TECHNISCHE UNIVERSITÄT DRESDEN

Chair for Compiler Construction

Towards Optimizing Compilers for Systems with Racetrack Memories

Date:
28.04.2020

Referees:
Prof. Dr.-Ing. Jeronimo Castrillon
Prof. Dr. Akash Kumar

Supervisor:
Asif Ali Khan

Master Thesis
Hauke Mewes

Statement of authorship

I hereby certify that I have authored this Master Thesis entitled *Towards Optimizing Compilers for Systems with Racetrack Memories* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 28. April 2020

[illegible]

Hauke Mewes

Abstract

Racetrack memory (RTM) is an emerging non-volatile memory that promises high speed, energy efficiency and unprecedented density. The high density in RTMs is achieved by storing multiple bits in a nanoscale tape-like cell called *racetrack*. Each racetrack has one or more access ports associated to it. The data in the racetracks needs to be *shifted* to the access port position before it can be accessed, which costs energy and increases the memory latency. Hence, the memory offsets of subsequent memory accesses linearly affect the memory performance. These shifts can be considerably reduced by transforming the input programs. However, as of now, automatic approaches exist only for scalar memory accesses.

This thesis presents the first automatic compilation framework that optimizes static loop programs over arrays for racetrack memories. To build this framework, the applicability of state-of-the-art program transformations to racetrack memories is explored. Based on that, RTM specific program schedule and memory layout transformations are suggested. The framework is implemented using the polyhedral compilation framework *Polly* to transform existing input code into a form that can achieve *minimal-offset locality*, i.e., mostly the same or neighboring memory locations are accessed. The framework reduces the number of RTM shifts in stencils and linear algebra code, thereby reducing the latency and energy consumption significantly.

Contents

1 Introduction

The memory system plays a key role in the performance and power consumption of computing devices. In order to meet the increasing computational power of the processing units, which is achieved mostly through an increase in the number of cores recently, there is a continuous demand for scaling down the technology feature size. However, the down-scaled conventional SRAM and DRAM technologies suffer from high leakage and refresh powers. To address these issues, various non-volatile memory technologies have emerged, including STT-RAM [31], RRAM [62] PCM [63], MRAM [6], and *racetrack memories* (RTMs) [40]. RTMs have two distinct advantages among the aforementioned memory technologies. They have a much higher write endurance than most other NVM technologies, comparable to SRAM and DRAM, and they can store up to 100 bits in a single memory cell [7]. However, this capacity benefit of storing multiple bits in a single cell of an RTM comes at a cost: These cells work like tapes. They have one or more access ports each to read or write the data. For that, the contents of the cells need to be *shifted* to the port position.

Thus, the latency to access a single bit in a cell is variable and depends on the position of the access port. If the data is not located far from the port position, it can be accessed fast. If it resides at multiple cells away from the port, it is a lot slower. Previous work has shown that this worst-case latency can be 25.6× higher compared to SRAM [54]. Hence, it is desirable to have memory accesses that either do not require shifting at all through reusing the same cell before proceeding, or shift once to access the neighboring domain. This requires a special kind of locality in the memory accesses, called *minimal-offset locality* by this thesis, as this is not solely affected by whether elements in the memory are close together as is the case for spatial locality, but also by the offsets of the memory cells. There are two major means to achieve minimal-offset locality. One can either change the order of the program statements to reduce the offsets between subsequent memory accesses, or change the layout of the data in the RTM. For the layout change, there is existing work [12, 13, 35, 29, 27] for placing scalar variables to minimize the shifts. However, there exists only a single optimization for minimizing the shifts in arrays that targets the specific use case of tensor contraction [30], and of now, there is no algorithm or automatic framework that can provide minimizations of shifts for array accesses.

Goal. This thesis aims to create the first automatic framework that can optimize a specific class of programs for minimal-offset locality. More specifically, this thesis focuses on optimizing *static control parts (SCoPs).* These are programs that mainly consist of affine loop nests and affine array accesses, allowing to use the so-called *polyhedral model* to analyze and transform the SCoPs. SCoPs are usually mathematical kernels like linear algebra operations, linear algebra solvers, or stencils, and typically make heavy use of arrays. Thus, they are the ideal starting point for automated optimizations minimizing the RTM shifts.

Contributions. This thesis provides an analysis of existing polyhedral schedule optimization techniques and their applicability to achieving minimal-offset locality. Then, it proposes a pattern-based schedule transformation algorithm to reduce the shifts in RTMs. Furthermore, a special memory layout transformation for stencils is introduced. The newly proposed algorithms are implemented on top of Polly [24], the polyhedral optimizer of the LLVM framework[32], and evaluated using RTSim [28] for their effect on the number of shifts, the latency and the energy consumption.

Structure. Chapter 2 provides the necessary background knowledge on racetrack memories and polyhedral optimizations that are required to understand the optimization algorithms presented in this thesis. Chapter 3 covers related work, including general polyhedral schedule optimization algorithms and existing software-based shift mitigation strategies for racetrack memories. Chapter 4 considers different approaches to schedule and memory layout transformations that aim at reducing the shifts for array accesses in racetrack memories. Chapter 5 evaluates the presented optimizations using different categories of programs having a polyhedral representation like linear algebra routines and stencils. Chapter 6 summarizes the thesis and discusses possible future work and proposals that could lead to further improvements.

2 Background

This chapter introduces the necessary concepts to understand how polyhedral compilation can help to optimize code for racetrack memory. First, the fundamentals of racetrack memories are introduced and the shift optimization problem is defined. Then the polyhedral model and important concepts related to it are introduced, including schedule trees and data dependency analysis.

2.1 Racetrack Memory

Racetrack memory (RTM) is a prototypical form of fast non-volatile memory which was first proposed in 2008 [40], the concept behind it dating back to 2004 [41]. An overview of its functionality and development is given by Parkin and Yang (in [42]), and by Bläsing et al. (in [7]), which are shortly summarized in the following.

Functionality. Racetrack memory consists of magnetic nanowires called *racetracks* that can be split into one or more magnetic regions called *domains*. Each domain has its own magnetization direction that can represent one bit. The domains are separated by domain walls. Thus, each nanowire can store multiple bits. As each nanowire represents a memory cell with multiple bits, the number of bits per nanowire is usually the same for all the nanowires in the memory. In order to access the data bits, each cell has one or more dedicated *access ports*. The port position is fixed and can only read or modify one domain at a time. Hence, for a port to access another domain, the domains need to be moved along the wire. This can be achieved by

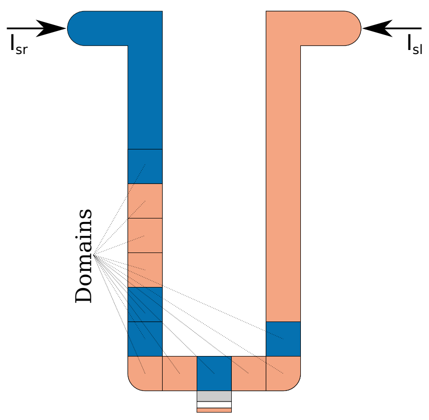
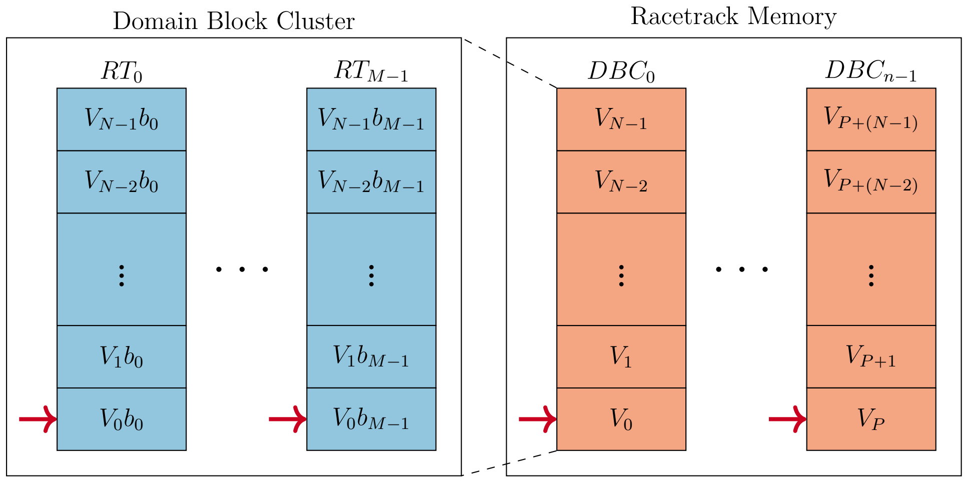


Figure 2.1: This depicts a single wire, or racetrack, which can store twelve domains. As one can see, the rest of the space on the wire is necessary to store the overflow bits that are used to store the shifted domains. I_{sr} and I_{sl} are applied to shift to the right or left respectively. This figure is adapted from figure 1 in [7, p. 3].

applying an electric current. Thus, overflow bits are required to store the moved bits. An illustration of a single wire can be found in figure 2.1.

RTM Evolution. The development of the current racetrack memory consists of four major steps. The first step was to show that one can use an electric current to move the domain in permalloy nanowires[26]. With that, a speed of $100\,\mathrm{m\,s^{-1}}$ is possible for the domains[40]. As permalloy nanowires are magnetically soft, the domains are relatively big and flexible in size. Hence, in a second step, the permalloy was replaced by Co/Ni, allowing smaller and more robust domains[14]. In a third step, it was possible to increase the speed of the domains up to $400\,\mathrm{m\,s^{-1}}$ by combining an ultrathin magnetic layer with an underlying heavy metal layer, for example cobalt with platinum[37]. This combination makes it possible to move the domains with a spin-polarized current that induces spin transfer torques[50]. However, this comes at the cost of limiting the domain density because of the demagnetizing fields produced by each domain, which causes neighboring domains to interfere with each other. This issue was solved in the current generation of RTM by combining two magnetic layers for a single wire which compensate each others magnetic moment, thus not interfering with neighboring wires any more. This increases the domain speed as well to up-to $1000\,\mathrm{m\,s^{-1}}$ [64].

Memory layout. For now, let us assume that a single nanowire can contain 64 domains, i.e., that each memory cell can store 64 bits. There are two possibilities to map a multi-bit variable to a racetrack. For example, if we have a double consisting of 64 bits, one could store it sequentially on a single



V: Variable
b: bit
DBC: Domain block cluster
RT: Racetrack
M: Number of racetracks per DBC
n: Number of DBCs per RTM
N: Number of domains per track
$P = N \cdot n$

Figure 2.2: This shows the architecture of the racetrack memory that is used for this thesis. The racetrack memory consists of a list of DBCs where each DBC is a group of consecutive racetracks. The ports of each racetrack can only be moved together in a lock-step fashion. This figure is adapted from figure 2 in [29, p. 4].

wire. This has the disadvantage that one has to shift through the whole wire to access the variable, performing 64 sequential reads of a single bit. This leads to a high latency. To avoid this, the idea of a *domain block cluster* (DBC) was developed in [53]. A DBC is a fixed-size group of nanowires, see figure 2.2. For now, we assume a DBC has 32 nanowires. Instead of storing the bits of a variable sequentially in a single nanowire, one can instead store the bits in different wires of a DBC in a bit-interleaved fashion. With that, a single DBC in our example can store up to 64 32-bit integers. To make the access easier, all ports of a DBC are aligned to the same positions in the racetracks and are moved together in a lock-step fashion. This memory pattern is particularly useful to store arrays. For example, if we have an arry of 192 32-bit integers, we could store it in three DBCs with the first containing the indices 0 to 63, the second containing 64 to 127 and the third 128 to 191. This is the memory layout considered in this thesis unless stated otherwise.

