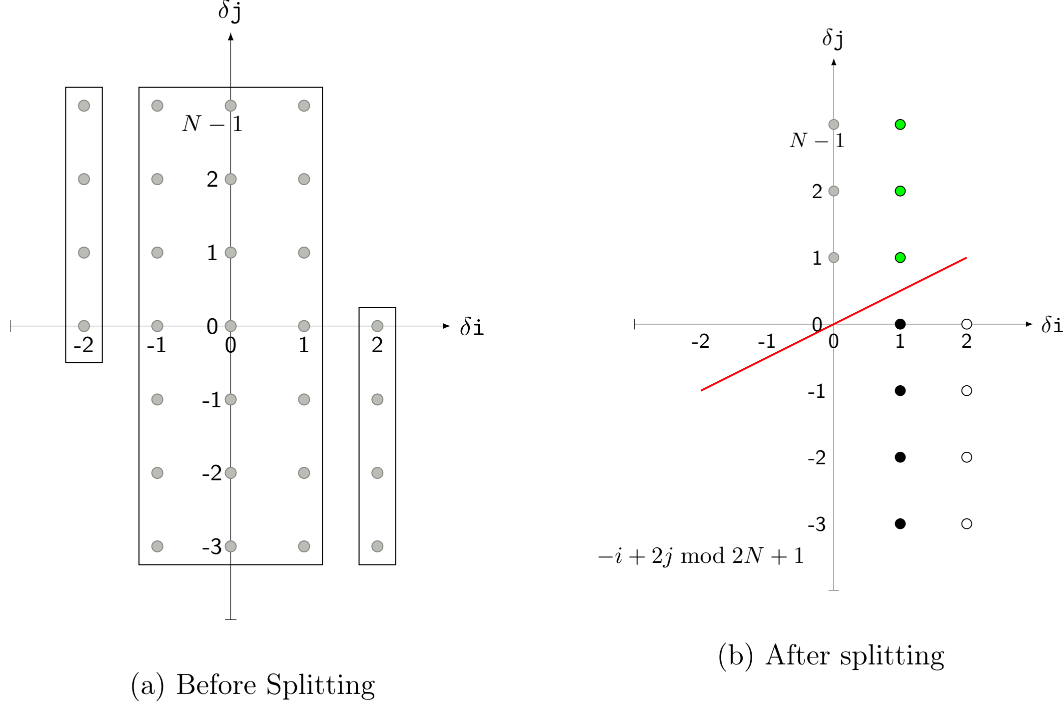
Hence, the tricky step is to build a set of parameter instances (*parameter selection*) $\mathcal{N}$ such that, for any constraint $c(\vec{N}) \geq 0$ of $\Delta(\vec{N})$ depending on $\vec{N}$, there exists $\vec{N}_0, \vec{N}_1 \in \mathcal{N}$ such that $c(\vec{N}_0) \neq c(\vec{N}_1)$. In other words, $\mathcal{N}$ must allow for *detecting* parametric constraints.

In general, the parameter selection $\mathcal{N}$ depends on the conflict set's *constraints shape*, which in turn depends on the program (iteration domains, array access function, scheduling function). We show that under suitable restrictions, we can guarantee the correctness of our widening operator. Finally, we show how to select the proper values of parameters.

Linear mappings We show how to compute near-optimal linear mappings σ with a limited amount of variables and constraints. On the Blur example, we obtain the mapping $\sigma_{blurx}(i, j) = -i + 2j \bmod 2N + 1$. Our method exploits the analogies *conflict sets* $\leftrightarrow$ *dependences*, *linear mappings* $\leftrightarrow$ *schedule*, *modulo* $\leftrightarrow$ *latency* and rephrase the ideas of affine scheduling for the inferrence of array mappings, in the same way as [14] on conflict *relations*. The main advantage

compared to [14] is a smaller number of constraints and variables.

The remainder of this chapter is structured as follows. Section 5.2 outlines the program restrictions considered in this chapter. Then, Section 5.3 proves that under these restrictions, the widening extrapolation is correct. Section 5.4 details our approach to compute array liveness with a such a widening operator. Section 5.5 presents an afficient algorithm to compute a linear mapping on the resulting conflict polyhedra. Finally, Section 5.6 presents an experimental validation of all our contributions.



(a) Before Splitting

(b) After splitting

Figure 5.3: Blur filter: Conflict polyhedron

5.2 Program Model

This section outlines the program restrictions of our approach: the program is assumed to be *totally unimodular* (Definition 5.2.3) and the dependences and the schedule are assumed to be *quasi-uniform* (Definitions 5.2.5 and 5.2.6).

Definition 5.2.1 (Unimodular, totally unimodular) *A matrix A is said to be* unimodular *if and only if* $|\det A| = 1$*. A is said to be* totally unimodular *if and only if any square non-singular submatrix of A is unimodular.*

Totally unimodular matrices play an important role in ILP to ensure that the vertices of a simplex has integer coordinates, hence that the solution of a rational linear program are the same

than the corresponding integer linear program. This notion allows us to get rid of the cases like $\{i \mid 2i \leq N\}$ – which are not suitable because the rational vertice $N/2$ is not integer for any N. A sufficient condition for a matrix A to be totally unimodular, which will be used in the proofs of section 5.3, is the following:

- All the elements are either unit or null, i.e. -1, 0 or 1.
- Each line has at most two non-null elements.
- For each line with two non-null elements, the non-null elements have an opposite sign.

Note that this is a specialization of Hoffman's sufficient condition. The actual condition covers more cases (where two line elements can share the same sign), but that specialization is sufficient for our purposes. We now inject this notion in the world of programs:

Definition 5.2.2 (Totally unimodular polyhedron) *A convex parametric polyhedron* $P(\vec{N}) = \{\vec{i} \mid A\vec{i} + B\vec{N} + \vec{c} \geq 0\}$ *is* totally unimodular *if and only if A is totally unimodular.*

Definition 5.2.3 (Totally unimodular program) *A program is totally unimodular if and only if all its iteration domains are totally unimodular.*

Totally unimodular programs are the focus of this chapter. Any static control program with for loops with step 1 or -1, whose bounds depends on parameters (any affine form) and counters (at most one counter with a coefficient 1) and whose conditions involved at most two counters with opposite signs (in the same equation side) is totally unimodular. Note that this definition is not very restrictive. For instance, all the Polybench kernels [51] are totally unimodular.

We assume the schedules to be *quasi-uniform*:

Definition 5.2.4 (Quasi-uniform matrix) *A matrix A is quasi-uniform if and only if each line has at most one non-null element and this non-null element is equal to 1.*

This includes compositions of projections, permutations and translations, which are very common in affine scheduling. However this excludes loop skewing. This restriction fits the sufficient conditions above for total unimodularity. We now inject this notion in the world of schedules:

Definition 5.2.5 (Quasi-uniform schedule) *A schedule* $\theta(\vec{i}) = A\vec{i} + B\vec{N} + \vec{c}$ *is said to be quasi-uniform if and only if A is quasi-uniform.*

On totally unimodular program, this means for loops must have a unit step.

Finally, we also expect direct dependences to be quasi-uniform.

Definition 5.2.6 (Quasi-uniform direct dependence) *A direct dependence piece* $h_S(\vec{i}) = A\vec{i} + B\vec{N} + \vec{c}\ \forall \vec{i} \in D$ *is said to be quasi-uniform if A is quasi-uniform. A direct dependence function is quasi-uniform if and only if all its pieces are quasi-uniform.*

To summarize, we expect the *program* to be totally unimodular, and both *schedules* and *direct dependences* to be quasi-uniform. All the polybench kernels satisfy in these constraints, except *durbin* (array access `y[k-i-1]` results in non-quasi-uniform dependence), *ludcmp* (`for i` loop

with step -1) and *adi* (array access `p[i][N-3-j+1]` results in non-quasi-uniform dependence). The classical definition of loop tiling, however, does not fit these constraints. To apply our method to tiled programs, tiles should be outlined in functions, then the contraction should be applied to functions separately. This extension is not addressed in this chapter and is left for future work.

5.3 Correctness

Consider a parametrized polyhedron $\Delta_a(\vec{N}) = \{\vec{x} \mid A\vec{x} + B\vec{N} + \vec{c} \geq 0\}$. When instanciating the parameter with some $\vec{N}_0$ and $\vec{N}_0 + \vec{\delta}$, both polyhedron instances will keep the same linear part A. The only changing part will be the constant part, incremented by $B\delta$ in $\Delta_k(\vec{N}_0 + \vec{\delta})$:

$$\begin{array}{rcl} \Delta_a(\vec{N}_0) & = & \{\vec{x} \mid A\vec{x} + B\vec{N}_0 + \vec{c} \geq 0\} \\ \Delta_a(\vec{N}_0 + \vec{\delta}) & = & \{\vec{x} \mid A\vec{x} + B\vec{N}_0 + \vec{c} + B\vec{\delta} \geq 0\} \end{array}$$

In this section, we show that, when the program is totally unimodular and all dependences and schedule are quasi-uniform, all the liveness constraints Δ_a involving the i-th parameter $\vec{N}_i$ will have a different constant part, when instantiated on $\vec{N}_0$ and some $\vec{N}_0 + \vec{\delta}$. This will ensure the correctness of our widening operator.

Lemma 5.3.1 (Total unimodularity of liveness constraints) *If P is totally unimodular, dependences are quasi-uniform, and θ is quasi-uniform, then the conflict relation is a union of polyhedra $\bowtie(\vec{N}) = \bigcup_{i=1}^{d}\{\vec{x} \mid \exists \vec{y} : \ A_i \begin{pmatrix} \vec{x} \\ \vec{y} \end{pmatrix} + B_i\vec{N} + c_i \geq 0\}$ where each A_i is totally unimodular.*

Proof. By definition, we have:

$$\bowtie(\vec{N}) := \ \{\vec{i},\vec{j} \mid \exists \vec{i}_C, \vec{j}_C : \vec{i},\vec{j} \in D_P \wedge \vec{i}_C, \vec{j}_C \in D_C : \vec{i} \prec_\theta \vec{j}_C \wedge \vec{j} \prec_\theta \vec{i}_C \wedge h_{PC}(\vec{i}_C) = \vec{i} \wedge h_{PC}(\vec{j}_C) = \vec{j}\}$$

This union comes from the expansion of $\prec_\theta$ in the ordering constraints:

$$\bowtie(\vec{N}) := \bigcup_{k,\ell} \ \{\vec{i},\vec{j} \mid \exists \vec{i}_C, \vec{j}_C : \vec{i},\vec{j} \in D_P \wedge \vec{i}_C, \vec{j}_C \in D_C : \vec{i} \prec_\theta^k \vec{j}_C \wedge \vec{j} \prec_\theta^\ell \vec{i}_C \wedge h_{PC}(\vec{i}_C) = \vec{i} \wedge h_{PC}(\vec{j}_C) = \vec{j}\}$$

From now, consider a term $\bowtie_{k,\ell}$ in this union.

Assuming:

- $D_P = \{\vec{i} \mid A_P\vec{i} + B_P\vec{N} + \vec{c}_P \geq 0\}$
- $D_C = \{\vec{i} \mid A_C\vec{i} + B_C\vec{N} + \vec{c}_C \geq 0\}$
- $\theta_P(\vec{i}) = T_P\vec{i} + U_P\vec{N} + \vec{v}_P$
- $\theta_C(\vec{i}) = T_C\vec{i} + U_C\vec{N} + \vec{v}_C$
- $h_{PC}(\vec{i}) = Q\vec{i} + r$

$\bowtie_{k,\ell}$ is defined by constraints $A\begin{pmatrix}\vec{x}\\ \vec{y}\end{pmatrix} + B\vec{N} + \vec{c} \geq 0$, $\vec{x} = (\vec{i}, \vec{j})$, $\vec{y} = (\vec{i}_C, \vec{j}_C)$, with (each line block correspond to a constraint):

$$A = \left(\begin{array}{cccc}
\vec{i} & \vec{j} & \vec{i}_C & \vec{j}_C \\
A_P & & & \\ \hline
 & A_P & & \\ \hline
 & & A_C & \\ \hline
 & & & A_C \\ \hline
-T_P & & & T_C \\
T'_P & & & -T'_C \\ \hline
 & -T_P & T_C & \\
 & T''_P & -T''_C & \\ \hline
-Id & & Q & \\
Id & & -Q & \\ \hline
 & -Id & & Q \\
 & Id & & -Q
\end{array}\right)$$

Where T'_P, T''_P are submatrices (k first lines) of T_P and T'_C, T''_C are submatrices (ℓ first lines) of T_C encoding the equalities in the first terms of the lexicographic ordering ($\prec_\theta^k$, $\prec_\theta^\ell$).

By hypothesis, all the elements are either -1, 0 or 1. Each line has at most two non-null elements, and for each line with two non-null elements, the non-null elements have an opposite sign. This is a sufficient condition for A to be totally unimodular. $\square$

We now demonstrate the core result of this section: under these hypothesis, the difference set $\Delta_a(\vec{N})$ have vertices defined by an affine mapping on program parameters $\vec{N}$. This ensures that when $\vec{N}$ is moving, the constraints depending on $\vec{N}$ are also moving (and then might be detected by the widening). We first show the result on the conflict relation polyhedron:

Theorem 5.3.2 (Affine vertices, relation) *Each vertice of $\bowtie(\vec{N})$ is defined by an integer affine mapping of $\vec{N}$.*

Proof. It is sufficient to show the property for $\bowtie_{k,\ell}(\vec{N})$ with arbitrary k, ℓ. From Lemma 5.3.1, $\bowtie_{k,\ell}(\vec{N})$ is defined by constraints: $A\begin{pmatrix}\vec{i}\\ \vec{j}\\ \vec{i}_C\\ \vec{j}_C\end{pmatrix} + B\vec{N} + \vec{c} \geq 0$, where A is totally unimodular and $\vec{i}_C$, $\vec{j}_C$ are existential variables. A vertice $\vec{y}$ is the solution of $A'\vec{y} + B'\vec{N} + \vec{c}' = 0$, such that $A'\vec{y} + B'\vec{N} + \vec{c}' \geq 0$ is a maximal subsystem of $A\vec{y} + B\vec{N} + \vec{c} \geq 0$ and A' is full row rank. Since A is totally unimodular, A' is totally unimodular and its columns (and y) can always be permuted so $A' = [U\ V]$, with U unimodular. Writing $\vec{y} = \begin{pmatrix}\vec{y}'\\ 0\end{pmatrix}$ (where the 0s cover the columns of V), we have: $U\vec{y}' + B'\vec{N} + \vec{c}' = 0$. Hence $y' = U^{-1}(-B'\vec{N} - \vec{c}')$. The vertice is then: $\vec{y} = \begin{pmatrix}U^{-1}(-B'\vec{N} - \vec{c}')\\ 0\end{pmatrix}$. Since U is unimodular, so is U^{-1}. Hence U^{-1} has integer coefficients,

hence the vertice is defined by the integer affine mapping: $\mu : \vec{N} \mapsto \begin{pmatrix} U^{-1}(-B'\vec{N} - \vec{c}') \\ 0 \end{pmatrix}$. The vertices of $\bowtie_{k,\ell} (\vec{N})$ are obtained by keeping only the $(\vec{i}, \vec{j})$ coordinates, hence a subset of lines of $\mu(\vec{N})$, on which the result holds.

Finally, we can demonstrate the main result:

Theorem 5.3.3 (Affine vertices, delta) *Each vertice of $\Delta(\vec{N})$ is defined by an integer affine mapping of $\vec{N}$.*

Proof. The conflict polyhedron $\Delta(\vec{N})$ is the *image* of each $(\vec{i}, \vec{j}) \in \bowtie$ through the *linear mapping* $\varphi(\vec{i}, \vec{j}) = \vec{i} - \vec{j}$:

$$\Delta(\vec{N}) = \varphi(\bowtie (\vec{N})) = \bigcup_{k,\ell} \varphi(\bowtie_{k,\ell} (\vec{N}))$$

By convexity and linearity, the vertices of $\Delta(\vec{N})$ are images through φ of (a subset of) vertices from $\bowtie (\vec{N})$. From theorem 5.3.2, each vertice μ of $\bowtie (\vec{N})$ is an affine integer mapping of $\vec{N}$, so are vertices $\varphi \circ \mu$ of $\Delta(\vec{N})$. □

It follows that, if a vertice $\mu(\vec{N})$ has a coordinate $D_i \cdot \vec{N} + \vec{d_i}$ depending on some $\vec{N_j}$ $(D_{ij} \neq 0)$, choosing a new parameter with the same coordinates – except for $\vec{N_i}$, replaced by $\vec{N_i} + 1$ – would cause the vertice to shift, hence all its neighboring faces. This way, that shift will cause all the constraints depending on $\vec{N_i}$ to change. As discussed at the beginning of this section, the change can only be the constant parts of the constraints. This is summarized in the following corollary:

Corollary 5.3.4 (Strict monotonicity) *Let $\vec{\delta_i} = (0, \dots 0, 1, 0 \dots 0)$ with $|\vec{\delta_i}| = |\vec{N}|$ and where 1 is in the ith coordinate. Then, the constraints $\Delta(\vec{N})$ involving N_i occurs with a different constant part in $\Delta(\vec{N} + \vec{\delta_i})$.*

5.4 Liveness Extrapolation by Widening

This Section presents our array liveness algorithm using the widening apparatus. Section 5.4.1 presents our algorithm to select the parameter instances. Then, Section 5.4.2 shows how to use the NLR algorithm [39] to retrieve the affine constraints for the different conflict set instances obtained. Finally, Section 5.4 presents our widening algorithm for the extrapolation. A general narrowing algorithm is also presented to bound the obtained conflict set.

5.4.1 Parameter Selection

We show how to select the first parameter instance, and how to derive the other required parameter instances thanks to Corollary 5.3.4.

Selecting the First Parameter We need to select value of parameters such that critical dependences hold, as for canonical array contraction. For this purpose, it is sufficient to select a value of $\vec{N}_0$ so that each dependence edge has *at least one instance* using algorithm 10. For each dependence (represented by a polyhedral set of instances $\Delta_{S,T}$), we retrieve the parameter domain such that at least one instance exists (line 4). Finally, Φ contains the set of parameters such that at least one instance of each dependence exists. It remains to output the smaller one (line 6).

Algorithm 10: GETPARAMETERS

Data: Arrays to contract $\mathcal{A}$, Direct dependence graph $\mathcal{G}$
Result: Parameter instance $\vec{N}_0$

```
1 begin
2 |   Φ := Universe
3 |   foreach dependence edge Δ_{S,T} ∈ G solved through an array of A do
4 |   |   Φ := Φ ∩ project(Δ_{S,T}, N)
5 |   end
6 |   return min_≪ Φ
7 end
```

Selecting the remaining parameters Following Corollary 5.3.4, constraints depending on $\vec{N}_i$ will have different constant parts from $\Delta_a(\vec{N}_0)$ to $\Delta_a(\vec{N}_0 + \vec{\delta}_i)$. Hence, the set of parameters $\mathcal{N}$ required to make the extrapolation is:

$$\mathcal{N} = \{\vec{N}_0\} \cup \{\vec{N}_0 + \vec{\delta}_i \mid i \in [\![1, |\vec{N}_0|]\!]\}$$

Example (cont'd) The dependence graph restricted to `blurx` accesses has 3 edges, from the write of `blurx`$[i][j]$ by some $\langle P, i, j\rangle$ to each of the 3 reads by $\langle C, i', j'\rangle$: `blurx`$[i'][j']$ (edge 1), `blurx`$[i'-1][j']$ (edge 2), `blurx`$['i-2][j']$ (edge 3). All the 3 edges occurs when $N-1$ is at least 2 from $\langle P, 0, 0\rangle$. Hence $\Phi = N \geq 3$ and $N_0 = 3$. The set of parameters is then $\mathcal{N} = \{3, 4\}$.

5.4.2 Infer Polyhedral Constraints from Traces

For each parameter set $\vec{N} \in \mathcal{N}$, we generate the program's execution trace, then we compute a liveness analysis from which we deduce a conflict set $\Delta_a(\vec{N})$ for each array a, as a finite set of integer vectors. This is similar to our method in the previous chapter.

We use the NLR algorithm [39] to retrieve polyhedral constraints from $\Delta_a(\vec{N})$. We simply interpret $\Delta_a(\vec{N})$ as an execution trace $\mathcal{T}_a(\vec{N})$, where the points are ordered with the lexicographic ordering. From that, NLR finds an *equivalent polyhedral program* generating $\mathcal{T}_a(\vec{N})$, where each statement S contains an affine function u_S computing the trace entries. We modified NLR to retrieve the polyhedral constraints of D_S, the iteration domain of S. Then, we retrieved the corresponding polyhedral domain with a simple projection:

$$P_S := \text{project}(\{(\vec{i}, \vec{j}) \mid \vec{i} \in D_S,\ \vec{j} = u_S(\vec{i})\},\ \vec{j})$$

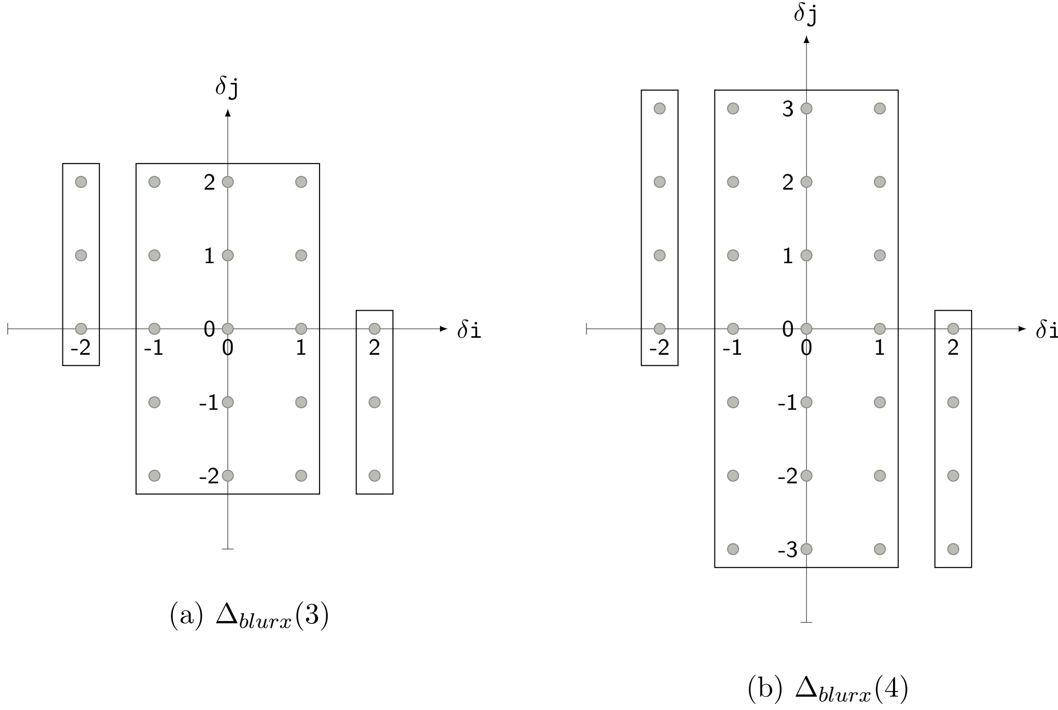
Finally the polyhedral constraints expressing $\Delta_a(\vec{N})$ are:

$$\Delta_a(\vec{N}) = \bigcup_{S \text{ statement}} P_S$$

Example (cont'd) The trace liveness analysis ends up with the set of points $\Delta_{blurx}(3)$ and $\Delta_{blurx}(4)$ depicted in Figure 5.4. NLR rephrases $\Delta_{blurx}(3)$ and $\Delta_{blurx}(4)$ as a program with three statements, each statement iterating on a subset framed on the Figure 5.4. From that, we infer: $\Delta_{blurx}(3) = \Delta_{blurx,1}(3) \cup \Delta_{blurx,2}(3) \cup \Delta_{blurx,2}(3)$ and $\Delta_{blurx}(4) = \Delta_{blurx,1}(4) \cup \Delta_{blurx,2}(4) \cup \Delta_{blurx,2}(4)$ with:

$$\begin{aligned}
\Delta_{blurx,1}(3) &= \{(\delta i, \delta j) \mid \delta i = -2, 0 \leq \delta j < 3\} \\
\Delta_{blurx,1}(4) &= \{(\delta i, \delta j) \mid \delta i = -2, 0 \leq \delta j < 4\} \\
\Delta_{blurx,2}(3) &= \{(\delta i, \delta j) \mid -1 \leq \delta i \leq 1, -3 < \delta j < 3\} \\
\Delta_{blurx,2}(4) &= \{(\delta i, \delta j) \mid -1 \leq \delta i \leq 1, -4 < \delta j < 4\} \\
\Delta_{blurx,3}(3) &= \{(\delta i, \delta j) \mid \delta i = 2, -3 < \delta j \leq 0\} \\
\Delta_{blurx,3}(4) &= \{(\delta i, \delta j) \mid \delta i = 2, -4 < \delta j \leq 0\}
\end{aligned}$$

Note that $\Delta_{blurx,\ell}(3) = \{\vec{\delta} \mid A_\ell \vec{\delta} + \vec{b}(3) \geq 0\}$ and $\Delta_{blurx,\ell}(4) = \{\vec{\delta} \mid A_\ell \vec{\delta} + \vec{b}(4) \geq 0\}$ for some matrix A_ℓ, $\ell = 1, 2, 3$. The only moving part is the constant vector.



(a) $\Delta_{blurx}(3)$

(b) $\Delta_{blurx}(4)$

Figure 5.4: Blur filter: Conflict polyhedron instances for `blurx` and NLR splitting

Note that the order in which the elements of $\Delta_a(\vec{N})$ are fed to the NLR algorithm is crucial, because of its greedy nature. We found that the lexicographic ordering leads to expected results.

How to derive an ordering which allows to retrieve a minimal set of convex polyhedra is an open problem.

Our widening method (described in the next section) expects the polyhedral constraints of the different instances $\Delta_a(\vec{N})$ to *only differ on the constant part* for any $\vec{N} \in \mathcal{N}$:

$$\Delta_{a,\ell}(\vec{N}) = \{\vec{\delta} \mid A_\ell \vec{\delta} + b_\ell(\vec{N}) \geq 0\} \quad \text{for any part } \ell$$

As discussed in the beginning of Section 5.3, this should always be the case. However, it can happen that NLR does not succeed to find it. A possible issue is that NLR retrieves a loop starting from three consecutive iterations. If N is too small, it is possible that that the loop detection fails, which would end up with a big union of single point polyhedra, whereas the next parameter $N+1$ would enable loop detection, giving proper polyhedra. To avoid that situation, we heuristically add an NLR *increment* to the value of $\vec{N}$ found by the parameter selection algorithm 10 to enforce loop detection. We found that an increment of 1 was sufficient on our benchmarks. In particular, it covers the case of *reduction dependences*, whose distance is 1 and which would led to select parameters covering only two iterations otherwise. Hence the actual $\mathcal{N}$ used on our experiments for the Blur filter kernel is $\mathcal{N} = \{4, 5\}$ instead of $\{3, 4\}$.

5.4.3 Widening Algorithm

For each array a, we obtain a set of polyhedral constraints, that we assume to only differ on the constant part:

$$\Delta_a(\vec{N}) = \bigcup_{\ell \in \mathcal{I}} \{\vec{\delta} \mid A_\ell \vec{\delta} + b_\ell(\vec{N}) \geq 0\} \quad \text{for any part } \ell$$

Where $\mathcal{I}$ is some finite indexing set. If it is not the case, our method simply fails.

The purpose of the widening is to find an *extrapolation* of the conflict polyhedra $\Delta_a(\vec{N})$, for $\vec{N} \in \mathcal{N} = \{\vec{N}_1, \ldots, \vec{N}_k\}$, denoted by $\nabla\left(\Delta_a(\vec{N}_1), \ldots, \Delta_a(\vec{N}_k)\right)$, which is *correct* for any parameter:

$$\Delta_a(\vec{N}) \subseteq \nabla\left(\Delta_a(\vec{N}_1), \ldots, \Delta_a(\vec{N}_k)\right) \quad \text{for any parameter } \vec{N} \tag{5.1}$$

This way, $\nabla\left(\Delta_a(\vec{N}_1), \ldots, \Delta_a(\vec{N}_k)\right)$ could be used as a correct conflict set for array contraction of a.

The main idea is to *remove constraints depending on parameters.* However, we need to operate the constraints in such a way we can detect the presence of a parameter. From corollary 5.3.4, if some parameter is involved in a constraint $A_\ell[i] \cdot \vec{\delta} + c_\ell(\vec{N})[i] \geq 0$ (i-th constraint), then $\exists \vec{N}, \vec{N}' \in \mathcal{N}$, $\vec{N} \neq \vec{N}'$ such that $c_\ell(\vec{N})[i] \neq c_\ell(\vec{N}')[i]$. Hence, *it is sufficient to remove the constraints whose constant part has changed.*

This is achieved by Algorithm 11: for each sub-polyhedra ℓ, the i-th constraint is kept only if the constant part is the same for all the parameters of $\mathcal{N}$ (line 9).