This works great if the array size is a multiple of the nanowire size. If this is not the case, then the required amount of DBCs is chosen and the last DBC then contains some unused bits.

A two-dimensional array with the dimensions $m \times n$ can be treated as m one-dimensional arrays that are stored sequentially in the memory. This

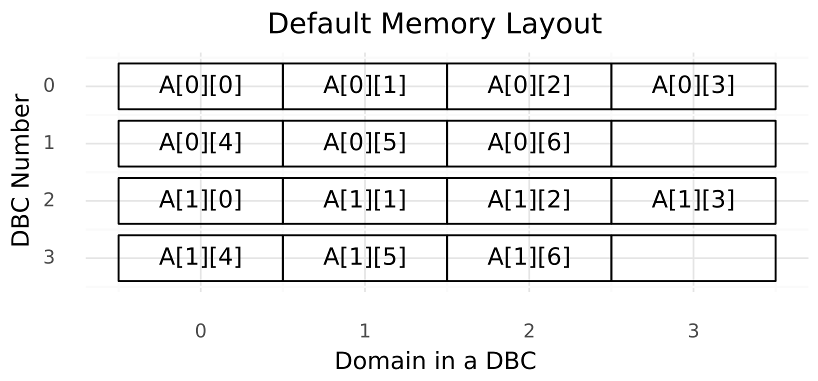


Figure 2.3: This shows the default memory layout of a two-dimensional array $A[2][7]$ on a DBC with size 4. The empty cells are unused domains, as the innermost dimension is not a multiple of the DBC size.

can be inductively generalized for arbitrary multi-dimensional arrays. For example, if we have a three-dimensional array with dimenions $16 \times 8 \times 80$, there are two DBCs used for each of the $16 \cdot 8 = 128$ outer two dimensions. A smaller example can be seen in figure 2.3.

Problem definition. Racetrack memory promises to be faster and more energy-efficient compared to all available memory technologies today [7]. However, unlike traditional memory like DRAM, the access latency to a specific memory cell is not constant, but dependendent on the position of the access port, as the domain needs to be shifted to the access port. For example, let us look at figure 2.3 again. If `A[0][3]` needs to be accessed, but the port for the DBC points to `A[0][0]`, the domain containing `A[0][3]` needs to be shifted three times, whereas when it points to `A[0][2]`, only one shift is necessary. Thus, reducing the shifts in a program should also reduce the latency. Furthermore, as the shifting requires a current, fewer shifts also lead to a smaller energy consumption of the racetrack memory. As summarized in [7], both effects are relevant in practice.

Hence, it is an interesting problem to reduce the number of shifts required in a racetrack memory. Both hardware and software optimizations can be used to achieve a reduction in the shifts [7]. This work investigates software optimizations. More specifically, it focuses on using racetrack memory as a *scratchpad memory* for storing the arrays that are currently used in a program. All scalar variables are assumed to be stored in registers and are not considered for the shift optimization. Thus, the problem that is tackled in this thesis is the following:

> Reduce the number of shifts for array accesses in a given static control part.

In the following, it is assumed that arrays already reside in the scratchpad memory and that each access port points to the first domain in its respective racetrack.

2.2 The Polyhedral Model

This section explains the polyhedral model. First, it introduces the basic definitions of representing a program in the polyhedral model. Second, it states some operations on integer sets and maps. Third, it defines schedule trees as an alternative schedule representation. Last, it explains the concept of dependency analysis in the polyhedral context.

2.2.1 Static Control Parts

The polyhedral model is a mathematical representation of a program. It can only represents programs with static control flow, referred to as *static control parts*, abbreviated as *SCoPs*.[1] In the following, the different parts of a SCoP are introduced. After that, the conditions for a program to be a valid SCoP can be derived.

Domain. The main idea of the polyhedral model is to represent each statement of a loop by its iteration variables. For example, let us consider the code in figure 2.4a. There are two statements involved. For the statement S, all combinations of variables are shown in figure 2.5 where each point represents an *instance* of the statement S. All the points put in a single set form the *domain* of the statement S. The statement instances of T can be represented as triples, as there are three loops surrounding T. When we combine these to sets in an appropriate way as described after the following definition, we get the domain of the entire SCoP. Thus, the domain exactly describes which statement instances of each statement should be executed.

To represent the domain mathematically, the following definition is introduced. It is based on [56], and used in [23, p. 23].

Definition 2.1 (Basic Integer Set).
Let $p, d, e, m \in \mathbb{N}$. A *basic integer set* Q is a mapping $Q : \mathbb{Z}^p \to 2^{\mathbb{Z}^d}, \vec{n} \to Q(\vec{n})$ where for some $A \in \mathbb{Z}^{m\times d}, B \in \mathbb{Z}^{m\times p}, D \in \mathbb{Z}^{m\times e}, \vec{c} \in \mathbb{Z}^m$, and for each $\vec{n} \in \mathbb{Z}^p$, it is

$$Q(\vec{n}) = \{\vec{x} \in \mathbb{Z}^d | \exists \vec{z} \in \mathbb{Z}^e : A\vec{x} + B\vec{n} + D\vec{z} + \vec{c} \geqslant 0\};$$

p is the *parameter dimensionality*, d the *set dimensionality*, e the number of existentially quantified variables in the constraints and m the number of constraints.

[1]There is some work to extend the polyhedral model in order to make it a bit more universally applicable.[5]

```
for(int i = 0; i <= 3; ++i) {
  for(int j = 0; j <= 3; ++j)
S:  C[i][j] = β * C[i][j];
  for(int k = 0; k <= 3; ++k)
    for(int j = 0; j <= 3; ++j)
T:    C[i][j] = C[i][j] + α * A[i][k] * B[k][j];
}
```

(a) This code shows a matrix multiplication combined with a matrix vector multiplication, known as the `gemm` kernel. There are many implementations of the `gemm` kernel possible. The one used throughout this thesis is the one as it is implemented in Polybench [65].

```
for(int i = 0; i <= n; ++i)
  for(int j = 0; j <= 3; ++j)
S:  B[i] += A[i][j]
```

(b) This is a kernel with a single statement and a parametric domain with the parameter n.

Figure 2.4: This shows two example SCoPs for explaining the details of SCoPs.

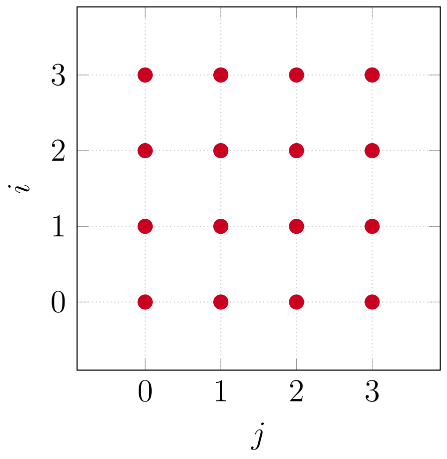


Figure 2.5: This shows the example domain for statement S in figure 2.4a.

In its direct form, this definition is a bit tedious to work with, but it is precise. To understand its parts, we take the domain for the single statement of figure 2.4b:

$$n \to \{ (i, j) \mid 0 \leqslant i \leqslant n \wedge 0 \leqslant j \leqslant 3 \} \tag{2.1}$$

First note that this example, along with the rest of the thesis, omits the matrix notation and instead describes each row of it as an inequation. In this example, $p = 1$, as there is one constant parameter to the SCoP (namely, the n), $d = 2$ as the the set is two-dimensional, $m = 4$ as there are four constraints on the values in the set, and $e = 0$ as there are no existentially quantified constraints.

For simple kernels like the one in figure 2.4b, the basic integer set is sufficient. But to describe domains of kernels with multiple statements, several additions to the definition are necessary. First of all, it is very useful to use a name for the tuple to describe the statement one is referring to with this set. For example, for figure 2.4b, the basic integer set that describes the domain can be written like this:

$$n \to \{\, \mathrm{S}(i,j) \mid 0 \leqslant i \leqslant n \wedge 0 \leqslant j \leqslant 3 \,\}$$

A basic integer set when assigned a tuple name is called a *named basic integer set*. For the rest of the work, the distinction between named and unnamed is omitted as we are usually only working with named basic integer sets. With this definition, it is possible to describe the space.

Definition 2.2 (Space)**.**
The *space of a named basic integer set Q* is the following triple:

- The number of parameters p,
- the dimensionality d,
- and the tuple name,

and is denoted by Space(Q).

For example, the space of set 2.1 is $(p = 1, d = 2, S)$. In the domain of a single statement p corresponds to the number of *constant parameters* that are necessary to describe the domain, the dimensionality d describes the number of loops that are surrounding the statement, called *statement parameters*, and the tuple name is the *statement name*.

Define an *integer set* as the union of finitely many basic integer sets with the same space. Define a *union integer set* as the union of finitely many integer set. This allows to describe the domain of a multi-statement SCoP in a single mathematical object. For example, the domain of the `gemm` kernel in figure 2.4a can be written as the following union integer set:

$$\{\, \mathrm{S}(i,j) \mid 0 \leqslant i \leqslant 3 \wedge 0 \leqslant j \leqslant 3 \,\} \cup \{\, \mathrm{T}(i,k,j) \mid 0 \leqslant i \leqslant 3 \wedge 0 \leqslant k \leqslant 3 \wedge 0 \leqslant j \leqslant 3 \,\}$$

For the remainder of this thesis, unless the context requires otherwise, basic integer set is abbreviated as basic set, integer set as set, and union integer set as union set.

Schedule. The domain of a SCoP describes which statement instances should be executed, but does not make any statement on the order of execution. For this, a SCoP needs a *schedule*. A schedule describes a partial order on the statements in which they should be executed. One way to describe the schedule is to map every of the statement instances to a d-tuple and order the execution by the lexicographic order of this map. An example for the

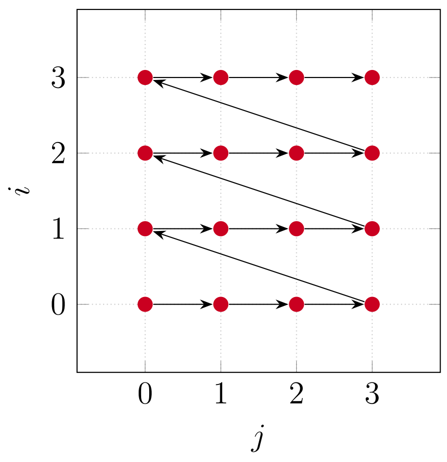


Figure 2.6: This shows the lexicographic ordering of the domain of S in figure 2.4a where each domain entry is mapped to itself. As can be seen from the arrow directions, this represents two nested for loops, where j is the inner one and i the outer one. Hence, this is the schedule of S as it shown in the code in figure 2.4a.

lexicographic order can be seen in figure 2.6. As we have already mentioned mapping, let us formally introduce maps:

Definition 2.3 (Basic Integer Map)**.**
Let $p, d_{in}, d_{out}, e, m \in \mathbb{N}$. A *basic integer map* M is a mapping $M : \mathbb{Z}^p \to 2^{\mathbb{Z}^{d_{in}} \times \mathbb{Z}^{d_{out}}}$ where for some $A_{in} \in \mathbb{Z}^{m \times d_{in}}, A_{out} \in \mathbb{Z}^{m \times d_{out}}, B \in \mathbb{Z}^{m \times p}, D \in \mathbb{Z}^{m \times e}, \vec{c} \in \mathbb{Z}^m$, and for each $\vec{n} \in \mathbb{Z}^p$, it is

$$M(\vec{n}) = \{\vec{x}_{in} \to \vec{x}_{out} \in \mathbb{Z}^{d_{in}} \times \mathbb{Z}^{d_{out}} | \exists z : A_{in}\vec{x}_{in} + A_{out}\vec{x}_{out} + B\vec{s} + D\vec{z} + \vec{c} \geqslant 0\};$$

p is the *number of parameters*, d_{in} the *input dimensionality*, d_{out} the *output dimensionality*, e the number of existentially quantified variables in the constraints and m the number of constraints.