Algorithm 11: WIDENING

Data: Conflict polyhedra $\Delta(\vec{N}_1), \ldots, \Delta(\vec{N}_k)$

Result: Widened polyhedron $\nabla\left(\Delta(\vec{N}_1), \ldots, \Delta(\vec{N}_k)\right)$

1 **begin**
2 Let $\Delta(\vec{N}_p) = \bigcup_{\ell \in \mathcal{I}} \{\vec{\delta} \mid A_\ell \vec{\delta} + \vec{b}_\ell(\vec{N}_p)\}$, $1 \leq p \leq k$
3 **if** $\Delta(\vec{N}_p)$ *cannot be written this way* **then**
4 FAIL
5 **end**
6 Let s_ℓ be the number of lines of A_ℓ
7 **foreach** $\ell \in \mathcal{I}$, $1 \leq i \leq s_\ell$ **do**
8 **if** $\forall p, q,\ 1 \leq p \neq q \leq k$: $\vec{b}_\ell(\vec{N}_p)[i] = \vec{b}_\ell(\vec{N}_q)[i]$ **then**
9 Add constraint $A_\ell[i] \cdot \vec{\delta} + \vec{b}_\ell(\vec{N}_p)[i] \geq 0$ to Φ_ℓ
10 **end**
11 **end**
12 **return** $\bigcup_{\ell \in \mathcal{I}} \Phi_\ell$
13 **end**

Example (cont'd) We obtain the polyhedra $\Delta_{blurx}(3) = \Delta_{blurx,1}(3) \cup \Delta_{blurx,2}(3) \cup \Delta_{blurx,2}(3)$ and $\Delta_{blurx}(4) = \Delta_{blurx,1}(4) \cup \Delta_{blurx,2}(4) \cup \Delta_{blurx,2}(4)$ with:

$$\begin{aligned}
\Delta_{blurx,1}(3) &= \{(\delta i, \delta j) \mid \delta i = -2, 0 \leq \delta j < 4\} \\
\Delta_{blurx,1}(4) &= \{(\delta i, \delta j) \mid \delta i = -2, 0 \leq \delta j < 5\} \\
\Delta_{blurx,2}(3) &= \{(\delta i, \delta j) \mid -1 \leq \delta i \leq 1, -4 < \delta j < 4\} \\
\Delta_{blurx,2}(4) &= \{(\delta i, \delta j) \mid -1 \leq \delta i \leq 1, -5 < \delta j < 5\} \\
\Delta_{blurx,3}(3) &= \{(\delta i, \delta j) \mid \delta i = 2, -4 < \delta j \leq 0\} \\
\Delta_{blurx,3}(4) &= \{(\delta i, \delta j) \mid \delta i = 2, -5 < \delta j \leq 0\}
\end{aligned}$$

Recall that $\Delta_{blurx,\ell}(3) = \{\vec{\delta} \mid A_\ell \vec{\delta} + \vec{b}_\ell(3) \geq 0\}$ and $\Delta_{blurx,\ell}(4) = \{\vec{\delta} \mid A_\ell \vec{\delta} + \vec{b}_\ell(4) \geq 0\}$ for some matrix A_ℓ, $\ell \in \mathcal{I} = \{1, 2, 3\}$. The only changing part is the constant vector. Hence, constraints for $N = 3$ and $N = 4$ might be paired together and our widening algorithm applies. Then, the widening removes the changing constraints, which results in the extrapolation $\nabla(\Delta_{blurx}(3), \Delta_{blurx}(4)) = \hat{\Delta}_{blurx,1} \cup \hat{\Delta}_{blurx,2} \cup \hat{\Delta}_{blurx,3}$ where:

$$\begin{aligned}
\hat{\Delta}_{blurx,1} &= \{(\delta i, \delta j) \mid \delta i = -2, 0 \leq \delta j\} \\
\hat{\Delta}_{blurx,2} &= \{(\delta i, \delta j) \mid -1 \leq \delta i \leq 1\} \\
\hat{\Delta}_{blurx,3} &= \{(\delta i, \delta j) \mid \delta i = 2, \delta j \leq 0\}
\end{aligned}$$

5.4.4 Narrowing

The widening tends to make the polyhedra open, hence to create conflict vectors $\vec{\delta}$ whose size can be of any size. Array contraction requires conflict vectors with bounded sizes. As it is, $\hat{\Delta}_{blurx}$ leads to an infinite modulo in the j direction.

Remark that an array a has a fixed size $M_1 \times \ldots \times M_d$, where d is the dimension of a. Hence, any cell $(i_1, \ldots i_d)$ of a is such that $0 \leq i_1, \ldots, i_d, i_1 < M_1, \ldots, i_d < M_d$. Therefore, any conflict vector is bounded, and we have:

$$\mathcal{C}_a := -M_r < \delta i_r < M_r \quad \forall 0 \leq r \leq d$$

This way, $\hat{\Delta}_a$ shall be bounded by intersecting with $\mathcal{C}_a$. By analogy with abstract interpretation, we call that process *narrowing*. In the following, *widening* will take on a broader sense and refer to *widening followed by narrowing*.

Example (cont'd) Remark that `blurx` has a bounded size $N \times N$. Hence, any cell (i, j) of `blurx` is such that $0 \leq i, j < N$. Therefore, $-N < \delta i, \delta j < N$. Hence, the conflict polyhedron *narrows* the conflict polyhedron. In consequence, the constraints of $\hat{\Delta}_{blurx}$ are intersected with $\mathcal{C}_{blurx} = \{(\delta i, \delta j) \mid -N < \delta i < N, -N < \delta j < N\}$ and we update the conflict polyhedron $\hat{\Delta}_a = \hat{\Delta}_{a,1} \cup \hat{\Delta}_{a,2} \cup \hat{\Delta}_{a,3}$, where:

$$\begin{aligned}
\hat{\Delta}_{a,1} &= \{(\delta i, \delta j) \mid \delta i = -2, 0 \leq \delta j < N\} \\
\hat{\Delta}_{a,2} &= \{(\delta i, \delta j) \mid -1 \leq \delta i \leq 1, -N < \delta j < N\} \\
\hat{\Delta}_{a,3} &= \{(\delta i, \delta j) \mid \delta i = 2, -N < \delta j \leq 0\}
\end{aligned}$$

5.5 Linear Allocation

This section describes our algorithm to derive a linear allocation $\sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ from a difference set Δ_a. We built on the ideas of the SMO algorithm [14], that we rephrase to operate on difference set Δ_a rather than a conflict relation $\bowtie$. Experimentally, we demonstrate that it leads to simpler constraints (as Δ sets have about two times less constraints than conflict relations). Hence, we believe that our approach is more likely to scale.

The algorithm proceeds similarly to greedy affine scheduling [31]. We compute each line $\vec{\tau}_k$ of M and each component $\vec{b}_k$ of $\vec{b}$ iteratively, starting from the first one $k = 1$. Then, resolved conflicts are removed and we iterate on the next dimension. Each line is computed with an ILP encoding the *correctness* and the *efficiency* of the mapping. Section 5.5.1 explains the construction of the correctness constraints (conflict satisfaction). Section 5.5.2 explains how to keep unsolved constraints for computing the subsequent dimensions. Finally, Section 5.5.3 explains how to encode the efficiency constraints (reducing the footprint).

5.5.1 Correctness

General Formulation

The mapping σ must be correct: any pair of conflicting array cells $a(\vec{i}) \bowtie a(\vec{j})$ must be mapped to different locations: $\sigma(\vec{i}) \neq \sigma(\vec{j})$. Let $\vec{\delta} \in \Delta_a \setminus \{0\}$. By definition, there exists some array index $\vec{i}$ such that $a(\vec{i}) \bowtie a(\vec{i} + \vec{\delta})$. Since $\vec{i} \neq \vec{i} + \vec{\delta}$ ($\vec{\delta} \neq 0$), we must have $\sigma(\vec{i}) \neq \sigma(\vec{i} + \vec{\delta})$. This means that for some k we must have:

$$\vec{\tau}_k \cdot \vec{i} \bmod \vec{b}_k \neq \vec{\tau}_k \cdot (\vec{i} + \vec{\delta}) \bmod \vec{b}_k \qquad \forall \vec{\delta} \in \Delta_a \setminus \{0\} \tag{5.2}$$

We will first focus on finding a $\vec{\tau}_k$ such that $\vec{\tau}_k \cdot \vec{i} \neq \vec{\tau}_k \cdot (\vec{i} + \vec{\delta})$, then we will show how to deduce the modulo $\vec{b}_k$ in Section 5.5.3. By linearity, this might be rephrased as $\vec{\tau}_k \cdot \vec{\delta} \neq 0$, and we say that $\vec{\tau}_k$ *satisfies* $\vec{\delta}$. Hence the following *correctness constraint*:

$$\textsc{correct}(\vec{\tau}_k) : \qquad \text{for all } \vec{\delta} \in \Delta_a \setminus \{0\} : \vec{\tau}_k \cdot \vec{\delta} \neq 0$$

Conflict Set Partitioning

Depending of the shape of Δ_a and its dimension, it is possible that no solution exists: no $\vec{\tau}_k$ can satisfy all the $\vec{\delta} \in \Delta_a$ at once. We choose to *greedily* satisfy *as much constraints as possible* and to delay the satisfaction of unsatisfied $\vec{\delta}$ to subsequent dimensions, *just like greedy affine scheduling*. Hence the strategy to partition Δ_a into sub-polyhedra $\Delta_a = \cup_{\ell \in \mathcal{I}} \Delta_{a,\ell}$, that hold the satisfaction granularity – either $\vec{\tau}_k$ satisfies *all* $\vec{\delta}$ of $\Delta_{a,\ell}$ for some $\ell \in \mathcal{I}$, or not. How to make such a partitionning is still an open problem [14]. The partionning of the conflict relation $\bowtie$ induced by the lexicographic ordering constraints between two conflicting liveness intervals $\bowtie = \cup_{k,\ell} \bowtie_{k,\ell}$ seems to provide a solution [14], but needs to be analyzed still. In this thesis, we will refine the partioning induced by the NLR reconstruction : $(\Delta_{a,\ell})_{\ell \in \mathcal{I}}$. This leads to consider the conjunction of the following constraints for $\ell \in \mathcal{I}$:

$$\textsc{correct}(\vec{\tau}_k, \ell) : \qquad \forall \vec{\delta} \in \Delta_{a,\ell} \setminus \{0\} : \vec{\tau}_k \cdot \vec{\delta} \neq 0 \tag{5.3}$$

Each $\textsc{correct}(\vec{\tau}_k, \ell)$ will have its own decision variable, equal to 1 when satisfied and 0 when not satisfied. This is described in the next section.

Partitionning strategy We need to write the constraints $\textsc{correct}(\vec{\tau}_k, \ell)$ for each $\ell \in \mathcal{I}$, hence the trade-off:

- Find a partitionning of Δ_a with a *fine enough granularity* to find an interesting mapping. At worst, the trivial partition of singletons $\Delta_a = \cup_{\vec{\delta} \in \Delta_a} \{\vec{\delta}\}$ (hence $\mathcal{I} = \Delta_a$, $\Delta_{a,\vec{\delta}} = \{\vec{\delta}\}$) will work for non-parameterized Δ_a.

- Avoid too many partitions, which would increase the complexity of the final integer linear program (there will be one constraint set per partition). Naturally, the trivial partition would have a prohibitive cost (as many constraints $\textsc{correct}(\vec{\tau}_k, \ell)$) as $\operatorname{card} \mathcal{I} = \operatorname{card} \Delta_a$)!

Since the conflict relation is symmetric, $a[\vec{i}] \bowtie a[\vec{j}]$ iff $a[\vec{j}] \bowtie a[\vec{i}]$. By definition of Δ_a, this entails that: $\vec{\delta} = \vec{i} - \vec{j} \in \Delta_a \iff -\vec{\delta} = \vec{j} - \vec{i} \in \Delta_a$. Hence Δ_a is 0-symmetric. In particular, if $\vec{\tau}_k$ satisfies $\vec{\delta}$ ($\vec{\tau}_k \cdot \vec{\delta} \neq 0$), then $\vec{\tau}_k$ also satisfies $-\vec{\delta}$, as $\vec{\tau}_k \cdot -\vec{\delta} = -(\vec{\tau}_k \cdot \vec{\delta}) \neq 0$.

Hence, we will just consider the “positive half” of Δ_a, using the lexicographic order:

$$\Delta_a^+ = \Delta_a \cap \{\vec{\delta} \mid \vec{\delta} \gg 0\}$$

Note that 0 is excluded from Δ_a^+, as the correctness constraints are defined over $\Delta_a \setminus \{0\}$ (Eq. 5.5.1). Solving $\vec{\tau}$ over Δ_a^+ is sufficient to ensure the correctness, as stated by the following theorem.

Theorem 5.5.1 *If $\vec{\tau}$ solves any $\vec{\delta} \in \Delta_a^+$, Then $\vec{\tau}$ solves any $\vec{\delta} \in \Delta_a \setminus \{0\}$.*

Proof. First, note that $\ll$ is a total order, hence for any $\vec{\delta} \neq 0$, either $\vec{\delta} \gg 0$ or $\vec{\delta} \ll 0$. Hence, with $\Delta_a^- = \Delta_a \cap \{\vec{\delta} \mid \vec{\delta} \ll 0\}$, $\Delta_a^- \cup \Delta_a^+$ is a partition of $\Delta_a \setminus \{0\}$. By contradiction, consider a $\vec{\tau}$ solving any $\vec{\delta} \in \Delta_a^+$ but that does not solve some $\vec{\delta_0} \in \Delta_a^-$ ($\vec{\tau} \cdot \vec{\delta_0} = 0$). Hence $0 = -(\vec{\tau} \cdot \vec{\delta_0}) = \vec{\tau} \cdot -\vec{\delta_0}$: $\vec{\tau}$ does not solve $-\vec{\delta_0} \in \Delta_a^+$, which contradicts the hypothesis.

However, this splitting may not be sufficient. Consider the conflict polyhedron Δ_{blurx} depicted on Figure 5.3(a). This will partition Δ_{blurx}^+ into three subsets with $\delta i = 0, 1, 2$. The subset $E = \{(\delta i, \delta j) \mid \delta i = 1, -N < \delta j < N\}$ will necessarily have some $\vec{\delta}$ with $\vec{\tau} \cdot \vec{\delta} > 0$ *and* some $\vec{\delta}$ with $\vec{\tau} \cdot \vec{\delta} < 0$. This does *not* suit our encoding of $\vec{\tau} \cdot \vec{\delta} \neq 0 \; \forall \vec{\delta} \in E$ by either $\vec{\tau} \cdot \vec{\delta} > 0 \; \forall \vec{\delta} \in E$ or $\vec{\tau} \cdot \vec{\delta} < 0 \; \forall \vec{\delta} \in E$, as described in next section.

Hence, we heuristically choose to refine the partitionning by intersecting with quadrants $\mathcal{Q}$ with all the lexicographically positive combinations of constraints $\vec{\delta_i} > 0$ and $\vec{\delta_i} \leq 0$ for all indices i. We remark that experimentally, this enables interesting solutions.

Example (cont'd) After splitting, by intersecting with $\Delta^+ = \{\vec{\delta} = (\Delta i, \delta j) \mid \vec{\delta} \gg 0\}$ and the lexicographically positive quadrants $\delta i > 0, \delta j > 0$ and $\delta i > 0, \delta j \leq 0$, we obtain the polyhedra $\tilde{\Delta}_{blurx} = \tilde{\Delta}_{blurx,1} \cup \tilde{\Delta}_{blurx,2} \cup \tilde{\Delta}_{blurx,3} \cup \tilde{\Delta}_{blurx,4}$, where:

$$\begin{aligned}
\tilde{\Delta}_{blurx,1} &= \{(\delta i, \delta j) \mid \delta i = 0, 1 \leq \delta j < N\} \\
\tilde{\Delta}_{blurx,2} &= \{(\delta i, \delta j) \mid \delta i = 1, 1 \leq \delta j < N\} \\
\tilde{\Delta}_{blurx,3} &= \{(\delta i, \delta j) \mid \delta i = 1, -N < \delta j \leq 0\} \\
\tilde{\Delta}_{blurx,4} &= \{(\delta i, \delta j) \mid \delta i = 2, -N < \delta j \leq 0\}
\end{aligned}$$

Encoding as an Integer Linear Program

The constraints CORRECT($\vec{\tau}_k, \ell$) cannot be directly encoded as an integer linear program, because of the quantification $\forall \vec{\delta} \in \Delta_{a,\ell}$ and the inequality (or disequality [63]) $\vec{\tau}_k \cdot \vec{\delta} \neq 0$. Hopefully, both problems have already been addressed in the polyhedral community. The quantification may be removed using the affine form of Farkas lemma, in the same way as for affine scheduling [32]; and the inequality $\vec{\tau}_k \cdot \vec{\delta} \neq 0$ may be rephrased by following the lines of [15]:

$$\vec{\tau}_k \cdot \vec{\delta} \neq 0 \Leftrightarrow 1 - (1 - \epsilon_{\ell,1})(\vec{c} \cdot \vec{N} + d + 1) \leq \vec{\tau}_k \cdot \vec{\delta} \leq -1 + (1 - \epsilon_{\ell,2})(\vec{c} \cdot \vec{N} + d + 1) \qquad (5.4)$$

This holds for $\vec{c}$ and d big enough. The variables $\epsilon_{\ell,1}, \epsilon_{\ell,2} \in \{0, 1\}$ make possible to express the three cases $\vec{\tau}_k \cdot \vec{\delta} = 0$, $\vec{\tau}_k \cdot \vec{\delta} \geq 1$ and $\vec{\tau}_k \cdot \vec{\delta} \leq 1$. While $\epsilon_{\ell,1} = 1$ implies the ≥ 1 case, $\epsilon_{\ell,2} = 1$ implies the ≤ 1 case. Also, $\epsilon_{\ell,1} = 0$ turns $1 - (1 - \epsilon_{\ell,1})(\vec{c} \cdot \vec{N} + d + 1) \leq \vec{\tau}_k \cdot \vec{\delta}$ to $-(\vec{c} \cdot \vec{N} + d) \leq \vec{\tau}_k \cdot \vec{\delta}$, which is always true providing $\vec{c}$ and d are big enough. Symmetrically, $\epsilon_{k,2} = 0$ makes the following constraint true (and so it can be disposed of): $\vec{\tau}_k \cdot \vec{\delta} \leq -1 + (1 - \epsilon_{\ell,2})(\vec{c} \cdot \vec{N} + d + 1)$. This is summarized in the following table:

$\epsilon_{\ell,1}$	$\epsilon_{\ell,2}$	Effect
0	0	True (no constraint)
1	0	$\vec{\tau}_k \cdot \vec{\delta} \geq 1$
0	1	$\vec{\tau}_k \cdot \vec{\delta} \leq -1$
1	1	False (cannot happen)

In particular, the constraints CORRECT$(\vec{\tau}_k, \ell)$ are solved if $\vec{\tau}_k \cdot \vec{\delta} \geq 1$ ($\epsilon_{\ell,1} = 1, \epsilon_{\ell,2} = 0$) or $\vec{\tau}_k \cdot \vec{\delta} \leq 1$ ($\epsilon_{\ell,1} = 0, \epsilon_{\ell,2} = 1$), hence if $\mu_\ell = \epsilon_{\ell,1} + \epsilon_{\ell,2} = 1$. Similarly, the constraints CORRECT$(\vec{\tau}_k, \ell)$ are *not* solved – neither $\vec{\tau}_k \cdot \vec{\delta} \leq -1$ nor $\vec{\tau}_k \cdot \vec{\delta} \geq -1$ for any $\vec{\delta}$ – when $\epsilon_{\ell,1} = \epsilon_{\ell,2} = 0$, hence $\mu_\ell = \epsilon_{\ell,1} + \epsilon_{\ell,2} = 0$. This way, μ_ℓ is a decision variable expressing if CORRECT$(\vec{\tau}_k, \ell)$ is solved. Note that $\mu_\ell = 0$ means that *there exists* $\vec{\delta} \in \Delta_{a,\ell}$ unsolved by $\vec{\tau}_k$. Some *other* $\vec{\delta} \in \Delta_{a,\ell}$ might be solved by $\vec{\tau}_k$. The final correctness constraints with μ_ℓ for each $\ell \in \mathcal{I}$ are denoted by CORRECT$(\vec{\tau}_k, \ell, \mu)$.

An objective function is then to maximize the partitions $\Delta_{i,\ell}$ solved, for $\ell \in \mathcal{I}$. Since $0 \leq \mu_\ell \leq 1$, an objective function to maximize is then $\nu = \sum_{\ell \in \mathcal{I}} \mu_\ell$. Note that ν is exactly the number of partitions solved by $\vec{\tau}$. When $\nu = \operatorname{card} \mathcal{I}$, $\vec{\tau}$ solves *all* the conflict vectors $\vec{\delta} \in \cup_{\ell \in \mathcal{I}} \Delta_{a,\ell}$. Otherwise, there remains unsolved conflicts $\vec{\delta}$ ($\vec{\tau} \cdot \vec{\delta} = 0$) which will have to be solved in the next dimensions, in the same way as for multidimensional scheduling.

5.5.2 Iterating on the Next Dimension

If $\vec{\tau}$ does not solve all the conflicts, there remains $\vec{\delta} \in \cup_{\ell \in \mathcal{I}} \Delta_{a,\ell}$ such that $\vec{\tau} \cdot \vec{\delta} = 0$. In other words, there exists conflicting array indices $\vec{i}$ and $\vec{i} + \vec{\delta}$ mapped to the same target array cell: $\vec{\tau} \cdot \vec{i} = \vec{\tau} \cdot (\vec{i} + \vec{\delta})$. Those unsolved conflict vectors $\vec{\delta}$ are gathered in the polyhedra $\Delta_{a,\ell}$ such that $\mu_\ell = 0$.

Like multidimensional affine scheduling, we focus on unsolved conflicts and we iterate the process to find the subsequent $\vec{\tau}$:

$$\begin{array}{l} \mathcal{I}' = \{\ell \in \mathcal{I} \mid \text{ s.t. } \mu_\ell = 0\} \\ \Delta'_{a,\ell} = \Delta_{a,\ell} \cap \{\vec{\delta} \mid \vec{\tau} \cdot \vec{\delta} = 0\} \text{ for each } \ell \in \mathcal{I}' \end{array}$$

The process stops when all the conflicts are solved: $\mathcal{I}' = \emptyset$.

5.5.3 Efficiency

The mapping $\sigma : \vec{i} \mapsto M\vec{i} \bmod \vec{b}$ must ensure a footprint $b_1 \ldots b_n$ as small as possible, hence the need to minimize the modulo vector $\vec{b}$. Since the lines $\vec{\tau}_k$ of M are computed iteratively, we will wrap in the same integer linear program the computation of $\vec{\tau}_k$ and the corresponding dimension b_k of $\vec{b}$ with the objective to minimize it.

Consider $\vec{\delta} \in \Delta_a \setminus \{0\}$. Simplifying Equation 5.2 by linearity, we deduce:

$$\vec{\tau}_k \cdot \vec{\delta} \bmod b_k \neq 0$$

It is *sufficient* to choose $b_k > |\vec{\tau}_k \cdot \vec{\delta}|$, the smallest one being $b_k := 1 + \max\{|\vec{\tau}_k \cdot \vec{\delta}|,\ \vec{\delta} \in \Delta_a\}$. Hence, to reduce the footprint, we need to *minimize the quantity* $|\vec{\tau}_k \cdot \vec{\delta}|$. Since $\vec{\delta} \in \Delta_a \setminus \{0\}$, and Δ_a is a polyhedron parametrized by $\vec{N}$, $|\vec{\tau}_k \cdot \vec{\delta}|$ must be bounded by some affine form of $\vec{N}$, $|\vec{\tau}_k \cdot \vec{\delta}| \leq \vec{e} \cdot \vec{N} + f$, similarly to latency constraints in affine scheduling:

$$\text{EFFICIENT}(\vec{\tau}_k, \ell, \vec{e}, f): \qquad -(\vec{e} \cdot \vec{N} + f) \leq \vec{\tau}_k \cdot \vec{\delta} \leq \vec{e} \cdot \vec{N} + f \qquad \text{for all } \vec{\delta} \in \Delta_{a,\ell} \tag{5.5}$$

It is then sufficient to *minimize lexicographically* the vector $(\vec{e}, f)$: first we minimize the parameter coefficient $\vec{e}$ to tend towards a constant modulo. Then, we minimize the constant part f.

Finally, the modulo is $b_k := 1 + \vec{e} \cdot \vec{N} + f$. Note that EFFICIENT$(\vec{\tau}_k, \ell, \vec{e}, f)$ is bound to $\Delta_{a,\ell}$. To be bound to Δ_a, we will consider the conjunctions of those constraints for $\ell \in \mathcal{I}$. As for the correctness constraints the universal quantification on $\Delta_{a,\ell}$ may be removed and turned to an integer linear program using the affine form of Farkas lemma [33, 3].

5.5.4 Algorithm

Algorithm 12 depicts our algorithm to compute the mapping $\sigma : \vec{i} \mapsto M\vec{i} \bmod \vec{b}$. Each line $\vec{\tau}_k$ of M and the corresponding modulo b_k are computed iteratively, starting from $k = 1$. At each iteration, we solve the integer linear program, as described above (line 4) which conssist of the conjunction of the correctness and efficiency constraints for each partition $\Delta_{a,\ell}$, $\ell \in \mathcal{I}$ of the difference set Δ_a. From the correctness constraints, we get the decisions variables μ_ℓ s.t. $\mu_\ell = 1$ if and only if $\vec{\tau}_k$ solves $\Delta_{a,\ell}$, and 0 otherwise. From the efficiency constraints, we have the coefficients $\vec{e}$ and f of the affine form bounding $|\vec{\tau}_k \cdot \vec{\delta}|$ for any $\vec{\delta} \in \cup_{\ell \in \mathcal{I}} \Delta_{a,\ell}$. As explained earlier, this yields the modulo $b_k := 1 + \vec{e} \cdot \vec{N} + f$. The objective function is *first* to greedily solve as much partition $\Delta_{a,\ell}$ as possible, hence to maximize $\sum_{\ell \in \mathcal{I}} \mu_\ell$, or to minimize $-\sum_{\ell \in \mathcal{I}} \mu_\ell$. *Second*, to minimize the modulo; by first minimizing the coefficients $\vec{e}$ of $\vec{N}$ and second by minimizing the constant part f. Once the integer linear program is solved, we obtain $\vec{\tau}_k$ and we retrieve b_k (line 6). Finally, we keep only the unsolved conflicts in Δ_a for the next iteration (line 8). If all the conflicts are solved ($\mathcal{I} = \emptyset$, no more partition of Δ_a to consider), the algorithm returns the solution (line 11).

Algorithm 12: LINEARALLOCATION

Data: Difference set $\Delta_a = \cup_{\ell \in \mathcal{I}} \Delta_{a,\ell}$

Result: Linear mapping $\sigma : \vec{i} \mapsto M\vec{i} \bmod \vec{b}$

```
1  begin
2      k := 1
3      repeat
4          min_≪(−Σ_{ℓ∈I} μ_ℓ, e, f)
           s.t.
           { ∧_{ℓ∈I} CORRECT(τ_k, ℓ, μ)
           { ∧_{ℓ∈I} EFFICIENT(τ_k, ℓ, e, f)
5          M_k := τ_k
6          b_k := 1 + e · N + f
7          I := {ℓ ∈ I | μ_ℓ = 0}
8          Δ_{a,ℓ} := Δ_{a,ℓ} ∩ {τ | τ · δ = 0} for each ℓ ∈ I
9          k := k + 1
10     until I = ∅
11     return σ : i ↦ Mi mod b
12 end
```

Example (cont'd) The main difficulty is to get a proper encoding of correctness and efficiency constraints in a single integer linear program. This is done thanks to the FKCC tool, which

provides an easy-to-use DSL to specify such constraints [3].

Correctness Figure 5.5 depicts the first half of the FKCC program, which encodes the correctness constraints. First, we declare decision variables (line 1), and define objective variables (line 2), conflict sets (lines 5-8) and $\vec{\tau}_k$ (encoded as an affine form H such that $H(\vec{\delta}) = \vec{\tau}_k \cdot \vec{\delta} + c$, the constant being later forced to zero, line 11). The first difficulty is to deal with Eq (5.5.1). Consider the first inequality (the same reasonning applies for the second inequality):

$$1 - (1 - \epsilon_{\ell,1})(\vec{c} \cdot \vec{N} + d + 1) \leq \vec{\tau}_k \cdot \vec{\delta} \quad \forall \vec{\delta} \in \Delta_a \setminus \{0\}$$

Since Δ_a contains only lexicographically positive vectors, we can safely write:

$$1 - (1 - \epsilon_{\ell,1})(\vec{c} \cdot \vec{N} + d + 1) \leq H(\vec{\delta}) \quad \forall \vec{\delta} \in \Delta_a$$

After simplification, this gives:

$$H(\vec{\delta}) + (1 - \epsilon_{\ell,1})(\vec{c} \cdot \vec{N}) - \epsilon_{\ell,1}(d + 1) + d \geq 0 \quad \forall \vec{\delta} \in \Delta_a$$

On the example, $\vec{N} = (N)$. Experimentally, it seems sufficient to take $\vec{c} = (10)$ and $d = 10$ as big enough values. This simplifies to:

$$H(\vec{\delta}) + (10 - 10\epsilon_{\ell,1}).N - 11\epsilon_{\ell,1} + 10 \geq 0 \quad \forall \vec{\delta} \in \Delta_a$$

We end up with an affine form $\varphi(\vec{\delta}, N) = H(\vec{\delta}) + (10 - 10\epsilon_{\ell,1}).N - 11\epsilon_{\ell,1} + 10$ non-negative on a non-empty polyhedron Δ_a. Note that decision variables ϵ are considered as parameters and are treated as special constants (line 1). This can be linearized by playing with the affine form of Farkas. This is completly automated with FKCC thanks to **solve** and **positive_on** primitives (lines 35-37) : `H_linear_p1` will contain an equivalent conjunction of affine constraints. Finally, line 42 filters the only required variables: the coefficients of H (actually $\vec{\tau}_k$). Repeating this process for each partition of Δ_a, we end-up with a conjunction of affine constraints expressing CORRECT($\vec{\tau}_k$).

Efficiency Consider Equation 5.5.3 and the first inequality:

$$-(\vec{e} \cdot \vec{N} + f) \leq \vec{\tau}_k \cdot \vec{\delta} \quad \text{for all } \vec{\delta} \in \Delta_{a,\ell}$$

This can be rewritten as:

$$H(\vec{\delta}) + (\vec{e} \cdot \vec{N} + f) \geq 0 \text{ for all } \vec{\delta} \in \Delta_{a,\ell}$$

Again, this translates to the non-negativity of the affine form $\psi(\vec{\delta}, \vec{N}) = H(\vec{\delta}) + (\vec{e} \cdot \vec{N} + f)$ on the non-empty polyhedron $\Delta_{a,\ell}$, and the affine Farkas lemma may also help to derive affine constraints (Figure 5.6, line 12). The same apparatus applies for the other inequality. Finally, the result is obtained by lexicographically minimizing the constraints (line 25) with variable order prescribed with the **keep** primitive (line 23). The final result is:

```
inv_nu = -4
nu = 4
bound_0 = 2
bound_1 = 0
H_0 = -1
H_1 = 2
H_2 = 0
x_11_counter = 1
x_12_counter = 0
x_21_counter = 1
x_22_counter = 0
x_31_counter = 0
x_32_counter = 1
x_41_counter = 0
x_42_counter = 1
```

The modulo is $b_1(N) = 1 + \text{bound}_0.N + \text{bound}_1 = 2N + 1$, $\vec{\tau}_1 = (H_0, H_1) = (-1, 2)$ – on Figure 5.3.(b), the red line has normal $\vec{\tau}_1$. The four conflict sets are solved (nu = 4), positively on $\hat{\Delta}_{blurx,1}$ and $\hat{\Delta}_{blurx,2}$ (above the red line), negatively on $\hat{\Delta}_{blurx,3}$ and $\hat{\Delta}_{blurx,4}$ (below the red line). Hence the algorithm ends after one iteration and returns the mapping $\sigma(i, j) = -i + 2j \bmod 2N + 1$.