This definition is analogous to the definition of the basic integer set, except that it uses a relation with an input and output dimension instead of a single set dimension. The schedule of figure 2.4b looks like this:

$$n \to \{\, (i, j) \to (o1, o2) \mid o1 = i \wedge o2 = j \,\} \Leftrightarrow n \to \{\, (i, j) \to (i, j) \,\}$$

Like with the basic set example, the matrix representation is replaced by a short notation of the constraints. Furthermore, in case an input parameter is equal to an output parameter, it can be written as on the right side. Similar to the basic sets, it can be useful to assign a name to the input or output dimension of the basic integer map. In that case it is called a *named basic integer map*. In our case, this distinction is omitted for the rest of this work, as nambed basic interger maps are used almost always. For example, the schedule of figure 2.4b can be named as follows:

$$n \rightarrow \{\, \mathrm{S}(i, j) \rightarrow (i, j) \,\}. \tag{2.2}$$

Hence, the spaces for basic integer maps look like this:

Definition 2.4 (Space II)**.**
The *space of a named basic integer map M* is the following quintuplet:

- The number of parameters p,
- the input dimension d_{in},
- the output dimension d_{out},
- the input tuple name,
- and the output tuple name,

and is denoted by Space(M).

For example, the space of map 2.2 is $(p = 1, d_{in} = 2, d_{out} = 2, S, \epsilon)$, where ϵ is the empty word. Define an *integer map* as the union of finitely many basic integer maps with the same space, and define a *union integer map* as the union of finitely many integer maps. For the rest of this thesis, unless the context requires otherwise, basic integer maps are abbreviated as basic maps, integer maps as maps and union integer maps as union maps.

For schedules, the output dimensionality of the schedule's integer map is called the *schedule dimension*, while the input dimensionality still describes the number of *statement parameters.* The schedule dimension is used to classify schedules either as *one-dimensional*, when the schedule dimension is equal to one, or as *multi-dimensional*, when the schedule dimension is bigger than one. For example, the schedule represented by equation 2.2, is multi-dimensional.

Memory Accesses. With the domain and the schedule, we have enough information to describe an abstract form of a program. However, memory accesses are a fundamental component of any program, and without them the programs would be very restricted in their capabilities. Let us recall that this work aims to optimize array memory accesses for racetrack memories. For this, we need a representation of array *memory accesses.* The usual way to do this in the polyhedral model is to encode the array access expressions as affine expressions of the statement instance vectors. These can be represented by integer maps, where for each array memory access in a statement, the statement instance is mapped to a tuple of affine array index expressions, one for each dimension of the array. These integer maps are called *memory access maps* or *memory access relations.* In the memory access maps, the input dimensionality also describes the number of statement parameters, the output dimensionality describes the array dimension, and the output tuple name is equal to the array name. The last output dimension of the memory access map is also called the *fastest-changing* dimension, as it usually is the one that is changed in the innermost loop nests of algorithms, and this is the dimension that is put into a single DBC in the default RTM memory layout.

For example, in figure 2.4b, the statement S accesses two arrays, namely `A` and `B`. This can be represented by the following two access maps:

$$n \to \{\, \mathrm{S}(i,j) \to \mathrm{A}(i,j) \,\}$$

and

$$n \to \{\, \mathrm{S}(i,j) \to \mathrm{B}(i) \,\}.$$

Furthermore, for the dependency analysis (cf. 2.2.4), it is important that each memory access is either marked as a read or a write access. For the running example, `A` is only read, but `B` is both read and written. Although the read and write occur at the same location, two distinct memory accesses will be created for this, one marked as a read and one as a write.

Now that all major components of a SCoP are defined, it can be summarized as follows:

Definition 2.5 (SCoP)**.**
A *static control part (SCoP)* is a mathematical representation of a part of a program. It contains the following parts:

- a union set representing the *domain* of the SCoP,
- a partial order on the domain that describes the *schedule* of the SCoP, and
- for each statement a list of *memory accesses*, where each distinct array access is a tuple consisting of its type (either read or write) and its access function, represented as an integer map.

The following conditions can be derived for a program to have a valid representation as a SCoP:

- There are four types of variables allowed in a SCoP: The first ones are integer variables that are constants from the perspective of the SCoP, called *parameters*. The other types are loop induction variables, arrays, and local variables limited to a single statement.
- The only control structures allowed are the following: First, loops that each have a single induction variable, a constant increment or decrement in each interation, called *stride*, and a lower and upper bound for the induction variable that are affine expressions in the surrounding induction variables and parameters. Second, conditional statements where conditions can only contain affine expressions in the surrounding induction variables and parameters.

- Array index expressions have to be affine expressions in the induction variables and parameters.
- Statements have to be side-effect free basic blocks that only take array elements, parameters, SCoP-local variables and loop induction variables as live-in and live-out variables.

All the above is adapted from [24]. With all that in mind, we can explain the term *polyhedral model.* For this, first introduce the definition of a *polyhedron*:

Definition 2.6 (Polyhedron)**.**
Let $d, m \in \mathbb{N}$. Let $A \in \mathbb{Z}^{m \times d}, \vec{b} \in \mathbb{Z}^m$. Then the set

$$\{\vec{x} \in \mathbb{Z}^d : A\vec{x} + \vec{b} \geqslant \vec{0}\}$$

is called a *polyhedron.*

This definition resembles the ones of basic integer sets and maps, with the exception that they have an additional existential quantifier. But when one removes the existential quantifier from the sets by letting $e = 0$ in definition 2.1 or 2.3, then both can be viewed as polyhedra by letting the parameter $\vec{n}$ be a part of the set instead of the domain of the mapping. Let us have a look at a short example for this. Take the set

$$n \rightarrow \{\, (i, j) \mid 0 \leqslant i \leqslant n \wedge 0 \leqslant j \leqslant 3 \,\}$$

again. This can be transformed into this:

$$\{\, (n, i, j) \mid 0 \leqslant i \leqslant n \wedge 0 \leqslant j \leqslant 3 \,\}$$

Hence, throughout this thesis, a basic set and map is viewed as a polyhedron if it is clear that no existential quantifiers are necessary for depicting them. This is the case for the domain of a SCoP, for example.

2.2.2 Operations on Integer Sets and Maps

This section introduces important operations on integer sets and maps. The following definitions are adapted from [55, p. 224f.]. For the remainder of this section, let $p, d_{in}, d_{out}, d \in \mathbb{Z}$, and let $\vec{n} \in \mathbb{Z}^p$.

First of all, maps can be split into their domain and range:

Definition 2.7 (Map Domain, Range, Inverse)**.**
Let $M : \mathbb{Z}^p \rightarrow 2^{\mathbb{Z}^{d_{in}} \times \mathbb{Z}^{d_{out}}}$ be a basic map. The *domain of* M is the basic set:

$$\operatorname{dom} M : \mathbb{Z}^p \rightarrow 2^{\mathbb{Z}^{d_{in}}}, \operatorname{dom} M(\vec{n}) = \{\vec{x} \in \mathbb{Z}^{d_{in}} | \exists \vec{y} \in \mathbb{Z}^{d_{out}} : \vec{x} \rightarrow \vec{y} \in M(\vec{n})\}$$

The *range of M* is the basic set:

$$\operatorname{ran} M : \mathbb{Z}^p \to 2^{\mathbb{Z}^{d_{out}}}, \operatorname{ran} M(\vec{n}) = \{\vec{y} \in \mathbb{Z}^{d_{out}} | \exists \vec{x} \in \mathbb{Z}^{d_{in}} : \vec{x} \to \vec{y} \in M(\vec{n})\}$$

The *inverse of M* is the basic map:

$$M^{-1} : \mathbb{Z}^p \to 2^{\mathbb{Z}^{d_{out}} \times \mathbb{Z}^{d_{in}}}, M^{-1}(\vec{n}) = \{\vec{y} \to \vec{x} | \vec{x} \to \vec{y} \in M(\vec{n})\}$$

All these operations are the same as in other branches of the mathematics: The *dom* operator grants access to the domain of a map, the *ran* operator to its image, and the inverse allows to switch the domain and range of the map.

Next, define the application of maps:

Definition 2.8 (Application of a Relation)**.**
Let $Q : \mathbb{Z}^p \to 2^{\mathbb{Z}^d}$ be a basic set, $M_1 : \mathbb{Z}^p \to 2^{\mathbb{Z}^d \times \mathbb{Z}^{d_{in}}}$ and $M_2 : \mathbb{Z}^p \to 2^{\mathbb{Z}^{d_{in}} \times \mathbb{Z}^{d_{out}}}$ be basic maps. If $\operatorname{Space}(Q) = \operatorname{Space}(\operatorname{dom} M_1)$, let:

$$M_1(Q) : \mathbb{Z}^p \to 2^{\mathbb{Z}^{d_{in}}}, M_1(Q)(\vec{n}) = \{\vec{y} \in \mathbb{Z}^{d_{out}} | \exists \vec{x} \in Q(\vec{n}) : \vec{x} \to \vec{y} \in M_1(\vec{n})\},$$

and if $\operatorname{Space}(\operatorname{ran} M_1) = \operatorname{Space}(\operatorname{dom} M_2)$, let:

$$\begin{aligned} &M_2 \circ M_1 : \mathbb{Z}^p \to 2^{\mathbb{Z}^d \times \mathbb{Z}^{d_{out}}}, (M_2 \circ M_1)(\vec{n}) = \\ &= \{\vec{x} \to \vec{z} \in \mathbb{Z}^d \times \mathbb{Z}^{d_{out}} | \exists \vec{y} \in \mathbb{Z}^{d_{in}} : \vec{x} \to \vec{y} \in M_1(\vec{n}) \text{ and } \vec{y} \to \vec{z} \in M_2(\vec{n})\} \end{aligned}$$

Applying a basic integer map to a basic integer set is the same as applying an arbitrary function to a set. The concatenation of two basic integer maps does the same as the concatenation of two arbitrary relations.

Next, define the delta set:

Definition 2.9 (Delta Set)**.**
Let $M : \mathbb{Z}^p \to 2^{\mathbb{Z}^d \times \mathbb{Z}^d}$ be a basic map. Define the *delta set of M* as:

$$\Delta(M) : \mathbb{Z}^p \to 2^{\mathbb{Z}^d}, \Delta(M)(\vec{n}) = \{\vec{\delta} \in \mathbb{Z}^d | \exists \vec{x} \to \vec{y} \in M(\vec{n}) : \vec{\delta} = \vec{y} - \vec{x}\}$$

Verbally, the delta set describes the subtraction of each domain element from its relating range elements. This is similar to the operation $f(x) - x$ for each x if f is a real function. Let us have a look at two small examples:

$$\Delta(\{\, \mathrm{T}(i, 0, j) \to \mathrm{T}(i, k, j) \,\}) = \{\, \mathrm{T}(0, k, 0) \,\}.$$

Second, we have:

$$\Delta(\{\, \mathrm{S}(i, j) \to \mathrm{S}(o, j) \mid -1 + i \leqslant o \leqslant 1 + i \,\}) = \{\, \mathrm{S}(i, 0) \mid -1 \leqslant i \leqslant 1 \,\}$$

For the last operation on integer maps, we need the following formal definition of the lexicographic ordering:

Definition 2.10 (Lexicographic ordering)**.**
Let $d \in \mathbb{N}$, and $\vec{a}, \vec{b} \in \mathbb{Z}^d$. We call $\vec{a}$ *lexicographically smaller than* $\vec{b}$, denoted by $\vec{a} \prec \vec{b}$, if there exists an $i \in \{1, \ldots, d\}$ such that $a_i < b_i$ and for all $j = 1, \ldots, i-1$, it is $a_j = b_j$. Denote $\vec{a} \preceq \vec{b}$ if $\vec{a} \prec \vec{b}$ or $\vec{a} = \vec{b}$.