5.6 Experimental Validation

This section presents an experimental validation of our contributions. Section 5.6.2 assesses the scalability of our liveness approach, while Section 5.6.3 assesses the scalability of our linear array contraction method and the accuracy of our liveness approach (despite the approximation made by the widening). Finally, Section 5.6.4 presents a detailed analysis of the results for each kernel benchmark.

5.6.1 Setup

We have implemented our liveness algorithm in PoLa. We applied by hand our linear mapping algorithm using the FKCC tool. We conducted our experiments on the kernels depicted on Figure 5.10. All the stencils are implemented with a *perfect loop* nest and *single-assignment arrays*.

- **Blur filter** (motivating example) applies a composition of two 1D convolutions on a picture modeled as a 2D array.
- **Jacobi 1D** is a 1D stencil, 3 points, with dependences from the previous timestamp.
- **Seidel 1D** is a 1D stencil, 3 points, with a dependence from the same timestamp.
- **Jacobi 2D** is a 2D stencil, 5 points, with dependences from the previous timestamp.
- **Heat 3D** is a 3D stencil, 7 points, with dependences from the previous timestamp.

The timings were measured on a machine equipped with an Intel core i9-10885H CPU 2.40GHz with 64GB of DDR memory. All the timings are measured in seconds.

5.6.2 Liveness Analysis

Table 5.1 summarizes the performances of our liveness analysis approach. The baseline is the PoCo implementation of the usual liveness analysis for arrays described in 2.2.1, exploring each

couple of liveness interval ($write1, read1$), ($write2, read2$) to build the conflict relation $\bowtie$ and then retrieving Δ_a by using a projection of $\{(\vec{i}, \vec{j}, \vec{\delta}) \mid \vec{\delta} = \vec{i} - \vec{j},\ \vec{i} \bowtie \vec{j}\}$ on $\vec{\delta}$.

For each kernel, we give the parameter selection $\mathcal{N}$ (Parameters), the cumulative trace size (Trace), the total execution time of our approach (Timing). As for the baseline, we provide the cumulative number of pairs of liveness intervals considered, including the empty ones (Iterations) and the projections (Projections) for the pairs of intersecting liveness intervals; as well as the total execution time (Timing). Finally, we give the speed-up factor (Speed-up) with the formula Timing(Baseline) / Timing(Our method). Our method appears to be more scalable than the baseline with a speed-up ranging from 1.9 for Heat 3D to 17 for the Blur filter kernel. Unsurpringly, the more parameters we have, the more traces we have to generate. This explains the gap between Blur and the 1D stencils Jacobi 1D and Seidel 1D. It is worth to note that trace size directly scales with the dimension of the loop nest, hence the total execution time. This explains the timing gap between 1D stencils (Jacobi 1D, Seidel 1D), 2D stencil (Jacobi 2D) and 3D stencil (Heat 3D). Still, we observe a speed-up with our method, as the complexity of the baseline also increases with the depth of the loop nest.

Figure 5.2 details step-by-step the performances of our method, that highlights which step dominates the execution time. (a) gives a normalized comparison between the benchmarks, while (b) gives the timing details. For each kernel, we provide the timing for the following:

- the computation of the PRDG (Dependence Analysis)
- the liveness analysis, computing $\Delta_a(\vec{N})$ for each parameter instance $\vec{N} \in \mathcal{N}$ (Liveness analysis)
- the inference of polyhedral constraints with NLR (NLR)
- the inference of $\mathcal{N}$ (Parameter selection)
- the generation of execution traces (Trace generation)

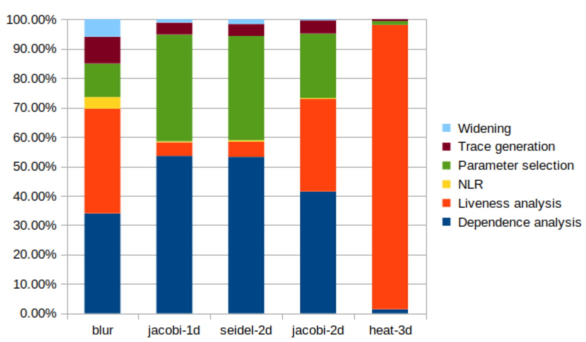
The execution tends to be dominated by the trace liveness analysis. Liveness analysis consists of two steps: a linear pass computing the liveness itself $\bowtie$, then a pass computing the conflict set $\Delta_a = \{\vec{i} - \vec{j} \mid \vec{i} \bowtie \vec{j}\}$. This pass is *quadratic* in the trace size, as we enumerate each couple $(\vec{i}, \vec{j})$. Hence that point should be improved on to enhance the scalability of our algorithm. The other steps remain negligible in comparison. In particular, the widening itself (NLR and widening) is particularly fast.

5.6.3 Linear Array Contraction

Table 5.3 illustrates the complexity of liveness constraints, when expressed with conflicts set (Our method), and when expressed as a conflict *relation*. We provide the number of polyhedral pieces ($|\Delta|$ and $|\bowtie|$), which impacts directly the complexity of the integer linear program required to derive the mapping σ. Also, we give the total cumulative number of constraints (constraints). The number of constraints directly impacts the number of Farkas variables, which would need to be projected out. In turn, this impacts the performance of the projection (**keep** construction in Figure 5.5 and Figure 5.6). $|\Delta|$ grows with the dimension of the array, because of the splitting.

Kernel	Our method			Baseline			**Speed-up**
	Parameters	Trace	Timing	Iterations	Projections	Timing	
Blur filter	4,5	40	0.007	81	39	0.1	**17**
Jacobi 1D	(6,4),(7,4),(6,5)	52	0.024	81	54	0.07	**2.9**
Seidel 1D	(6,4),(7,4),(6,5)	52	0.024	81	37	0.1	**4.4**
Jacobi 2D	(6,4),(7,4),(6,5)	336	0.19	256	160	0.84	**4.4**
Heat 3D	(6,4),(7,4),(6,5)	1856	5.7	900	504	11.3	**1.9**

Table 5.1: Liveness analysis is faster with our approach (timing in seconds)



(a) Normalized

Kernel	Dependence analysis	Liveness analysis	NLR	Parameter selection	Trace generation	Widening	Total
Blur filter	0.002	0.002	0.0002	0.00079	0.0006	0.0004	0.007
Jacobi 1D	0.013	0.0011	0.0001	0.0088	0.0009	0.0002	0.024
Seidel 1D	0.012	0.0012	0.0001	0.008	0.0009	0.0004	0.024
Jacobi 2D	0.078	0.059	0.0005	0.041	0.008	0.0006	0.19
Heat 3D	0.078	5.5	0.002	0.07	0.03	0.0006	5.7

(b) Raw data

Table 5.2: Liveness timing step-by-step (seconds)

Kernel	Our method (Δ)		Baseline ($\bowtie$)		Compaction factor
	$\|\Delta\|$	constraints	$\|\bowtie\|$	constraints	
Blur filter	4	12	3	24	2
Jacobi 1D	2	6	2	15	2.5
Seidel 1D	2	6	2	14	2.3
Jacobi 2D	6	28	7	77	2.75
Heat 3D	14	87	7	106	1.2

Table 5.3: Liveness constraints are simpler with our approach

Kernel	Our method	Baseline (from exact Δ)	Overhead (%)
Blur filter	$(i,j) \mapsto -i + 2j \bmod 2N+1$	same	0
Jacobi 1D	$(t,i) \mapsto i - t \bmod N$	$(t,i) \mapsto i - t \bmod N-1$	$\frac{1}{N-1}$
Seidel 1D	$(t,i) \mapsto -i \bmod N$	$(t,i) \mapsto -i \bmod N-2$	$\frac{2}{N-2}$
Jacobi 2D	$(t,i,j) \mapsto \begin{pmatrix} -2t + i \bmod N+2 \\ -j \bmod N \end{pmatrix}$	$(t,i,j) \mapsto \begin{pmatrix} -2t + i \bmod N \\ -j \bmod N-2 \end{pmatrix}$	$\frac{4}{N-2}$
Heat 3D	$(t,i,j,k) \mapsto \begin{pmatrix} -2t + i \bmod N+2 \\ -j \bmod N \\ -k \bmod N \end{pmatrix}$	$(t,i,j,k) \mapsto \begin{pmatrix} -2t + i \bmod N \\ -j \bmod N-2 \\ -k \bmod N-2 \end{pmatrix}$	$\frac{6N-4}{(N-2)^2}$

Table 5.4: Mappings found and footprint overhead due to widening

The compaction factor is the ratio constraints($\bowtie$) / constraints(Δ). It is due to the factoring enabled by the Δ-sets: differents pieces with $(\vec{i}, \vec{i} + \vec{\delta})$ in $\bowtie$ are factored as a single $\vec{\delta}$ in Δ. We believe that this gain in complexity makes the conflict set Δ more appropriate than the conflict relation $\bowtie$ to design scalable array contraction algorithms. Unfortunately, the tool SMO implementing [14] was not stable enough to work with the conflict relations we automatically generated on our examples. Hence, no timing measures nor deeper scalability experiments were possible.

Table 5.4 summarizes the mappings obtained by applying our method on the conflict sets obtained with our method (after splitting), and on the exact conflict sets obtained with the original liveness analysis algorithm of PoCo, simplified to keep only the lexicographically positive conflicts $\{(\vec{i}, \vec{j}) \mid \vec{i} \bowtie \vec{j},\ \vec{j} - \vec{i} \gg 0\}$ and using ISL coalescing as expected by [14]. The widening over-approximates the conflict sets and may increase the mapping footprint. We measured that increasing, as a percent of the footprint obtained from the exact conflict set (Overhead):

$$\text{overhead} = \frac{\text{footprint}(\sigma_{widening}) - \text{footprint}(\sigma_{exact})}{\text{footprint}(\sigma_{exact})}$$

For instance, on Jacobi 2D, we obtain: $\frac{(N+2)N - N(N-2)}{N(N-2)} = \frac{4}{N-2}$. On all our examples, the overhead is $\mathcal{O}(\frac{1}{N})$, which tends to be negligible for big values of N. If we want to apply our method on tiled programs, the overhead will depend on the tile size in the same way. Then, the tile size should be sufficiently large to keep a reasonnable overhead.

5.6.4 Detailed Results

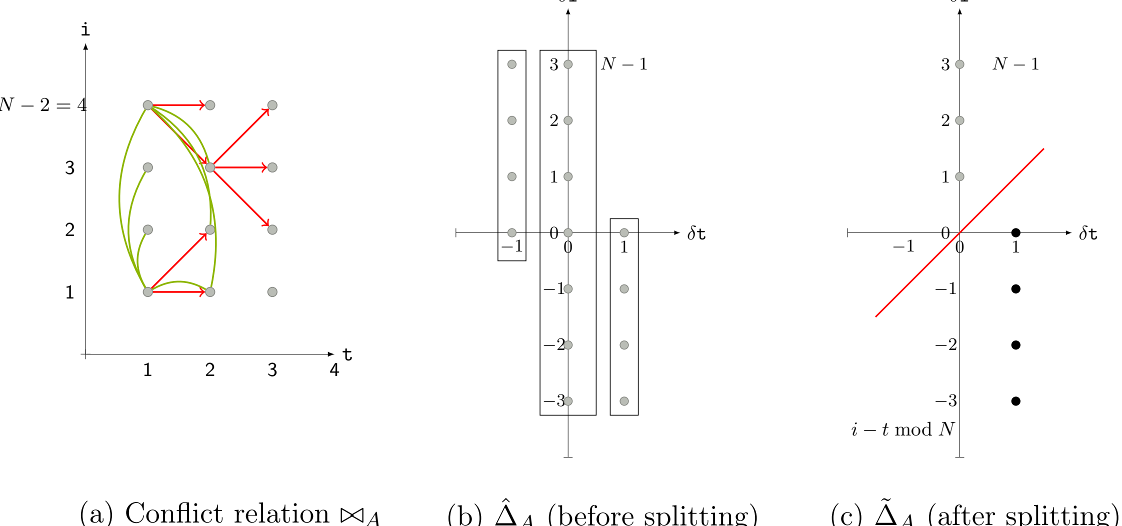
This section discusses the detail of the results obtained for each kernel: conflict sets, splitting, linear allocation.

Blur filter

With the NLR increment, the parameter selection is $N = \{4, 5\}$, which ends-up with the same behavior and the same results than provided throughout this chapter.

Jacobi 1D

Figure 5.7 depicts the liveness analysis for the Jacobi 1D kernel (a) and the results found by our approach ((b) and (c)).



(a) Conflict relation $\bowtie_A$ (b) $\hat{\Delta}_A$ (before splitting) (c) $\tilde{\Delta}_A$ (after splitting)

Figure 5.7: Jacobi 1D: Liveness analysis

On (a), we depict the iteration domain (a single one, since the loop nest is perfect) and some direct dependences (in red). All the kernels considered are single assignement, hence, we may draw directly the conflict relation on the iteration domain, binding an iteration to the array cell it writes. The critical dependence is $(t, i) \to (t+1, i+1)$, which induces conflicts depicted in green. After widening and narrowing, we get the conflict set depicted in (b): $\hat{\Delta}_A = \hat{\Delta}_{A,1} \cup \hat{\Delta}_{A,2} \cup \hat{\Delta}_{A,3}$, where:

$$\hat{\Delta}_{A,1} = \{(\delta t, \delta i) \mid \delta t = -1, 0 \leq \delta i < N\}$$
$$\hat{\Delta}_{A,2} = \{(\delta t, \delta i) \mid \delta t = 0, -N < \delta i < N\}$$
$$\hat{\Delta}_{A,3} = \{(\delta t, \delta i) \mid \delta t = 1, -N < \delta i \leq 0\}$$

After splitting, we obtain the polyhedron $\tilde{\Delta}_A = \tilde{\Delta}_{A,1} \cup \tilde{\Delta}_{A,2}$, where:

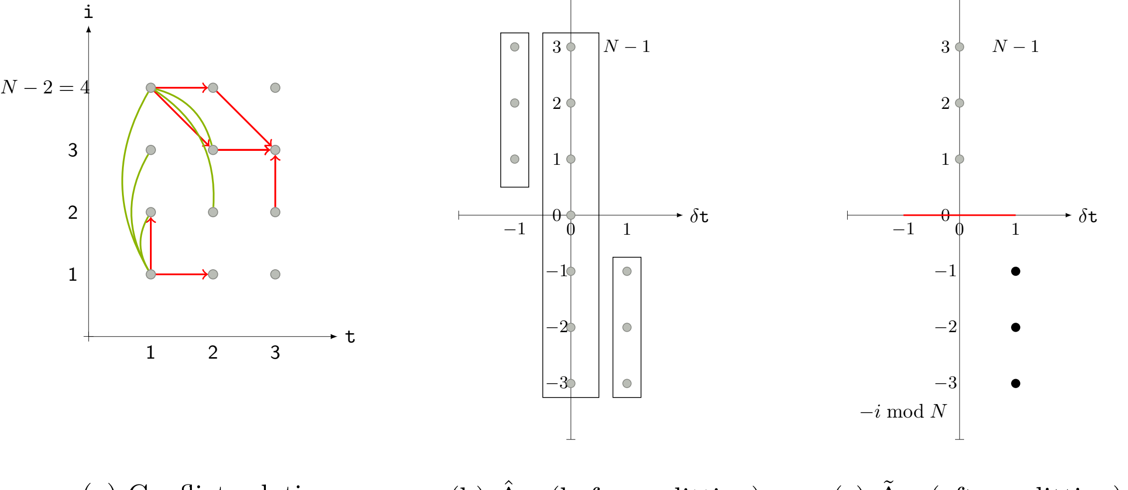
$$\tilde{\Delta}_{A,1} = \{(\delta t, \delta i) \mid \delta t = 0, 1 \leq \delta i < N\}$$
$$\tilde{\Delta}_{A,2} = \{(\delta t, \delta i) \mid \delta t = 1, -N < \delta i \leq 0\}$$

Note that the exact conflict set should have $-(N-2) \leq \delta i \leq N-2$ instead. This is the approximation made by the liveness analysis.