Furthermore, define the lexicographic minimum:

Definition 2.11 (Lexicographic Minimum)**.**
Let $M : \mathbb{Z}^p \to 2^{\mathbb{Z}^{d_{in}} \times \mathbb{Z}^{d_{out}}}$ be a map. The *lexicographic minimum of* M is the map:

$$\text{lexmin}(M)(\vec{n}) := \{\vec{x} \to \vec{y} \in M(\vec{n}) | \forall \vec{z} \in \mathbb{Z}^{d_{out}} : \vec{x} \to \vec{z} \in M(\vec{n}) \Rightarrow \vec{y} \preceq \vec{z}\}$$

This operation maps every element of the domain to the lexicographic minimum of its relating range elements. For example, it is:

$$\text{lexmin}(\{\, \mathrm{S}(i,j) \to \mathrm{S}(o,j) \mid -1 + i \leqslant o \leqslant 1 + i \,\}) = \{\, \mathrm{S}(i,j) \to \mathrm{S}(-1+i,j) \,\}$$

The above are all the operations that are necessary to understand the algorithms presented in this thesis. Additionally, the notion of *injectivity* is required:

Definition 2.12 (Injectivity)**.**
Let $M : \mathbb{Z}^p \to 2^{\mathbb{Z}^{d_{in}} \times \mathbb{Z}^{d_{out}}}$ be a map. M is called *injective* if for each $\vec{n} \in \mathbb{Z}^p$ and for each $(\vec{x_1}, \vec{y_1}), (\vec{x_2}, \vec{y_2}) \in M(\vec{n})$ with $\vec{y_1} = \vec{y_2}$, it follows $\vec{x_1} = \vec{x_2}$.

Verbalized, injectivity means that each range element is related exactly one domain element.

2.2.3 Schedule Trees

The union maps from section2.2.1 allow to represent all possible schedules for SCoPs. As can be seen in chapter 3, there exist different approaches to calculate schedules for an entire SCoP. However, when only parts of the schedule need to be modified, the map representation of a schedule is a bit inconvenient. By contrast, the *schedule tree* representation, which is an alternative way to encode the order of statement instances in a SCoP, allows to easily modify parts of a schedule while leaving the rest untouched [58]. The main idea is to keep the original structure of the loop nests and statements of the underlying program while maintaining the abstraction of the polyhedral model. For this, a tree structure is ideal, as a a program consisting of loops and statements has a tree-like structure. A schedule tree consists of the following *node types*:

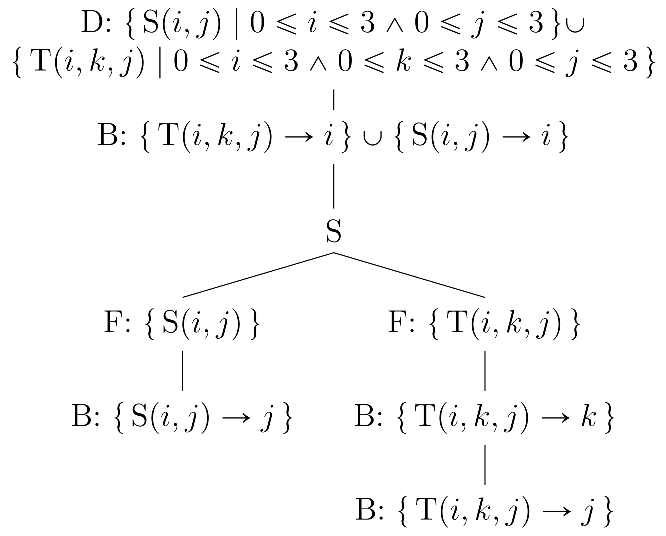

D: $\{ \mathrm{S}(i,j) \mid 0 \leqslant i \leqslant 3 \wedge 0 \leqslant j \leqslant 3 \} \cup$
$\{ \mathrm{T}(i,k,j) \mid 0 \leqslant i \leqslant 3 \wedge 0 \leqslant k \leqslant 3 \wedge 0 \leqslant j \leqslant 3 \}$
B: $\{ \mathrm{T}(i,k,j) \to i \} \cup \{ \mathrm{S}(i,j) \to i \}$
S
F: $\{ \mathrm{S}(i,j) \}$
F: $\{ \mathrm{T}(i,k,j) \}$
B: $\{ \mathrm{S}(i,j) \to j \}$
B: $\{ \mathrm{T}(i,k,j) \to k \}$
B: $\{ \mathrm{T}(i,k,j) \to j \}$

Figure 2.7: This is one possible schedule tree for the SCoP in figure 2.4a. This schedule tree is equivalent to the following schedule map: $\{ \mathrm{S}(i,j) \to (i,0,j,0) \} \cup \{ \mathrm{T}(i,k,j) \to (i,1,k,j) \}$. The 0 and 1 in the second dimension of the output vector are used to represent the second node. The 0 in the forth dimension of the first set is just a filler to let the schedule have a unique, single dimension which is four in this case.

- *Domain node (D):* A domain node always forms the root of a schedule tree and describes the statement instances that are ordered by this tree. If the schedule belongs to a SCoP, then the domain node exactly describes the domain of a SCoP. A domain node always has exactly one child.

- *Filter node (F):* A filter node tells which statements are scheduled by its subtree, i.e., it filters out statements that are scheduled by other parts of the tree.

- *Sequence node (S):* A sequence node describes that its children should be executed in order. It has always at least two childen. The children of a sequence node are always filter nodes.

- *Band node (B):* A band node is a representation of a loop. It describes the order of statement instances in its branch by a schedule map, just like the map representation of a schedule. However, unlike in the map representation of schedules, this schedule usually is a *partial schedule* as it does not have the full dimensionality of the SCoP schedule or does not provide a schedule for all of the SCoP's statements.

- *Leaf node (L):* Each branch of the tree is terminated by a leaf node, which has no function other than indicating termination. In the following, unless it is important, the leaf node is omitted.

To understand how these node types work together, it is best to have a look at an example. For this, let us build the schedule tree for the `gemm` SCoP in figure 2.4a, wich can be found in figure 2.7. As explained, the root of the schedule tree is a domain node. The i loop of the SCoP surrounding both statements translates to a band node that contains a one-dimensional schedule map for both statements. Below that is sequence node as in the program, the remaining loops are nested around a single statement each only, hence they need to be ordered. As the j loop for statement S comes first, the first child of the sequence node is a filter node for S. The other child is the filter node for T. Below the filter node for S resides a single band node that represents the j loop around S. The child of the filter for T is a band node for the k loop. This band's child is another band representing the j loop.

2.2.4 Data Dependency Analysis

This section explains data dependency analysis in a polyhedral context. The terminology is adapted from [49]. Data dependencies in a program describe the order in which variables are accessed in a program. The general problem for computing data dependencies for any program is undecidable. However, in the case of SCoPs and affine array accesses, things turn out to be a bit different. To understand this, it is necessary to distinguish two kinds of dependencies: *memory-based dependencies* and *value-based flow dependencies.* A *memory-based dependency* from statement instance $S(\vec{i})$ to statement instance $T(\vec{j})$ arises when

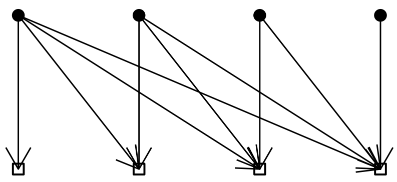
- $S(\vec{i})$ is executed before $T(\vec{j})$, and
- both $S(\vec{i})$ and $T(\vec{j})$ access the same memory location.

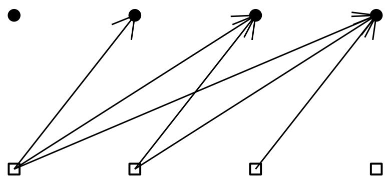
Depending on the type of the memory access, there are several possible combinations: read-after-write (RAW), also called true dependencies, write-after-read (WAR), also called anti-dependencies, write-after-write (WAR), also called output dependencies, and read-after-read (RAR), which is usually of no interest. On the other hand, a *value-based flow dependency* from $S(\vec{i})$ to $T(\vec{j})$ is a memory-based RAW dependency where no write to the shared memory location occurs between the execution of $S(\vec{i})$ and $T(\vec{j})$. The difference is illustrated in figure 2.8.

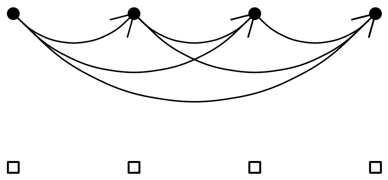
For SCoPs, i.e., programs with a polyhedral representation, memory-based dependency analysis is NP-complete, as it is equivalent to solving an integer linear program [48]. Exact value-based flow dependency analysis is decidable [17], and there exist different algorithms [17, 49, 36] with a

```
for(int i = 0; i <= 3; ++i) {
  for(int j = 0; j <= 3; ++j) {
S:  C[j] = T[i][j]
  }
  for(int j = 0; j <=; ++j) {
T:  A[i] += C[j]
  }
}
```

(a) This represents an arbitrary kernel to demonstrate the different kinds of dependencies.


(b) Memory-based RAW dependencies


(c) Memory-based WAR dependencies


(d) Memory-based WAW dependencies

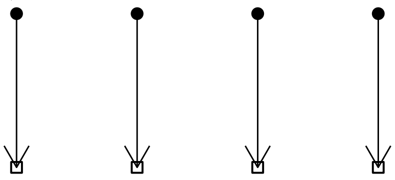
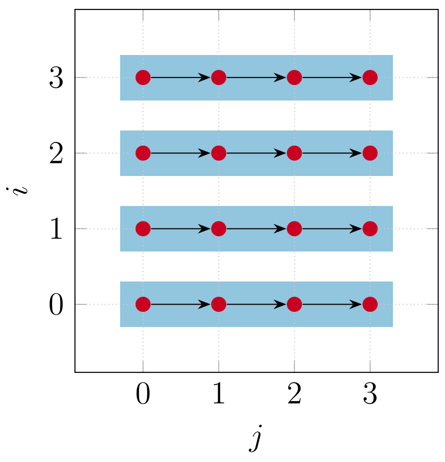

(e) Value-based flow dependencies

Figure 2.8: This shows the different kind of dependencies of `C[0]` through the i loop in 2.8a. The upper row with the dots represents the four statement instances of S where `C[0]` is written (i.e., where $j = 0$), the lower row represents the four statement instances of T where `C[0]` is read (and $j = 0$ again). The figure is adapted from [57, p. 107].