From $\tilde{\Delta}_A$, we apply our linear contraction algorithm, which finds the mapping $i - t \bmod N$. Remark the red hyperplane whose normal is $\vec{\tau}_1 = (-1, 1)$: it solves all the conflicts, since there is no intersection with any $\vec{\delta} \neq 0$ of $\tilde{\Delta}_A$.

Seidel 1D

Figure 5.8 depicts the Seidel 1D liveness analysis and the results obtained by our approach.



(a) Conflict relation $\bowtie_A$ (b) $\hat{\Delta}_A$ (before splitting) (c) $\tilde{\Delta}_A$ (after splitting)

Figure 5.8: Seidel 1D: Liveness analysis

As for Jacobi 1D, note the direct dependences (in red), including the critical one $(t, i) \rightarrow (t, i+1)$, which induces the conflicts depicted in green. Our method produces, after widening and narrowing, the conflict polyhedron depicted in (b): $\hat{\Delta}_a = \hat{\Delta}_{A,1} \cup \hat{\Delta}_{A,2} \cup \hat{\Delta}_{A,3}$, where:

$$\hat{\Delta}_{A,1} = \{(\delta t, \delta i) \mid \delta t = -1, 1 \leq \delta i < N\}$$
$$\hat{\Delta}_{A,2} = \{(\delta t, \delta i) \mid \delta t = 0, -N < \delta i < N\}$$
$$\hat{\Delta}_{A,3} = \{(\delta t, \delta i) \mid \delta t = 1, -N < \delta i \leq -1\}$$

After splitting, we obtain the conflict polyhedron depicted in (c): $\tilde{\Delta}_A = \tilde{\Delta}_{A,1} \cup \tilde{\Delta}_{A,2}$, where:

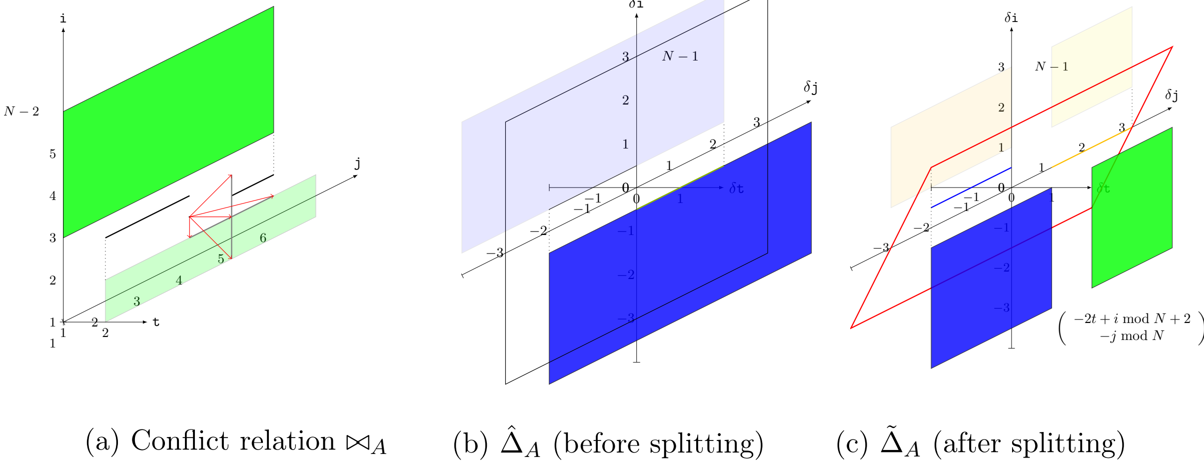
$$\tilde{\Delta}_{A,1} = \{(\delta t, \delta i) \mid \delta t = 0, 1 \leq \delta i < N\}$$
$$\tilde{\Delta}_{A,2} = \{(\delta t, \delta i) \mid \delta t = 1, -N < \delta i \leq -1\}$$

Again, the exact conflict set should satisfy $-(N-3) \leq \delta i \leq N-3$. This is the approximation made by our widening.

On $\tilde{\Delta}_A$, we apply our linear array contraction method which derives the normal to the red hyperplane, $\vec{\tau}_1 = (0, -1)$, corresponding to the mapping $\sigma_A(t, i) = -i \bmod N$. That mapping solves all the conficts $\vec{\delta} \in \tilde{\Delta}_A$.

Jacobi 2D

Figure 5.9 illustrates the liveness analysis on the Jacobi 2D kernel.



(a) Conflict relation $\bowtie_A$ (b) $\hat{\Delta}_A$ (before splitting) (c) $\tilde{\Delta}_A$ (after splitting)

Figure 5.9: Jacobi 2D: Liveness analysis

(a) depicts direct dependences from some source iteration $\vec{s} = (t_0, i_0, j_0)$. The critical dependence is $(t, i, j) \to (t+1, i+1, j)$. We depicted in green the set of writes conflicting with the write at $\vec{s}$. (c) shows the conflict set obtained after widening and narrowing: $\hat{\Delta}_A = \hat{\Delta}_{A,1} \cup \hat{\Delta}_{A,2} \cup \hat{\Delta}_{A,3} \cup \hat{\Delta}_{A,4} \cup \hat{\Delta}_{A,5}$, where:

$$\begin{aligned}
\hat{\Delta}_{A,1} &= \{(\delta t, \delta i, \delta j) \mid \delta t = 1, \delta i = 1, -N < \delta i < N, -N < \delta j \leq -1\} \\
\hat{\Delta}_{A,2} &= \{(\delta t, \delta i, \delta j) \mid \delta t = 1, -N < \delta i \leq 0, -N < \delta j < N\} \\
\hat{\Delta}_{A,3} &= \{(\delta t, \delta i, \delta j) \mid \delta t = 0, -N < \delta i, \delta j < N\} \\
\hat{\Delta}_{A,4} &= \{(\delta t, \delta i, \delta j) \mid \delta t = -1, 0 \leq \delta i < N, -N < \delta j < N\} \\
\hat{\Delta}_{A,5} &= \{(\delta t, \delta i, \delta j) \mid \delta t = -1, \delta i = -1, 1 \leq \delta j < N\}
\end{aligned}$$

After the splitting, we obtain the polyhedron depicted in (c): $\tilde{\Delta}_A = \tilde{\Delta}_{A,1} \cup \tilde{\Delta}_{A,2} \cup \tilde{\Delta}_{A,3} \cup \tilde{\Delta}_{A,4} \cup$

$\tilde{\Delta}_{A,5} \cup \tilde{\Delta}_{A,6}$, where:

$$
\begin{array}{ll}
\tilde{\Delta}_{A,1} = \{(\delta t, \delta i, \delta j) \mid \delta t = 1, \delta i = 1, -N < \delta j \leq -1\} & \text{(blue segment)} \\
\tilde{\Delta}_{A,2} = \{(\delta t, \delta i, \delta j) \mid \delta t = 1, -N < \delta j \leq 0, -N < \delta i \leq 0\} & \text{(blue rectangle)} \\
\tilde{\Delta}_{A,3} = \{(\delta t, \delta i, \delta j) \mid \delta t = 1, 1 \leq \delta j < N, -N < \delta i \leq 0\} & \text{(green rectangle)} \\
\tilde{\Delta}_{A,4} = \{(\delta t, \delta i, \delta j) \mid \delta t = 0, 1 \leq \delta i < N, -N < \delta j \leq 0\} & \text{(light yellow, left)} \\
\tilde{\Delta}_{A,5} = \{(\delta t, \delta i, \delta j) \mid \delta t = 0, 1 \leq \delta i < N, 1 \leq \delta j < N\} & \text{(light yellow, right)} \\
\tilde{\Delta}_{A,6} = \{(\delta t, \vec{\delta i}, \delta j) \mid \delta t = 0, \delta i = 0, 1 \leq \delta j < N\} & \text{(orange line)}
\end{array}
$$

Note that the exact conflict set should satisfy $-(N-3) \leq \delta i, \delta j \leq N-3$, this is the approximation made by our method.

We applied our linear contraction algorithm on $\tilde{\Delta}_A$:

- The *first iteration* computes the normal $\vec{\tau}_1 = (-2, 1, 0)$, hence the first mapping dimension $(t, i, j) \mapsto (-2t + i \bmod N + 2)$. The intersection between the hyperplane directed by $\vec{\tau}_1$ (depicted in red) and $\tilde{\Delta}_A$ is the segment $S = [(0, -(N-2), 0), (0, (N-2), 0)]$ which is non-null, hence we need a second iteration to satisfy those conflicts.

- The *second iteration* on unsolved conflicts S gives the normal $\vec{\tau}_2 = (0, 0, -1)$, hence the second mapping dimension $(t, i, j) \mapsto (-j \bmod N)$. The intersection between the hyperplane whose normal is $\vec{\tau}_2$ is $\{0\}$, hence our algorithm terminates and finally outputs the mapping $\sigma_A(t, i, j) = \begin{pmatrix} -2t + i \bmod N + 2 \\ -j \bmod N \end{pmatrix}$.

Heat 3D

After widening and narrowing, we obtain the conflict polyhedron $\hat{\Delta}_A = \hat{\Delta}_{A,1} \cup \hat{\Delta}_{A,2} \cup \hat{\Delta}_{A,3} \cup \hat{\Delta}_{A,4} \cup \hat{\Delta}_{A,5} \cup \hat{\Delta}_{A,6} \cup \hat{\Delta}_{A,7}$, where:

$$
\begin{array}{l}
\hat{\Delta}_{A,1} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 1, \delta i = 1, \delta j = 0, -N < \delta k \leq -1\} \\
\hat{\Delta}_{A,2} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 1, \delta i = 1, -N < \delta j \leq -1, -N < \delta k < N\} \\
\hat{\Delta}_{A,3} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 1 - N < \delta i \leq 0 - N < \delta j, \delta k < N\} \\
\hat{\Delta}_{A,4} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 0, -N < \delta i, \delta j, \delta k < N\} \\
\hat{\Delta}_{A,5} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = -1, -N < \delta i \geq 0, -N < \delta j, \delta k < N\} \\
\hat{\Delta}_{A,6} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = -1, \delta i = -1, 1 \leq \delta j < N, -N < \delta k < N\} \\
\hat{\Delta}_{A,7} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = -1, \delta i = -1, \delta j = 0, 1 \leq \delta k < N\}
\end{array}
$$

After splitting, we obtain a polyhedron $\tilde{\Delta}_A = \cup_{\ell=1}^{14} \tilde{\Delta}_{A,\ell}$, with:

$$\begin{array}{l}
\tilde{\Delta}_{A,1} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 1, \delta i = 1, \delta j = 0, -N < \delta k < 0\} \\
\tilde{\Delta}_{A,2} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 1, \delta i = 1, -N < \delta j < 0, -N < \delta k \leq 0\} \\
\tilde{\Delta}_{A,3} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 1, \delta i = 1, -N < \delta j < 0, 1 \leq \delta k < N\} \\
\tilde{\Delta}_{A,4} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 1, -N < \delta i \leq 0, -N < \delta j, \delta k \leq 0\} \\
\tilde{\Delta}_{A,5} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 1, -N < \delta i, \delta k \leq 0, 1 \leq \delta j < N\} \\
\tilde{\Delta}_{A,6} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 1, -N < \delta i, \delta j \leq 0, 1 \leq \delta k < N\} \\
\tilde{\Delta}_{A,7} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 1, -N < \delta i \leq 0, 1 \leq \delta j, \delta k < N\} \\
\tilde{\Delta}_{A,8} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 0, 1 \leq \delta i < N, -N < \delta j, \delta k \leq 0\} \\
\tilde{\Delta}_{A,9} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 0, 1 \leq \delta i, \delta j < N, -N < \delta k \leq 0\} \\
\tilde{\Delta}_{A,10} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 0, 1 \leq \delta i, \delta k < N, -N < \delta j \leq 0\} \\
\tilde{\Delta}_{A,11} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 0, 1 \leq \delta i, \delta k < N, 1 \leq \delta j < N\} \\
\tilde{\Delta}_{A,12} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 0, \delta i = 0, 1 \leq \delta j < N, -N < \delta k \leq 0\} \\
\tilde{\Delta}_{A,13} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 0, \delta i = 0, 1 \leq \delta j, \delta k < N\} \\
\tilde{\Delta}_{A,14} = \{(\delta t, \delta i, \delta j, \delta k) \mid \delta t = 0, \delta i = 0, \delta j = 0, 1 \leq \delta k < N\}
\end{array}$$

Again, we should have $-(N-3) \leq \delta i, \delta j, \delta k \leq N-3$, this is the approximation made by our method. We apply our linear array contraction algorithm which finds the mapping in three iterations:

- *Iteration 1* finds the dimension $(t,i,j,k) \mapsto -2t+i \bmod N+2$, which left unsolved $\tilde{\Delta}_{A,12}$, $\tilde{\Delta}_{A,13}$ and $\tilde{\Delta}_{A,14}$. We then iterate on these three polyhedra intersected with $-2\delta t + \delta i = 0$.
- *Iteration 2* finds the dimension $(t,i,j,k) \mapsto -j \bmod N$, which still leaves unsolved $\tilde{\Delta}_{A,14} \wedge -2\delta t + \delta i = 0$. We then iterate on that polyhedron intersected with $-\delta j = 0$.
- *Iteration 3* finds the dimension $(t,i,j,k) \mapsto -k \bmod N$, which solves $\tilde{\Delta}_{A,14} \wedge -2\delta t + \delta i = 0 \wedge -\delta j = 0$. Hence our algorithm terminates and outputs the mapping $\sigma_A(t,i,j,k) = \begin{pmatrix} -2t+i \bmod N+2 \\ -j \bmod N \\ -k \bmod N \end{pmatrix}$

5.7 Conclusion

In this chapter, we proposed a fast and scalable liveneness analysis based on trace analysis. The generality is ensured by a widening operator, followed by a narrowing operator to bound the extrapolation. We prove the correctness of our approach when the program is totally unimodular, and when both dependences and schedule are quasi-uniform, notions that we introduced for this purpose. Experimental evaluation shows that our approach is faster than the state-of-the-art polyhedral liveness analysis, which opens the way for scalable channel allocation in the context of HLS. Also, we proposed a linear array allocation algorithm, rephrasing the ideas of affine scheduling on difference sets. We show that it ensures lighter ILP problems (less variables and constraints) than a similar rephrasing on conflict relations, which is the approach of the

SMO algorithm [14]. With our linear allocation algorithm, we evaluate how the liveness over-approximation introduced by the widening impacts the quality of the final allocation. We show that, on all the benchmarks, the overhead is small and tends towards 0 when the parameters increase.