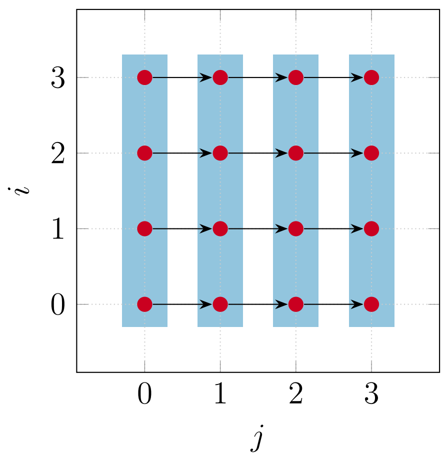
suitable performance for most practical problems. An implementation of a combination of these algorithms is available in `isl` [61].

For this thesis, value-based flow dependency analysis is necessary as it tells whether one is allowed to modify the schedule of a SCoP or not. Memory-based dependency analysis would be too restrictive for that. Only transformations that do not violate all the value-based flow dependencies are valid. Otherwise one would change the semantics of the SCoP. Especially, flow dependencies can be used to analyze whether a loop (or band node in a schedule tree) carries a dependency or not. To carry a dependency means that reordering the statement instances in this loop or band can lead to a violation of some flow dependency. If a loop carries no flow dependency, its elements can be re-arranged or it can be executed in parallel.

To check if a loop or band, represented by a partial one-dimensional schedule map, carries some dependencies or not, one can do the following [23, p. 61]: Assume there is a union map S describing the schedule and a union



(a) This shows a partial schedule that does not carry a dependency. Hence, the four bands can be executed in any order.



(b) This shows a partial schedule that carries a dependency. Thus, the four bands have to be executed from left to right.

Figure 2.9: Both figures shows schedule bands interacting with the dependencies. The points depict the statement instances, the arrows mark the dependencies, and the blocks show the statement instances that belong to the same band.

map D describing all the dependencies. Then calculate the expression:

$$\Delta(S \circ (S \circ D^{-1})^{-1})$$

This gives the distance between the domain and range of the dependencies in the schedule. If this distance is zero, then the schedule S does not carry any of the dependencies in D, because all tuples of dependent statement instances only appear within a single value of the schedule S. If the schedule S describes a band in a schedule tree, this means that all blocks of the band could be executed in parallel. In this case, the band is called *coincident*. If the distance is anything other that zero, then this schedule map does carry some of the dependencies in D, hence S cannot be changed without potentially violating some dependencies. Both cases are illustrated in figure 2.9.

3 Related Work

This section provides an overview of the literature relevant to this work. First, it presents a summary of the polyhedral schedule optimization techniques and their applications in real-world systems. In the second part, it gives an overview of compiler optimizations for memory systems in general and racetrack memories in particular.

3.1 Polyhedral Schedule Optimizations

The polyhedral schedule optimization can be broadly classified into three different categories. First, there are cost-model based approaches that optimize a given cost function. The second group of approaches finds efficient schedules by iteratively searching the entire schedule space. A third group of approaches uses pattern-based optimizations.

3.1.1 Cost-model Based Optimizations

Feautrier was among the first to introduce single- and multi-dimensional polyhedral scheduling methods [18, 19]. He shows how the Farkas lemma can be used to transform the polyhedral scheduling problem into a linear problem (LP). The LP incorporates program dependencies and adds additional constraints to maximize the dependencies that are carried in each dimension of the resulting multidimensional schedule and to minimize the size of the coefficients. The LP is solved through finding the lexicographic minimum of the solutions. Feautrier's work is still relevant today; his scheduler is still used as a fallback [60, p. 20] in the `isl` scheduler, and the theoretical foundation he provides to polyhedral scheduling in general is still the basic theory behind most subsequent work.

The state-of-the-art scheduling algorithm is Pluto [9, 10]. Pluto aims at optimizing parallelism and locality at the same time. The idea behind this is to minimize the distance of dependencies between tiling hyperplanes, i.e., groups of statements that are scheduled together. This distance is used as a cost function. It is not minimized directly, but bounded by an affine form. By applying the Farkas Lemma, one can find a system of linear inequalities that can be solved. From the first solution, additional constraints are added until enough linear independent solutions are found. Experimental results demonstrate that Pluto is able to provide significant speedup for a majority of the tested kernels that are checked for both sequential and parallel execution.

A modification to Pluto is presented in [59]. Verdoolaege and Isoard extend the Pluto algorithm to optimize for consecutive memory accesses, thereby improving the spatial locality. For this, a special constraint based on the affine access function is added that should enforce consecutivity. They present an algorithm to efficiently satisfy this additional constraint. This is somewhat interesting from the perspective of this work as consecutive memory accesses might provide a starting point for further optimizations. However, consecutivity in a random access memory is not the same as minimizing the number of shifts in a racetrack memory, as the latter might benefit from a reversal of the current access pattern to let the port move instead of jumping to the start of the DBC.

3.1.2 Iterative Optimizations

Pouchet et al. apply the Farkas Lemma to the dependency polyhedron to construct the complete one-dimensional schedule space for a given SCoP [45]. Without bounds, this space can still be infinite as there are cases where scaling a solution with an arbitrary integer leads to a new solution. Thus, they put tight constraints on the schedule coefficients to reduce the schedule space to a comprehensible size. All schedules in the reduced space are exhaustively tested with several compilers, taking the performance measurements for each, to find the best program version for each of the compilers, which might be different. They conclude that except in some special cases, their optimization yields significant performance gains. Furthermore, they notice that the optimal schedule can differ a lot between several compilers, as the performance highly depends on which backend optimizations can be applied afterwards. This differs greatly between the compilers. Overall, the optimal schedules are quite complex, and it is difficult to say which part of the schedule is actually responsible for the speedup.

Pouchet et al. extend this approach in [44]. While their first paper [45] only covers one-dimensional schedules, this one constructs a schedule space region of multi-dimensional schedules. As the schedule space is no longer exhaustively enumerable, they introduce a heuristic that restricts the search space to allow scalable traversal. They find that the heuristic increases performance for kernels with few statements, while for a large number of statements, the time limit is hit before good schedules are found. To improve in the latter case, they develop a set of operators for a genetic algorithm that further increases the scalability of the approach. They discover that for kernels with ten statements at most, the heuristic comes close to the genetic algorithm, while for the larger kernels the genetic algorithm provides much higher performance gains.

Ganser et al. aim for tiling and parallelizing the input code [20] by further expanding the previous work [44]. For this, they modify the creation of the schedule space regions to include some with outer loops not carrying dependencies in order to allow for outer parallel loops, and drop the constraints on the coefficients. Furthermore, they use Chernikova's algorithm to calculate the dual representation of a polyhedron to sample the regions. The dual version of a polyhedron is not constraint-based, but instead based on vertices and rays[1].

For sampling, they use both a random exploration and a genetic algorithm. They show that for selected kernels, both their random exploration and their genetic algorithm outperform the approach in [44].

3.1.3 Pattern-based Optimizations

Gareev et al. contribute a specific pattern-based optimization for tensor contraction in [22]. By searching the memory accesses for a specific pattern, they detect tensor contraction and then apply selected optimizations such as tiling, data layout transformations and vectorization, with specific parameters selected for tensor contraction. They show that they can reach the performance of specialized, state-of-the-art BLAS libraries.

The optimization suggested in [22] is implemented in the mainline branch of Polly, but the implementation shows a major drawback of pattern-based algorithms: Even for a simple case like that, the code can get rather verbose. To tackle this, Chelini et al. suggest so-called *loop tactics* [11]. Loop tactics provide a declarative way to describe computational patterns in the polyhedral model and support transformations that use pairs of matchers and builders to implement the pattern-based optimizations. The authors provide an implementation of their suggested framework for LLVM and a source-to-source compiler. They implement the pattern from [22] and a data layout transformation for SIMD code. They find that loop tactics reduce the code complexity and code size of the optimizations in the compiler. They further

[1]This is due to the Farkas-Minkowski-Weyl theorem and steps a bit deeper into the mathematics behind the polyhedral model, which is not the topic of this work. For further read on duality, see[51, p. 87ff.]

incorporate a replacement of code with vendor-optimized library routines for special kernels like BLAS.

The loop tactics framework is exactly what is needed to make the pattern-based optimizations explored in this thesis easier to implement. However, the paper is too recent to incorporate its framework into the implementation for this thesis. Future work to improve what is shown in this thesis should, however, use the loop tactics framework instead of the manual implementation.

3.1.4 Use Case Specific Optimizations

As this thesis targets the specific use case of applying polyhedral optimization techniques to minimizing the shifts in racetrack memories, it is of interest to see other areas where specific use cases were tackled using polyhedral techniques.

As polyhedral schedule optimizations often aim at parallelizing existing code, it is a natural fit for GPUs. An automatic c-to-cuda code generation is presented in [4]. PPCG [2] is another source-to-source compiler to map sequential programs on a modern GPU, and is improved upon in [52]. Polyhedral process networks are mapped onto GPUs in [2]. Another use case for polyhedral optimization techniques is tensor contraction. Next to the aforementioned pattern-based optimization [22], the polyhedral model is applied to optimize tensorflow computation graphs [47] and for optimizing tensor operations on computing-in-memory architectures [16]. Similar to tensor operations, stencils are another kind of use case that fit into the polyhedral model. They are automatically tiled for parallelism in [3], and automatic time-tiling is proposed in [8]. Specialized platforms, like FPGAs [46] and systolic arrays [15], also benefit from polyhedral optimization techniques.

3.1.5 Summary

In the aforementioned cost-model based and iterative algorithms, a shared core idea is to utilize the Farkas Lemma. The pattern-based techniques are complementary to those, and can be applied after the other two or as standalone. However, they are more likely to work well with the cost-model based approaches than the iterative ones. The reasons for this are as follows: The resulting schedules are sometimes very complex and might remove the ability to detect any patterns in those. Furthermore, the iterative approach requires to rate the code by a metric that is computed from the actual behaviour of the code, thus running different versions of the code repeatedly. Hence, for this thesis, it was chosen to combine existing cost-model based approaches with a new pattern detection.

3.2 Shift Optimizations in Racetrack Memories

This section provides an overview of software optimization techniques for shifts migitation in racetrack memories. There are various hardware solutions to reduce the shifts in RTMs, however, as this thesis only investigates software-based optimizations, they are beyond the scope of this work.

An important predecessor to the shift optimization problem for racetrack memories is given by Liao et al. [34]. They target digital signal processors (DSPs) for their optimization. DSPs store their data in stacks which are usually addressed by one or more address registers. Incrementing or decrementing the address register by one can be done in the same operation as a load of the address, whereas moving the address further in the stack requires an extra operation. Hence, the problem of placing variables in the DSP stack is similar to the problem of placing variables in DBCs, as in both cases, one does ideally want to have movements of size one.

The main goal in [34] is to minimize the code size by maximizing the number of postincrement and postdecrement operations for address generation. As an added bonus, this also benefits performance. For this, the authors define the simple offset assignment problem (SOA) involving only a single address register, and the general offset assignment problem (GOA) with an arbitrary number of address registers. They show, by reduction to the maximum weight path covering problem, that SOA is NP-complete. Hence, they present a heuristic for SOA and utilize this heuristic in a second heuristic to tackle GOA. They find that their heuristic can reduce the number of address instructions by around 46 %. Analogously to DSPs, SOA and GOA can be defined for RTMs by either looking at a single or multiple DBCs.

Chen et al. claim to be the first ones who proposed a software-based approach to address shift migitation in racetrack memory [12, 13]. They show that the problem of optimal variable placement for minimal shifts in racetrack memories is NP-complete. They introduce a grouping-base data placement algorithm and compare it to an ILP solver, a placement arranged according to first variable occurrence, and the SOA algorithm with modified cost function from [34]. They find that their heuristic greatly outperforms SOA and no-op, and is on par with the ILP solution.