There is room for many improvements. First, the complexity of our liveness analysis might be further reduced by limiting the number of traces. Note that the number of traces is $1 + |\vec{N}|$, $|\vec{N}|$ being the number of parameters. A possible clue is to reduce the number of parameters. For instance, by introducing an upper bound parameter $M > \vec{N}_i$ and deal only with M when producing the traces. Experimentally, this provides correct results on all our benchmarks, though the correctness still has to be formally proven (or a counter-example to be exhibited).

Also, the final goal is to use that technique in the context of DPN buffer allocation, as for the canonical allocation technique presented in the previous chapter. An adaption is required, as the tiled programs are not totally unimodular (because of tiling constraints $\vec{T}_S = \phi_S(\vec{i})/\vec{b}$). However, the technique might be applied on tiles separately. A possible clue to investigate would be to allocate buffers on some well-chosen generic tile, ensuring the correctness for the whole program.

```
1  parameters := {x_11,x_12,x_21,x_22,x_31,x_32,x_41,x_42};
2  objective := []->{[inv_nu,nu]: inv_nu = -nu and nu = (x_11 + x_12) +
3  (x_21 + x_22) + [...] and x_11 >= 0 and x_11 <= 1 and [...] };
4
5  D1 := [] -> {[di,dj,N]: di = 0 and 1 <= dj and dj <= N-1};
6  D2 := [] -> {[di,dj,N]: di = 1 and 1 <= dj and dj <= N-1};
7  D3 := [] -> {[di,dj,N]: di = 1 and -1*(N-1) <= dj and dj <= 0};
8  D4 := [] -> {[di,dj,N]: di = 2 and -1*(N-1) <= dj and dj <= 0};
9
10 #H(di,dj) = H_0.di + H_1.dj + H_2
11 H := affine_form(2) with H;
12
13 #Force constant H_2 to 0 (later)
14 H_linear := []->{[H_0,H_1,H_2]: H_2 = 0};
15
16 ################################################################################
17 # Conflict resolution:
18 #   H(di,dj) >= 1 - (1-x_l1)(a.N + b + 1)   (1)
19 #   H(di,dj) <= -1 + (1-x_l2)(a.N + b + 1)  (2)
20 #   a,b big enough, for all (di,dj,N) \in D1 U D2 U D3 U D4
21 ################################################################################
22
23 # (1) is written as:
24 # H(di,dj) + (a - a.x_l1).N - (b+1)x_l1 + b >= 0
25 # (2) is written as:
26 # -H(di,dj) + (a - a.x_l2).N - (b+1)x_l2 + b >= 0
27 # a = [10, ..., 10], b = 10
28
29 to_index := {[di,dj,N]->[di,dj]};
30 to_parameter := {[di,dj,N]->[N]};
31
32 #D1
33
34 #H(di) >= 1 - (1-x_l1)(a.N + b + 1)   (1)
35 H_solve_p1 := solve (H.to_index)
36                         + {[di,dj,N]-> (10-10*x_11)*N + (-11)*x_11 + 10}
37                         - (positive_on D1) = 0;
38 #H(di) <= -1 + (1-x_l2)(a.N + b + 1)  (2)
39 H_solve_n1 := solve {[di,dj,N]-> (10-10*x_12)*N + (-11)*x_12 + 10}
40                         - (H.to_index) - (positive_on D1) = 0;
41 #Both
42 H_solve1 := keep H_0,H_1,H_2 from H_solve_p1*H_solve_n1;
43
44 #same for D2, D3, D4
45 [...]
46
47 H_solve := H_solve1*H_solve2*H_solve3*H_solve4;
```

Figure 5.5: Blur filter: encoding correctness constraints

```
1  ##############################################################################
2  # Bound: -(uN + w) <= tau.(di,dj) <= uN + w
3  #          forall (di,dj,N) \in D1 U D2 U D3 U D4
4  ##############################################################################
5
6  #bound(N) = bound_0.N + bound_1
7  bound := affine_form(1) with bound;
8
9  #D1
10
11 #H(di,dj) >= -(bound_0.N + bound1) for all (di,dj,N) \in D1
12 H_lo_1 := solve (H.to_index) + (bound.to_parameter) - (positive_on D1) = 0;
13 #H(di,dj) <= bound_0.N + bound_1 for all (di,dj,N) \in D1
14 H_up_1 := solve (bound.to_parameter) - (H.to_index) - (positive_on D1) = 0;
15 #both
16 H_bound_1 := keep bound_0,bound_1,H_0,H_1,H_2 from H_lo_1 * H_up_1;
17
18 #D2, D3, D4: same
19 [...]
20
21 H_bound := H_bound_1*H_bound_2*H_bound_3*H_bound_4;
22
23 final := keep inv_nu,nu,bound_0,bound_1,H_0,H_1,H_2,x_11,x_12,x_21,x_22,x_31,x_32,
       x_41,x_42 from objective*H_bound*H_solve*H_linear;
24
25 lexmin(final);
```

Figure 5.6: Blur filter: encoding efficiency constraints

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++) {
    blurx[i][j] = in[i][j] +
      in[i][j+1] +
          in[i][j+2];
    if(i>=2)
      out[i][j] =
          blurx[i-2][j] +
    blurx[i-1][j] +
        blurx[i][j];
  }
```

(a) Blur filter

```

for (i = 1; i <= N - 2; i++)
  A[0][i] = In[i];
for (t = 1; t < TSTEPS; t++)
  for (i = 1; i <= N - 2;
      i++)
    A[t][i] = 0.3*
      ((i==1?In[0]:A[t-1][i-1])
          +
       A[t-1][i] +
       (i==N-2?In[N-1]:A[t-1][i+1]);
```

(b) Jacobi 1D

```

for (i = 1; i <= N - 2; i++)
  A[0][i] = In[i];

for (t = 1; t < TSTEPS; t++)
  for (i = 1; i <= N - 2;
      i++)
    A[t][i] = 0.3 *
      ((i==1?In[0]:A[t][i-1])
          +
       A[t-1][i] +
       (i==N-2?In[N-1]:A[t-1][i+1]);
```

(c) Seidel 1D

```

for (i = 1; i <= N - 2; i++)
  for (j = 1; j <= N - 2; j++)
    A[0][i][j] = In[i][j];

for (t = 1; t < TSTEPS; t++)
  for (i = 1; i <= N - 2; i++)
    for (j = 1; j <= N - 2; j++)
      A[t][i][j] = 0.2 *
    ((i==1?  In[0][j]:  A[t-1][i-1][j])+
     (i==N-2?In[N-1][j]:A[t-1][i+1][j])+
                        A[t-1][i][j]+
     (j==1?  In[i][0]:  A[t-1][i][j-1])+
     (j==N-2?In[i][N-1]:A[t-1][i][j+1]);
```

(d) Jacobi 2D

```

for (i = 1; i <= N - 2; i++)
  for (j = 1; j <= N - 2; j++)
    for (k = 1; k <= N - 2; k++)
      A[0][i][j][k] = In[i][j][k];