Another approach to SOA for RTMs is presented by Mao et al. in [35]. They suggest three simple heuristics: either sort the variables according to first access, place the variables most frequently accessed in the middle of the racetrack, or place the variables most frequently accessed first in the racetrack. Additionally, they suggest a genetic algorithm to solve the problem, which is further enhanced by using the simple heuristics for some of the initial values. They find that this combination can find near-optimal solutions.

Khan et al. propose a third way to tackle SOA in [29] called *ShiftsReduce*.

Based on existing heuristics for data placement on a DSP, they develop a heuristic for racetrack memory that improves on [13] by considering temporal locality of accesses. They, using the OffsetStone benchmark, show that their heuristic improves on the existing ones significantly. In addition, they use a genetic algorithm that provides near-optimal solutions. Finally, they show that the reduction in shifts is directly connected to a reduction in energy consumption.

While ShiftsReduce focuses on intra-DBC variable placement, in [27] Khan et al. present a heuristic that optimizes the inter-DBC data placement. The main idea is to calculate the live-ranges of each variable and to group variables that have disjoint life-ranges in their access order together. In addition, they present a genetic algorithm that should solve both the inter and intra-DBC data placement at once to have a near optimal solution as a baseline. Alongside their approach, they evaluate the inter-DBC placement strategy by Chen et al. [12, 13]. They find that their heuristics likely performs within a magnitude of the optimal solution, as a comparison with the genetic algorithm shows.

Shift reduction in arrays is addressed in [30] by Khan et al. More specifically, they present an optimized memory layout and an optimized schedule to reduce the number of shift operations in tensor contractions. Furthermore, they add preshifting that proactively moves data to ports to hide the shift latency. They find that the combined effect of their optimizations give RTMs a competitive edge over SRAM in both performance and energy consumption. Basically, this paper motivates and provides a starting point for this thesis. It shows that optimizing the memory access pattern of arrays plays an important role in utilizing racetrack memory for performance and energy efficiency. However, Khan et al. suggest manual transformations and tackle only a single kernel, namely tensor contraction, while this work presents an automated end-to-end compilation framework that optimizes schedules and layouts for RTMs.

Multanen et al. present a strategy to place instructions in RTM to reduce the shifts [39] alongside an instruction fetch and memory architecture. The core idea is to split each basic block into two pieces, revert the second half of the basic blocks, and put the reverted half after the first in the DBC. Two access ports are used to utilize this instruction memory layout. This architecture and strategy achieves a significant reduction in number of shifts and cycle count in all 12 evaluated benchmarks.

To the best of the author's knowledge, this thesis is the first work that explores polyhedral optimizations for optimizing memory access patterns in racetrack memories.

4 Optimizing Compilers for Racetrack Memories

This chapter explores schedule and layout transformations to achieve minimal-offset locality in RTMs. First, a distinction of shifts into compulsory and overhead shifts is provided. Second, schedule transformations that can reduce the shifts are explained. Third, different schedule optimization techniques that can perform the aforementioned transformations are explored, including a novel pattern-based schedule transformation algorithm. Forth, a novel pattern-based layout transformation algorithm to optimize the memory layout of stencil kernels is suggested. Last, the integration of the optimization techniques into the LLVM framework is explained.

4.1 The Overhead Shifts

RTMs are sequential by design when one wants to use them to their full capacity. Even in the best case when accessing neighboring memory locations, ports need to be moved to the next locations, thus incuring a shift of one. However, there are cases when the ports are only moved to reset them to the first location in their respective DBC. Between each of these multiple shifts, the ports are not doing something useful. This allows to classify shifts into two categories, *compulsory shifts* and *overhead shifts*. *Compulsory shifts* are the ones that are required due to the nature of the RTMs and the memory layout and access pattern of the program. *Overhead shifts* are the shifts that are used to reset a port to its starting location, usually to the first position, and which cause a high latency and can potentially be avoided by some

```
for(int i = 0; i <= 63; ++i)
  A[i] = 1;
```

Figure 4.1: This shows simple program that sets every value of an array to one and has no overhead shifts.

```
for(int i = 0; i <= 3; ++i)
  for(int k = 0; k <= 3; ++k)
    for(int j = 0; j <= 3; ++j)
T:    C[i][j] = C[i][j] + α * A[i][k] * B[k][j];
```

Figure 4.2: This figure shows a simplified `gemm` kernel. It is a shorter version of figure 2.4a. It has overhead shifts in the arrays `B` and `C`.

program transformation. For example, consider the program in figure 4.1. With the default memory layout, the array `A` fits entirely into one DBC of the RTM. Assuming that the racetracks in DBCs have a single port and it points to the index zero, the porgram incurs 63 shifts in total. This number, when using a single DBC, is optimal, as the port has to reach every single entry of the DBC, hence it has to be moved at least 63 times. This can be generalized a bit more: When the ports of all DBCs point to the first entry each and one only accesses each array cell at most once in increasing order, there should be no overhead shifts. Hence, when only a single access to each memory cell occurs, there is little room for optimization.

This changes, however, when there are multiple accesses to the same memory location in a program. For this, let us have a closer look at the T statement of the `gemm` kernel again, which can be found in figure 4.2. Especially, we want to investigate the array `B`, which is a 4×4 array. With the default layout, `B` requires 4 DBCs. Again, we assume that all ports point to the first entry in each DBC. For $i = 0$, as j increases from 0 to 3, every access requires a single shift in each of the four DBCs However, in the second iteration of the i loop, the same access pattern is used, but now the ports point to the 4th entry of each DBC. Hence, they have to be moved back to the first entry again, which requires 3 *overhead shifts* per DBC. The same happens again with $i = 2$ and $i = 3$. This process is demonstrated in figure 4.3. The reason for these overhead shifts is that each entry of `B` is accessed more than once, and in between these accesses, other entries of `B` in the same DBC are accessed. Without changing the semantics of the original program, there are two ways to deal with this:

- One can change the schedule of the original program without violating the value-based flow dependencies, or
- one can change the default memory layout of the array in the RTM to a tailored one for the specific program.

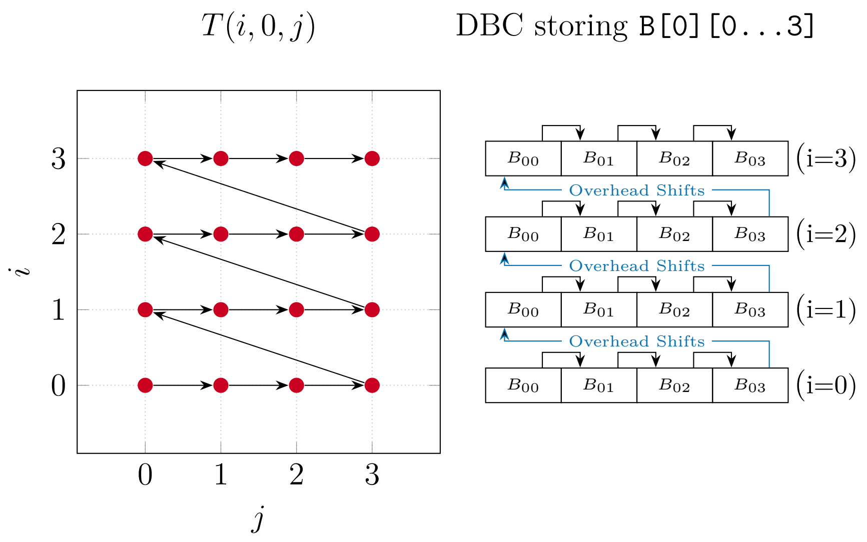


Figure 4.3: This demonstrates the overhead shifts of `B` in figure 4.2. On the left, the statement instances $T(i, 0, j)$ are shown, i.e., for the k loop, only $k = 0$ is viewed, as $k > 0$ accesses a different part of `B` stored in a different DBC. The arrows depict their schedule. On the right, the access sequence of the port of the DBC that stores `B[0][0...3]` is shown. `B[0][j]` is abbreviated as B_{0j}. There are 12 compulsory shifts and 9 overhead shifts occuring.

4.2 Possible Schedules Avoiding Overhead Shifts

This section provides three examples for schedule transformations that can mitigate the overhead shifts in RTMs. These transformations are loop alternation, loop fusion and loop interchange.

One possibility for a schedule transformation to reduce the overhead shifts suggested by Khan et al. in [30] is to alternate some of the loops. When considering the array `B` again, this means that in the second iteration of i, the j loop is not incrementing from zero to three, but starting at three and moving back to zero. This means that the backwards shifting of the ports pointing to `B` is not an overhead, but each shift is used to read a value of `B`. In $i = 2$, j maintains its original order, while for $i = 3$, j has to be reverted again. The code for this can be found in figure 4.4a, and a graphical representation of this can be found in figure 4.5. This transformation of the loop order from uni- to bi-directional is called the *alternation transformation*, which improves a special kind of the spatial locality that is called the *minimal-offset locality* in this thesis.

Another way to reduce RTM shifts is to group together accesses to the exact same array cell. For this, we have to look at the example code in

```
for(int i = 0; i <= 3; ++i)
  for(int k = 0; k <= 3; ++k)
    if(i % 2 == 0)
      for(int j = 0; j <= 3; ++j) // forward iteration
T:      C[i][j] = C[i][j] + α * A[i][k] * B[k][j];
    else
      for(int j = 3; j >= 0; --j) // backward iteration
T:      C[i][j] = C[i][j] + α * A[i][k] * B[k][j];
```

(a) This figure shows the transformed version of 4.2 that reduces the overhead shifts of `B` by alternating the j loop based on i.

```
for(int i = 0; i <= 3; ++i)
  for(int k = 0; k <= 3; ++k)
    if((i % 2) + (k % 2) != 1)
      for(int j = 0; j <= 3; ++j) // forward iteration
T:      C[i][j] = C[i][j] + α * A[i][k] * B[k][j];
    else
      for(int j = 3; j >= 0; --j) // backward iteration
T:      C[i][j] = C[i][j] + α * A[i][k] * B[k][j];
```

(b) This figure shows the transformed version of 4.2 that reduces the overhead shifts of both `B` and `C` by alternating the j loop based on i and k.

Figure 4.4: This shows two optimized versions of the `gemm` kernel.

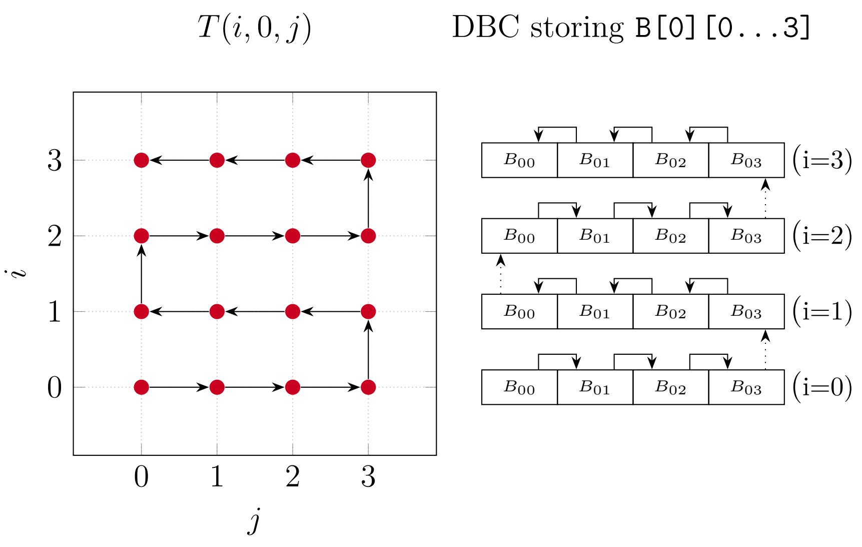


Figure 4.5: This showcases the benefits of the alternation transformation in figure 4.4a. On the left, the statement instances $T(i, 0, j)$ are shown, i.e., for the k loop, only $k = 0$ is viewed, as $k > 0$ accesses a different part of `B` stored in a different DBC. The arrows depict their schedule. On the right, the access sequence of port of the DBC that stores `B[0][0...3]` is shown. `B[0][j]` is abbreviated as B_{0j}. When compared to figure 4.3, one can see that the overhead shifts are avoided.

```
for(int i = 0; i <= 3; ++i)
S:  B[i] = 2 * A[i];
for(int i = 0; i <= 3; ++i)
T:  C[i] = 3 * A[i];
```

(a) This is a sample code where the shifts of `A` can be reduced through loop fusion.

```
for(int i = 0; i <= 3; ++i) {
S:  B[i] = 2 * A[i];
T:  C[i] = 3 * A[i];
}
```

(b) This shows the transformed code where loops are fused.

Figure 4.6: This shows a code example where the shifts are reduced through loop fusion.

```
for(int i = 0; i <= 3; ++i) {
  for(int j = 0; j <= 3; ++j) {
S:  B[i][j] = A[j];
  }
}
```

(a) This is a sample code where the shifts of `A` can be reduced through loop interchange.

```
for(int j = 0; j <= 3; ++j) {
  for(int i = 0; i <= 3; ++i) {
S:  B[i][j] = A[j];
  }
}
```

(b) This is the sample code with the interchanged loop nest.

Figure 4.7: This shows a code example where the shifts are reduced through loop interchange.

figure 4.6. The array `A` is accessed twice, once in each loop. As in the previous paragraph for `B`, there are overhead shifts when moving the port of `A` back to the first entry when executing the statement instance $T(0)$. We can of course revert the loop surrounding T to remove those overhead shifts. However, another possibility is to fuse both loops together which, at least for this kernel, requires even fewer shifts, as each `A[i]` is now accessed twice before the access port is moved to the next location. This example illustrates that loop fusion can also be effective in reducing the number of shifts. An effect similar to loop fusion can sometimes be achieved by loop interchange as demonstrated in figure 4.7. By exchanging the j and the i loop, the shifts for accessing `A` are minimized as `A` is only processed once instead of once per iteration of i.

4.3 Schedule Transformation Techniques

This section examines the availabe polyhedral scheduling techniques summarized in section 3.1 in the context of optimizing shifts in racetrack memories.

Two of the commonly known polyhedral schedule optimization techniques are based on the Farkas Lemma, compare section 3.1.1 and 3.1.2. Hence, let us have a look at the Farkas Lemma in its affine form [18, p. 328][51, p. 92ff.], adapted from [45, p. 4].

Theorem 4.1.
Let $Q = \{\vec{x} : A\vec{x} + \vec{b} \geqslant \vec{0}\}$ be a polyhedron and let $f : Q \to \mathbb{Z}$ be an affine function. Then $f(Q) \geqslant 0$ if and only if it is a positive affine combination:

$$f(\vec{x}) = \lambda_0 + \vec{\lambda}^T(A\vec{x} + \vec{x}), \text{ with } \lambda_0 \geqslant 0 \text{ and } \vec{\lambda}^T \geqslant \vec{0}$$

λ_0 and $\vec{\lambda}^T$ are called Farkas multipliers.

We can use one-dimensional schedules to demonstrate the core idea of scheduling using the Farkas Lemma. For this, consider the dependency polyhedra from 4.2:

$$\{\, \mathrm{S}(i,j) \to \mathrm{T}(i,k,j) \,\} \tag{4.1}$$

and

$$\{\, \mathrm{T}(i,k,j) \to \mathrm{T}(i,1+k,j) \,\}.$$

Let us now assume that we want to construct affine one-dimensional schedules for S and T. The prototypical schedules have the form:

$$\{S(i_S, j_S) \to (s_1 \cdot i_S + s_2 \cdot j_S + s_0)\} \cup \{T(i_T, k_T, j_T) \to (t_1 \cdot i_T + t_2 \cdot k_T + t_3 \cdot j_T + t_0)\}.$$

They can also be represented as two affine functions of the statement instances:

$$\theta_S(i_S, j_S) = s_1 \cdot i_S + s_2 \cdot j_S + s_0$$

and

$$\theta_T(i_T, k_T, j_T) = t_1 \cdot i_T + t_2 \cdot k_T + t_3 \cdot j_T + t_0.$$

These schedule functions need to fulfil the dependencies. This means that, on every point $(i_S, j_S, i_T, k_T, j_T)$ in the dependency polyhedron 4.1, the following has to hold to satisfy the dependency:

$$\theta_T(i_T, k_T, j_T) > \theta_S(i_S, j_S) \Leftrightarrow \theta_T(i_T, k_T, j_T) - \theta_S(i_T, j_T) - 1 \geqslant 0$$

Now one can apply the Farkas Lemma on the affine form

$$\theta_T(i_T, k_T, j_T) - \theta_S(i_T, j_T) - 1$$

as the dependency is given by a polyhedron. With this, one obtains constraints on the schedule coefficients $s_0, s_1, s_2, t_0, t_1, t_2, t_3$ involving the Farkas multipliers. This can be done for the other dependency polyhedron as well. In the end, one gets a system of linear inequalities with the Farkas multipliers and the schedule coefficients. All coefficients and multipliers that satisfy this system form a valid schedule. This system of linear inequalities is used both in the cost-model based approaches from section 3.1.1 as well as in the iterative approaches from section 3.1.2.

This method has a slight disadvantage in the sense that it limits the schedule space to affine transformations of the original schedule. However, not all desirable transformations are affine. Tiling is one example of a transformation that is non-affine, and is often applied as a post-processing to the schedulers in sections 3.1.1 and 3.1.2. The *alternation transformation* presented in section 4.2, is also non-affine, as is shown in the following. For this, assume that there exists an affine schedule function $f : \mathbb{Z}^2 \to \mathbb{Z}, (i, j) \to f(i, j)$ with the following properties:

$$0 < f(i, j + 1) - f(i, j) \text{ if } i \text{ is even,}$$

as, for even i, (i, j) is executed before $(i, j + 1)$, and

$$0 > f(i, j + 1) - f(i, j) \text{ if } i \text{ is odd,}$$

as, for odd i, (i, j) is executed after $(i, j + 1)$. As f is affine, it is

$$f(i, j + 1) - f(i, j) = f(0, 1).$$

With both inequalities from above, it follows:

$$0 < f(0, 1) \text{ and } 0 > f(0, 1),$$

a contradiction. Hence, f either does not have the desired properties or is not affine. Thus, the alternation transformation cannot be performed by cost-model based or iterative schedule optimization algorithms. Hence, a different approach for the alternation transformation is required. As they are already used in practice for other specific use-cases, pattern-based transformations look like a promising candidate for this kind of transformation, especially since this transformation is local to loops and can be applied to an existing schedule.

When looking at the example in section 4.2, the main idea of the alternation can be summarized as follows:

> Whenever the same array indices are accessed more than once in different statement instances of the same statement, alternate the loop that occurs in the fastest-changing array index.

Before going into the details of the algorithm, we look at a useful definition:

Definition 4.1 (Equal memory access indices).
Two memory access maps M_1 and M_2 with $\text{Space}(M_1) = \text{Space}(M_2)$ have *equal memory access indices* if, for each statemement instance, the difference between the two memory location vectors is a constant vector. More formally, M_1 and M_2 of S have *equal memory access indices* if $\Delta((M_2 \circ M_1^{-1})^{-1})$ contains only one element.

Note that this forms an equivalence relation, i.e., it is reflexive, symmetric, and transitive.

For example, the accesses `A[i][j]`, `A[i][j+1]` and `A[i-1][j-1]` all have equal memory access indices, while all pairs of `A[i][j]`, `A[2i][j]`, `A[j][i]` do not have equal memory access indices.

The following describes the implementation of this idea into a proper pattern-matching algorithm on a SCoP.

The algorithm can be split into two parts and an additional preprocessing step. The preprocessing ensures that the schedule tree meets certain requirements, the first part extracts all necessary information from the memory accesses, while the second part uses this knowledge to find and transform the appropriate nodes in the schedule tree.

Let us first have a look at the preprocessing, which ensures that:

- The schedule tree must not contain bands with more than one dimension. To get this, we traverse the schedule tree and split each band with more than one dimension into multiple band nodes having one dimension each, without changing the schedule order.

- Each band in the schedule tree that does not carry any dependency needs to be marked as coincident. This can either be done manually, as described in section 2.2.4, or it may be there anyways as an output of the `isl` scheduler.

The algorithm can be found in algorithm 1. It is executed for each statement of the SCoP. It gets provided with the statement name as *stmtId*, the list of all memory accesses as *memoryAccesses*, and the schedule tree as *scheduleTree*. It returns the modified schedule tree.

While explaining the algorithm, let us look at the code from figure 4.2 as an accompanying example.

The list of memory accesses contains an integer map for each array access in the code. The memory accesses belonging to *stmtId* are grouped by the different array names (see line 2) into a single integer map. Next, the algorithm iterates through those groups (see line 3). For our example, we get three integer maps for the arrays `A`, `B`, and `C`. In the following, we will

Algorithm 1 This shows the algorithm that detects and performs the alternation transformation on a single statement.

```
1: procedure OPTIMIZESTATEMENT($stmtId, memAccesses, scheduleTree$)
2:   $G$ = group $memAccessess$ of $stmtId$ by name
3:   for $g \in G$ do
4:     if $g$ is not injective then
5:       M = group $g$ by equal memory access indices
6:       for $m \in M$ do
7:         $m_{fixed}$ = fix uninvolved dims of $m$ to smallest
8:         $L = \text{lexmin}(m_{fixed}^{-1})$
9:         $D = \Delta((L \circ m)^{-1})$
10:        $AltBaseDims$ = get alternation base candidates from $D$
11:        $AltLoopDims$ = get alternation loop candidates from $m$
12:        if $AltLoopSet$ has exactly one element then
13:          $Leaf$ = find leaf of $stmtId$ in $scheduleTree$
14:          $AltBand$ = FINDBANDNODEABOVE($Leaf$, $AltLoopDims$, {F, D})
15:          $AltBaseBand$ = FINDBANDNODEABOVE($AltBand$,
               ↪ $AltBaseDims$, {D})
16:          if $Leaf$ has single statement and $AltBand$ exists and
               ↪ $AltBand$ is coincident and $AltBaseBand$ exists then
17:            scheduleTree = alternate $AltBand$ based
               ↪ on $AltBaseBand$ in $scheduleTree$
18:          end if
19:        end if
20:      end for
21:    end if
22:  end for
23:  return $scheduleTree$
24: end procedure
25: procedure FINDBANDNODEABOVE($Node$, $Dims$, $NodeTypes$)
26:   while type of $Node$ is not in $NodeTypes$ do
27:     $Node$ = parent of $Node$
28:     if $Node$ is of type B then
29:       Dim = get dim of $Node$
30:       if $Dim \in Dims$ then
31:         return $Node$
32:       end if
33:     end if
34:   end while
35:   return $\epsilon$
36: end procedure
```

only look at the iteration for the array B. It has the following single access relation which is the already grouped set g:

$$g := \{\, \mathrm{T}(i,k,j) \to \mathrm{B}(k,j) \,\}$$

Next, the grouped set g is checked for injectivity (line 4). This injectivity check represents the core idea of the algorithm. Injectivity for memory access maps means that whenever the same index vector is accessed in an array, this access happens only in a single statement instance. In other words, this means that no single array cell is accessed by more than one statement instance. Hence, it is likely that one does not get a significant benefit from the alternation transformation. Otherwise, in case the memory access map is not injective, then there are cells that are accessed by multiple statements. In our example, the injectivity check of g yields that it is note injective. For example, both $S(0,0,0)$ and $S(1,0,0)$ map to $B(0,0)$. Thus, we found a potential candidate for our optimization.