for (t = 1; t < TSTEPS; t++)
  for (i = 1; i <= N - 2; i++)
    for (j = 1; j <= N - 2; j++)
      for (k = 1; k <= N - 2; k++)
    A[t][i][j][k] = 0.16 *
      ((i==1?  In[0][j][k]  :A[t-1][i-1][j][k])+
       (i==N-2?In[N-1][j][k]:A[t-1][i+1][j][k])+
       (j==1?  In[i][0][k]  :A[t-1][i][j-1][k])+
       A[t-1][i][j][k]+
       (j==N-2?In[i][N-1][k]:A[t-1][i][j+1][k])+
       (k==1?  In[i][j][0]  :A[t-1][i][j][k-1])+
       (k==N-2?In[i][j][N-1]:A[t-1][i][j][k+1]));
```

(e) Heat 3D

Figure 5.10: Benchmarks

Chapter 6

Conclusion

Our research aimed at investigating the use of trace analysis in the process of automatic optimizations. We have focused on the problem of memory allocation, and outlined suitable program models for both constant and parametrized memory mappings. We realise that both in the context of HLS, and for computing linear parametrized mappings on polyhedral programs. We demonstrate experimentally that our methods scale better than the state-of-the-art approaches. This scalability is attained thanks to the simplification of the problems, due to the fact that we operate on traces. Projections and maximisation can be done directly on small set of points instead of using expensive polyhedral methods.

In this chapter, we will conclude the work presented in this manuscript by summarizing our contributions. We shall then end on questions we left open, as well as discussing potentially interesting future work that could further valorize our contributions.

6.1 Contributions

Canonical Array Contraction In the context of Data-Aware Process Networks where programs are *tiled*, we created a lightweight method to size the many buffers that the DPN form induces. Because of this, even with the simplest example kernels, the underlying method for sizing their buffers needs to be as lightweight as possible. We demonstrate that trace analysis on a small part of a trace is sufficient for this purpose:

- We outline the program model suitable for this approach. We propose an analysis of the program and its dependencies to deduce the program buffers' *localizability* and θ-uniformity. A program and its arrays being *localizable* ensures that the buffers can be sized with constant modulo mappings and might be derived from a single execution trace.
- In turn, when the *localizable* buffers are *θ-uniform*, we can select the trace to be considered for liveness analysis. A direct dependence being *θ-uniform* means the associated dependence function does not change while moving along the *tile band*.
- We present our *trace analysis* algorithm, which analyses the liveness on the trace and applies an instance of the *successive modulo* to derive memory allocations.

- We present the results of our experimental validation, by comparing our method to the baseline *parametrized successive modulo* algorithm and its relaxed *non-parametrized* version. We first demonstrate that most POLYBENCH kernels satisfy our assumptions, hence our program model is not too restrictive. Then, we demonstrate the scalability of our method, showing speed-ups ranging from 7.8 to 329 over the *parametrized* baseline, and from 1.1 to 11.5 over the relaxed *non-parametrized* baseline.

Linear Array Contraction We iterated on this trace-based approach by considering parametric memory mappings of *linear* form. This led to the creation of another similar lightweight approach, where we once again execute programs with small parameters to obtain small traces of executions, in order to infer array liveness thanks to a *widening* operator. Furthermore, we propose a polyhedral linear array contraction algorithm that operates on conflict sets instead of conflict relations. This way, the constraints are lighter than the SMO approach.

- In a similar fashion to our *canonical* methodology, we detail the required program properties for our approach to guarantee the correctness of our approach. We introduce the notion of *totally unimodular* programs. Also, the schedules and direct dependences of the program are expected to be *quasi-uniform*. Less kernels have only *quasi-uniform* dependences: this requirement excludes 3 kernels from our benchmarks.

- We go over our *liveness extrapolation* algorithm, which notably includes our *trace analysis algorithm*. It consists in reconstructing conflict polyhedra from the execution trace, using the NLR algorithm. It is worth to note that the latter has been re-tooled to output conflict polyhedra rather than loop nests.

- We present our approach to perform *linear array contraction*. We built an algorithm which operates on *difference sets* rather than conflict relations, greatly reducing the constraints to be solved.

- To conclude, we present our experimental results, describing the setup then comparing our array liveness approach to the baseline. We discuss its scalability by enumerating the speed-ups over the kernels, ranging from 1.9 to 17. We show that some overhead originates from the extrapolation of the conflict set, leading to mappings of greater size than using the exact *difference set*, but of which overhead is in $\mathcal{O}(\frac{1}{N})$. Finally, we give detailed results on each benchmark.

6.2 Publications

The ideas developed in Chapter 4 have been published to C3PO'22 [P2] and IMPACT'22 [P3] into a preliminary form. The IMPACT'22 publication is a preliminary paper which describes a technique that uses an oracle to determine if the deduced mapping is correct, while the C3PO'22 paper is a more comprehensive description of this technique, which no longer relied on an oracle, and instead measured the maximal footprint of the program at each execution point to infer its correctness. In addition to this PhD work, we contributed a COMPAS'22 paper [P1], which

exploits this memory sizing to apply *array scalarization* (or *scalar promotion*), a technique that transform array accesses into register accesses to speed up computations.

6.3 Perspectives

We discuss future research directions that arise from our work. Our experimental validations showed interesting results, and we believe there is much untapped potential in this trace-based methodology.

6.3.1 An Improved Conflict Set Algorithm

In our approach, the *successive modulo* method used to determine the contraction coefficients iterates over the entire difference set by measuring the maximum distance alongside each dimension iteratively, and keeping the maxima. This can be improved if there exists an non-costly, efficient way to find the vertices of the polyhedra representing the difference set. A faster way of finding those moduli would then be to only measure the distance between the origin and the vertices. This would greatly reduce the time spent in the liveness analysis, especially for respectively highly tiled programs and highly-dimensional programs, where we see the liveness analysis pass dominating the execution time, because the liveness iterates over all the points of the difference set. We believe this improvement to be most impactful on runtimes, and on the easier side to implement.

6.3.2 Global Array Space Optimization

We realise *intra-array contraction* in our work. Therefore we did not consider creating an *inter-array contraction* approach. We think that, for this enhancement to be correct, one would perhaps need to relax the correct-by-construction constraints we have established and consider *speculative* execution, similar to Apollo's method. We believe this improvement can be impactful on the memory contraction factor, though it would no longer target HLS.

6.3.3 Other Compilation Optimizations

As stated in our motivation for our *trace-based* method for the *array contraction* optimization, we believe that many other polyhedral program transformations could also benefit from a lightened approach using trace analysis. Large-scale scheduling and tiling on compute-intensive programs, such as Polybench kernels or Deep-Neural Networks, could benefit from smaller execution traces. We believe this research direction can be fruitful, as the theoretical work on program model is already done, and both scheduling and tiling are problems that are relatively close to memory allocation.

Appendix A

Résumé du travail de thèse

Dans ce chapitre, nous donnons une présentation synthétique de ce document en Français. De nombreux détails seront donc omis de cette partie, notamment les algorithmes, exemples et preuves, et nous invitons les lecteur·ices à consulter le document en Anglais si besoin.

A.1 Introduction

Le domaine de la Synthèse Haut-Niveau (HLS) concerne les techniques de compilation de programmes haut-niveau en circuits. Pour se faire, toutes les décisions de compilation, comme l'ordonnancement ou l'allocation, seront prises à priori de la phase de production du circuit. Un circuit peut être vu comme à un programme synchrone bas niveau massivement parallèle. Les techniques de compilation employées devront donc pouvoir passer à l'échelle. Celles du domaine du Calcul Haute-Performance sont précises et expressives, mais ne conviennent pas pour des programmes de telle taille, les rendant inefficaces pour la HLS. Nous nous sommes donc penchés sur l'invention de nouvelles techniques pour pallier à ce coût prohibitif. L'intuition de départ fût de s'inspirer des techniques du domaine de l'analyse dynamique, dont le principe réside dans l'analyse des informations du programme qui sont disponibles à l'exécution, afin de déduire les optimisations de compilation. En somme, la méthode générale est donc d'exécuter puis étudier un programme source, et en déduire des optimisations de manière plus rapide que les analyses statiques classiques.

La méthodologie Polytrace. Notre motivation est donc de produire des résultats similaires aux analyses statiques du domaine de la compilation polyédrique, en travaillant avec des algorithmes plus efficaces sur des objets plus légers. Pour cela, nous exploitons la prédictabilité des nids de boucles affines (SCoPs), de la même manière que les méthodes polyédriques. Cependant, notre approche est de travailler sur les traces d'exécution des programmes. Il y a donc deux hypothèses formulées, la première étant la rapidité d'une telle approche comparée aux analyses polyédriques classiques. La seconde est qu'il est possible de déterminer, pour quels paramètres d'exécution, la trace du programme analysée mène à un résultat correct.

A.2 Contraction de tableau canonique

Dans le contexte des *data-aware process networks* (DPN), où les programmes sont tuilés, nous proposons une méthode rapide pour compiler les canaux de communication (*buffers*) d'un DPN. Les programmes sous cette forme comportent généralement un très grand nombre de canaux de communication, donc la méthode de compilation utilisée (*contraction de tableau*) doit être la plus légère possible. Nous montrons que l'analyse de trace, sur une petite partie de celle-ci, est suffisante pour réaliser cette optimisation.

- Nous définissons le modèle de programmes adapté pour cette approche. Nous proposons une analyse du programme et de ses dépendences pour en déduire la *localisabilité* et la *θ-uniformité* des tableaux temporaires du programme. S'ils sont *localisables*, ils peuvent être de taille constante et il suffit d'analyser une seule trace pour conclure. S'ils sont en plus *θ-uniformes*, il est possible de selectionner la trace pour obtenir un résultat correct.

- Nous présentons notre algorithme d'*analyse de trace*, qui réalise l'analyse de durée de vie sur la trace et utilise la technique des *modulos successifs* pour en déduire les allocations mémoires correctes.

- Nous présentons les résultats de nos expérimentations, en comparant notre approches avec une implémentation de référence des *modulos successifs paramétrés* et une version relaxée *non-paramétrée*. Nous montrons que la plupart des exemples de POLYBENCH satisfont nos hypothèses, et donc que notre modèle de programme n'est pas trop contraignante. Ensuite, nous démontrons que notre méthode passe à l'échelle, en montrant que les temps d'analyse obtenus sont 7.8 à 329 fois plus petits que ceux de la version *paramétrée*, et de 1.1 à 11.5 fois plus petits que ceux de la version *non-paramétrée*.

A.3 Contraction de tableau linéaire

En suivant l'idée de l'approche basée sur l'analyse de trace, nous nous sommes penchés sur les fonctions d'allocations paramétrées, donc *linéaires*. Nous avons créé une méthode similaire à la première, qui analyse l'exécution de trace. Cette fois, nous considérons plusieures traces, générées avec de petits paramètres, et reconstruisons une allocation paramétrée en utilisant le *widening*. De plus, nous décrivons un algorithme polyédrique de contraction de tableau, qui travaille sur des ensembles de conflits au lieu de relations de conflits. Cela permet de travailler sur des contraintes plus légères que celles de l'approche de SMO.

- D'une manière similaire à notre approche *canonique*, nous détaillons les propriétés du programme qui sont requises pour assurer que notre méthode est correcte. Les programmes doivent être *totalement unimodulaires*. De plus, les ordonnancements et les dépendences directes du programme doivent être *quasi-uniformes*. Cette notion exclut 3 programmes de notre base de tests.

- Ensuite, nous décrivons notre algorithme d'*extrapolation des durées de vie*, dans lequel se trouve notre algorithme d'*analyse de traces*. Il consiste en la reconstruction d'un polyèdre

de conflits à partir de traces d'exécution, en utilisant l'algorithme *NLR*. Ce dernier a été modifié pour calculer des polyèdres de conflits plutôt que des nids de boucle.

- Nous présentons notre approche pour la *contraction linéaire de tableau*. Nous présentons un algorithme qui calcule sur des *ensembles de différences* plutôt que des relations de conflits, ce qui réduit considérablement le nombre de contraintes à résoudre par la suite.

- Pour conclure, nous étayons nos résultats expérimentaux, en décrivant la configuration necéssaire, puis en comparant notre approche à une implémentation de référence. Nous discutons de la capacité de cette approche de passer à l'échelle, en étudiant les temps d'analyses des deux méthodes, et en constatant des facteurs de réduction allant de 1.9 à 17. Nous montrons que l'extrapolation de l'ensemble de conflits induit des fonctions d'allocations plus grandes que celles déduites en utilisant l'ensemble des différences exact, mais que ce surplus de taille est en $\mathcal{O}(\frac{1}{N})$. Enfin, nous détaillons les résultats sur chaque exemple.

Personal Bibliography

[P1] Alec Sadler, Christophe Alias, and Hugo Thievenaz. A polyhedral approach for scalar promotion. In *Conférence francophone d'informatique en Parallélisme, Architecture et Système (COMPAS'22)*, 2022.

[P2] Hugo Thievenaz, Keiji Kimura, and Christophe Alias. Lightweight array contraction by trace-based polyhedral analysis. In *C3PO 2022-International Workshop on Compiler-assisted Correctness Checking and Performance Optimization for HPC*, 2022.

[P3] Hugo Thievenaz, Keiji Kimura, and Christophe Alias. Rephrasing polyhedral optimizations with trace analysis. In *12th International Workshop on Polyhedral Compilation Techniques (IMPACT'22)*, 2022.

Bibliography

[1] Aravind Acharya, Uday Bondhugula, and Albert Cohen. Polyhedral auto-transformation with no integer linear programming. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 529–542, 2018.

[2] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers principles, techniques & tools*. pearson Education, 2007.

[3] Christophe Alias. fkcc: the farkas calculator. In *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part II 3*, pages 526–536. Springer, 2020.

[4] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+ cl@ k: An implementation of lattice-based array contraction in the source-to-source translator rose. *ACM SIGPLAN Notices*, 42(7):73–82, 2007.

[5] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to high-level synthesis for fpga. *ACM SIGPLAN Notices*, 47(8):285–286, 2012.

[6] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. Fpga-specific synthesis of loop-nests with pipelined computational cores. *Microprocessors and Microsystems*, 36(8):606–619, 2012.

[7] Christophe Alias and Alexandru Plesco. Data-aware process networks. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 1–11, 2021.

[8] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[9] Corinne Ancourt, Coelho Fran, and Irigoin Ronan Keryell. A linear algebra framework for static hpf code distribution. *A; a*, 1(t2):1, 1993.

[10] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 39–50, 1991.

[11] Ivan Augé, Frédéric Pétrot, François Donnet, and Pascal Gomez. Platform-based design from parallel C specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(12):1811–1826, 2005.

[12] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers 16*, pages 209–225. Springer, 2004.

[13] Somashekaracharya G Bhaskaracharya, Uday Bondhugula, and Albert Cohen. Automatic storage optimization for arrays. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(3):1–23, 2016.

[14] Somashekaracharya G Bhaskaracharya, Uday Bondhugula, and Albert Cohen. Smo: An integrated approach to intra-array and inter-array storage optimization. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 526–538, 2016.

[15] Uday Bondhugula, Aravind Acharya, and Albert Cohen. The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(3):1–32, 2016.

[16] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 101–113, 2008.

[17] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[18] Juan Manuel Martinez Caamaño, Aravind Sukumaran-Rajam, Artiom Baloian, Manuel Selva, and Philippe Clauss. Apollo: Automatic speculative polyhedral loop optimizer. In *IMPACT 2017-7th International Workshop on Polyhedral Compilation Techniques*, page 8, 2017.

[19] João M. P. Cardoso and Pedro C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. 2009.

[20] Philippe Clauss, Federico Javier Fernández, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE transactions on very large scale integration (VLSI) systems*, 17(8):983–996, 2009.

[21] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. Gaut: A high-level synthesis tool for dsp applications: From c algorithm to rtl architecture. *High-level synthesis: From algorithm to digital circuit*, pages 147–169, 2008.

[22] Philippe Coussy, Gwenole Corre, Pierre Bomel, Eric Senn, and Eric Martin. High-level synthesis under i/o timing and memory constraints. In *2005 IEEE International Symposium on Circuits and Systems*, pages 680–683. IEEE, 2005.

[23] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[24] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, 1989.

[25] Alain Darte, Alexandre Isoard, and Tomofumi Yuki. Extended lattice-based memory allocation. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 218–228, 2016.

[26] Alain Darte, Rob Schreiber, and Gilles Villard. Lattice-based memory allocation. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 298–308, 2003.

[27] Eddy De Greef, Francky Catthoor, and Hugo De Man. Array placement for storage size reduction in embedded multimedia systems. In *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 66–75. IEEE, 1997.

[28] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. Toward speculative loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4229–4239, 2020.

[29] Paul Feautrier. Parametric integer programming. *RAIRO-Operations Research*, 22(3):243–268, 1988.

[30] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20:23–53, 1991.

[31] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II: Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[32] Paul Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 34:459–487, 2006.

[33] Paul Feautrier and Christian Lengauer. Polyhedron model. *Encyclopedia of parallel computing*, 1:1581–1592, 2011.

[34] P. M. Gruber and C. G. Lekkerkerker. *Geometry of Numbers*. North Holland, second edition, 1987.

[35] Don E Heller and Ilse CF Ipsen. Systolic networks for orthogonal decompositions. *SIAM Journal on Scientific and Statistical Computing*, 4(2):261–269, 1983.

[36] Alexandre Isoard. *Extending Polyhedral Techniques towards Parallel Specifications and Approximations*. PhD thesis, 2016.

[37] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP congress*, volume 74, pages 471–475, 1974.

[38] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM (JACM)*, 14(3):563–590, 1967.

[39] Alain Ketterlin and Philippe Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 94–103, 2008.

[40] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, 1973.

[41] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The alpha language and its use for the design of systolic arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 3(3):173–182, 1991.

[42] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel computing*, 24(3-4):649–671, 1998.

[43] Amy W Lim and Monica S Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel computing*, 24(3-4):445–475, 1998.

[44] Amy W Lim, Shih-Wei Liao, and Monica S Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 103–112, 2001.

[45] Junyi Liu, John Wickerson, Samuel Bayliss, and George A Constantinides. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1802–1815, 2017.

[46] Andrew Makhorin. Glpk (gnu linear programming kit). *http://www. gnu. org/s/glpk/glpk. html*, 2008.

[47] Antoine Morvan, Steven Derrien, and Patrice Quinton. Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(3):339–352, 2013.

[48] A Mozipo, D Massicote, Patrice Quinton, and Tanguy Risset. Automatic synthesis of a parallel architecture for kalman filtering using mmalpha. In *International conference on parallel computing in electrical engineering (PARELEC 98)*, pages 201–206, 1999.

[49] Arjun Pitchanathan, Christian Ulmann, Michel Weber, Torsten Hoefler, and Tobias Grosser. Fpl: Fast presburger arithmetic through transprecision. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–26, 2021.

[50] Alexandru Plesco. *Transformations de Programmes Et Optimisations de L'architecture Mémoire Pour la Synthèse de Haut Niveau D'accélérateurs Matériels*. PhD thesis, Citeseer, 2010.

[51] L.N Pouchet. Polybench/c 3.2. `https://web.cs.ucla.edu/~pouchet/software/polybench/`, 2012.

[52] Reese T Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138, 1959.

[53] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5):773–815, 2000.

[54] Patrice Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*, pages 208–214, 1984.

[55] Edwin Rijpkema, Ed F Deprettere, and Bart Kienhuis. Deriving process networks from nested loop algorithms. *Parallel Processing Letters*, 10(02n03):165–176, 2000.

[56] Gabriel Rodríguez, José M Andión, Mahmut T Kandemir, and Juan Touriño. Trace-based affine reconstruction of codes. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 139–149, 2016.

[57] Andreas Simbürger, Sven Apel, Armin Größlinger, and Christian Lengauer. Polyjit: Polyhedral optimization just in time. *International Journal of Parallel Programming*, 47(5):874–906, 2019.

[58] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 24–33, 1998.

[59] William Thies, Frédéric Vivien, and Saman Amarasinghe. A step towards unifying schedule and storage optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):34–es, 2007.

[60] Alexandru Turjan. *Compiling nested loop programs to process networks*. Leiden University, 2007.

[61] Peter Vanbroekhoven. Dynamic single assignment. In *Symposium in Program Acceleration through Application and Architecture driven Code, Knoxville, TN, USA*, pages 5–7, 2002.

[62] Sven Verdoolaege. Polyhedral process networks. *Handbook of Signal Processing Systems*, pages 1335–1375, 2013.

[63] Sven Verdoolaege. A polyhedral compilation library with explicit disequality constraints. In *IMPACT 2024-14th International Workshop on Polyhedral Compilation Techniques*, 2024.

[64] Frédéric Vivien. On the optimality of feautrier's scheduling algorithm. In *Euro-Par 2002 Parallel Processing: 8th International Euro-Par Conference Paderborn, Germany, August 27–30, 2002 Proceedings 8*, pages 299–309. Springer, 2002.

[65] Tomofumi Yuki and Sanjay Rajopadhye. Memory allocations for tiled uniform dependence programs. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, pages 13–22. Citeseer, 2013.

[66] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 9–18, 2013.

Résumé

Cette thèse, intitulée “Compilation d’allocation mémoire par analyse de trace avec passage à l’échelle”, étudie l’utilisation de l’analyse de traces pour calculer des allocations mémoire efficaces. Le but plus large est d’exploiter les informations du programme qui sont disponibles à l’exécution pour aller plus vite que les techniques d’analyses statiques tradictionnelles, car elles ne parviennent pas à passer à l’échelle pour des noyaux de programmes qui sont plus lourds en termes de nombres d’instructions et qui sont de dimension élevée. Le cas d’étude est la synthèse haut-niveau dans le modèle polyédrique où le passage à l’échalle est indispensable. Nous proposons de nouvelles méthodes de contraction de tableaux basée sur des analyses de trace passant à l’échelle. Nous donnons une étude théorique et pratique de ces méthodes, en donnant des preuves pour des nouveaux algorithmes et en les implémentant pour quantifier leurs performances en terme de temps d’analyse mais aussi de facteur de réduction de mémoire. Les programmes utilisés pour mesurer les performances sont des exemples adaptés provenant de la collection de programmes PolyBench. Les modifications sont généralement les facteurs de parallélisation, le tuilage, et l’ajustement du padding. Les contributions clés de ces travaux sont la création de deux nouvelles méthodes d’analyse pour la contraction de tableau. La première a pour but de donner des tailles constantes pour les tableaux temporaires, ce qui importe grandement dans le contexte des canaux de communications des applications de calcul haute-performance. La seconde méthode est une approche plus générale qui reconstruit les conflits entre cases tableau, puis une allocation de tableau linéaire capable de produire des tailles paramétriques pour les tableaux. L’ensemble a mené à l’implémentation d’un outil nommé PoLa qui comporte 5637 lignes de code en C++.

Abstract

This thesis, titled “Scalable trace-based compile-time memory allocation”, studies the use of trace analysis to infer efficient memory allocations. The broader goal is to use program information at runtime as a way to outpace traditional static analysis techniques, which fail to scale for kernels of high dimensions and many statements. The use-case is polyhedral High-Level Synthesis where scalable methods are required. We propose novel methods for array contraction based on lightweight, scalable, trace analysis. Then, we provide a thorough theoretical study of our algorithms as well as implementations to quantify performance in analysis time but also memory reduction. The example programs used for performance measurement were variations of the polyhedral benchmark suite “PolyBench”. The variations are most often parallelism factor, tiling, and adjustment of padding. Key contributions of this thesis are the design of two new methods of analysis for the array contraction optimization. The first one focuses on yielding constant sizes for temporary arrays, which is relevant in the context of communication buffer sizing for High-Performance Computing applications. The second method is a more general approach to memory allocation which reconstructs the array liveness information and the conflict relation between array cells, as well as an algorithm to compute linear allocations able to produce parametric sizes for the arrays. This research lead to the implementation of a tool named PoLa that totals 5637 lines of C++ code.