Next, the access map for a single array is split into smaller groups of equal memory access indices (compare line 5), which does nothing for our example, as g consists of a single access map anyways, thus $M = g$.

After this, we check each of those groups individually for optimization. For this, we need to find the first statement that accesses each memory cell in the original schedule. If the memory access group relation is called m (see line 6), this can be computed by $\mathrm{lexmin}(m^{-1})$. However, this might fail if there are statement parameters that are not involved in the memory accesses at all. These need to be fixed to the minimum of that statement parameter in m, respectively, before calculating $L := \mathrm{lexmin}(m_{fixed}^{-1})$. Note that the minimum of the statement parameters is usually zero, but can be different if the corresponding loop does not start at zero. For our example, this yields:

$$L = \mathrm{lexmin}(m_{fixed}^{-1}) = \{\, \mathrm{B}(i_0, i_1) \to \mathrm{T}(0, i_0, i_1) \,\}$$

Next, we calculate the relative offset of the first access to all subsequent accesses with $D := \Delta((L \circ m)^{-1})$ (compare line 9). This set contains one element per distinct memory access to the same cell, thus it captures the pattern of the accesses. For our example, D becomes:

$$D = \Delta((L \circ S)^{-1}) = \{\, \mathrm{T}(i,0,0) \,\}$$

This can be used to check which statement parameters are responsible for a repeated access as follows (cf. line 10): Iterate through all set dimensions of D. Let us name the current dimension i. Project out the i-th dimension of D and see whether the new arising set is not equal to D any more. The projection can be achieved by fixing the i-th set dimension to zero as the vector $\vec{0}$ is always contained in D. From this we get a set that contains all the statement parameters that cause multiple accesses. These are potential candidates that can be used as a base for alternation, i.e., *alternation base candidates*. For our example, a potential alternation base candidate is the i loop of the statement $T(i, k, j)$, as when fixing its dimension to zero, the set D becomes:

$$\{\, \mathrm{T}(0, 0, 0) \,\},$$

In the other two dimensions, D does not change when fixing those to zero, so these are no alternation base candidates.

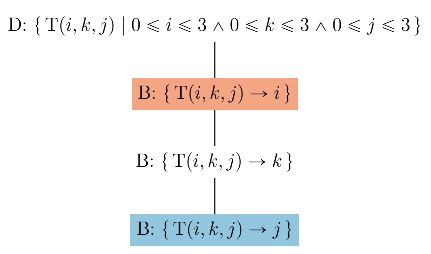
Next, we need the loop that is to be alternated, namely the *alternation candidate*. The following explains line 11. For finding the alternation candidate, take m^{-1}, drop all the constraints involving the input, i.e., the memory dimensions, except the last one, fix all statement parameters that are not affected by any constraint to zero, drop the constraints of the last input dimension, and then take all the indices of the range of the result that are not equal to zero. For our example, the process described above yields the set:

$$\{\, \mathrm{T}(0, 0, j) \,\}$$

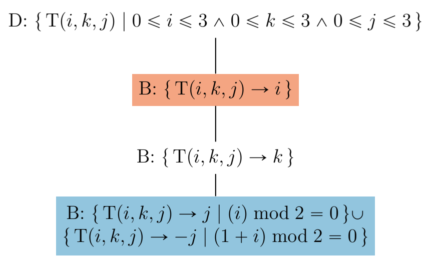
Here, only the third dimension related to the j loop is not zero.

From what is described in the previous paragraph, we get a list of statement parameters that occur in the fastest-changing dimension of the memory access relation m, which is the dimesion that is consecutively put into the DBCs. If there is only a single parameter, then this statement parameter corresponds to the loop we want to alternate. If not, i.e., if there are multiple statement parameters present that are involved in the fastest-changing dimension of m, it is unlikely that the alternation transformation is beneficial. This is due to the fact that there multipe statement parameters in the fastest-changing dimension, then the movement of the DBC ports is affected by multiple loops at once. Thus, for our example, the j loop is an alternation candidate, as it is the only statement parameter that occurs in the fastest-changing dimension of m.

Additionally, we have a list of alternation base candidates. We now first search the schedule tree for the band node related to the alternation candidate. However, this node has to have two properties: First, it has to be marked as coincident (cf. section 2.2.4). Second, it is only allowed to schedule *stmtId*, not any other statement. Hence, one can implement this by going from the leaf of the statement up to the first filter node in the schedule tree (or domain node, if there is no filter node at all). If the band is found, one can continue to traverse the schedule tree upwards to find one of the alternation base candidates calculated previously. This is achieved in the algorithm by two consecutive calls to `findBandNodeAbove` in lines 14 and 15. For the alternation base band, no further restriction apply as it is left unchanged. For our example, the bands for the alternation candidate and the alternation base candidate are marked in figure 4.8a.



(a) This is the schedule tree for the code in figure 4.2.


D: { T(i, k, j) | 0 ⩽ i ⩽ 3 ∧ 0 ⩽ k ⩽ 3 ∧ 0 ⩽ j ⩽ 3 }
B: { T(i, k, j) → i }
B: { T(i, k, j) → k }
B: { T(i, k, j) → j | (i) mod 2 = 0 } ∪ { T(i, k, j) → −j | (1 + i) mod 2 = 0 }

(b) This is the schedule tree for the code in figure 4.4a.

Figure 4.8: This shows the schedule tree for the code in figure 4.2 before and after the alternation transformation, for which the code can be found in figure 4.4a. The alternation base candidate and the alternation candidate are highlighted each.

When both bands are found, we can alternate the one based on the other. This is done by splitting the band's schedule into two parts with distinct domains, one for the iteration where the selected alternation candidate band is even and one where it is odd. For the part where the alternation candidate band is even, we use the same schedule as for the original band. For the odd part, the original schedule is reverted. This achieves the alternation transformation. For our example, the transformed schedule tree can be found in figure 4.8b.

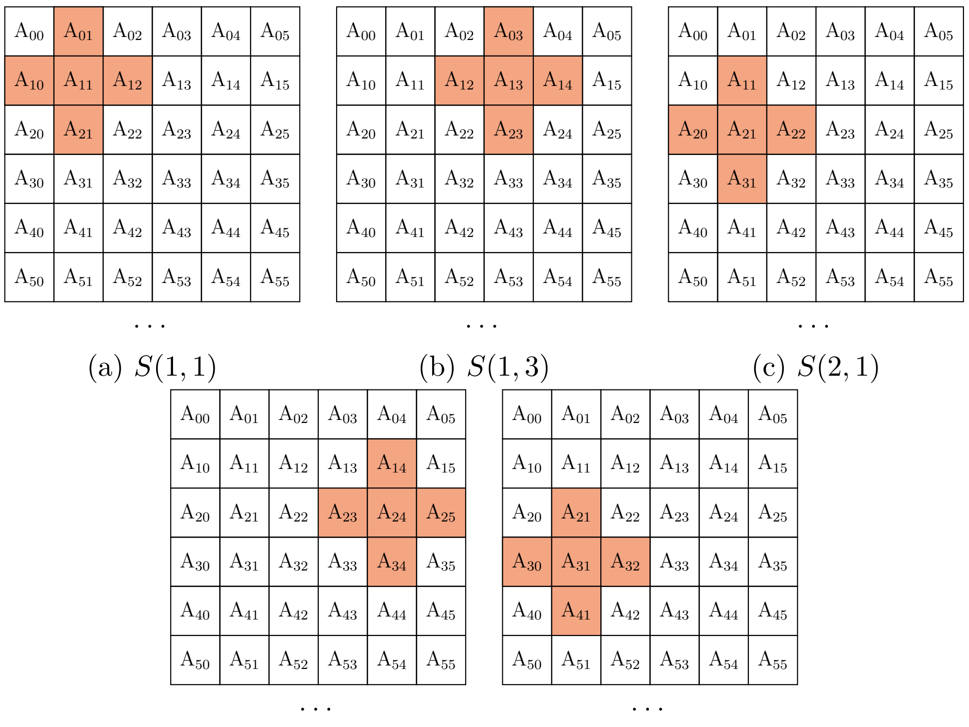
4.4 Layout Transformation

As explored in section 4.3, different schedule transformation techniques can be used to reduce the shifts in applications using racetrack memories. However, sometimes it might not be possible to change the schedule as much as needed, for example, if there are dependencies blocking the desired transformation. Hence, it is promising to investigate another possibility to reduce the shifts, that is, changing the layout of the array in the racetrack memory. Until this point, the default layout described in section 2.1 was assumed. While this is an easy solution that allows for simple continuous access when accessing the entire array once, it is not optimal for other memory access patterns.

To demonstrate this, look at a special kind of programming kernel called *stencil*. A stencil is a program where an array is updated iteratively by some fixed pattern. For example, they are useful in the domains of image and video processing, computation of fluid dynamics, and weather forecasting models. An example for a stencil code can be found in figure 4.9. The special memory access that occurs in stencils is that, unlike `gemm` where the number of accesses to a memory location is matrix size dependent, the number of

```
for(int i = 1; i <= 10; ++i) {
  for(int j = 1; j <= 4; ++j) {
S:  B[i][j] = A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1];
  }
}
```

Figure 4.9: This shows an example stencil processing a 12×6 array with a 5 points stencil.



(a) $S(1,1)$ (b) $S(1,3)$ (c) $S(2,1)$ (d) $S(2,4)$ (e) $S(3,1)$

Figure 4.10: This example demonstrates the repeated memory accesses for the array `A` in figure 4.9. Five statements are depicted that are executed in the order from left to right and top to bottom, with some missing statements in between. In each graphic, `A[x][y]` is depicted as A_{xy}. One can see that `A[2][1]` is accessed in $S(1,1)$, $S(2,1)$, and $S(3,1)$, while other array elements of the same row are accessed in between, namely `A[2][3]` in $(1,3)$ and `A[2][4]` in $S(2,4)$, for example. After the execution of $S(3,1)$, `A[2][1]` is not accessed any more.

accesses to a memory location in stencils is always fixed. This access pattern is demonstrated in figure 4.10.

The idea for a transformed layout for stencils is the following:

> Shift to the next element only when the current location is not accessed any more.

To achieve this, one can transform the memory layout of the array in the

DBC. Instead of storing the fastest-changing dimension in the DBC, it is put into consecutive DBCs. This has to be done for as many second innermost array entries as the stencil reads at the same time. This ensures that the domains are only shifted once their value was used completely for this stencil. For the stencil in figure 4.9, the number of second innermost dimensions accessed at the same time is three (i, $i-1$, and $i+1$), as demonstrated in figure 4.10, so we need to store three of the second innermost dimensions in the first DBC entry each. Figure 4.11 illustrates the original and the transformed layout. The example shows that the layout transformation does not necessarily preserve the number of DBCs. In this case, the number of DBCs is reduced, however, depending on the size of each DBC, it can also increase. This is highly dependent on the array sizes.

The described layout transformation only makes sense for arrays with at least two dimensions. Define the following layout transformation function for the two dimensional case:

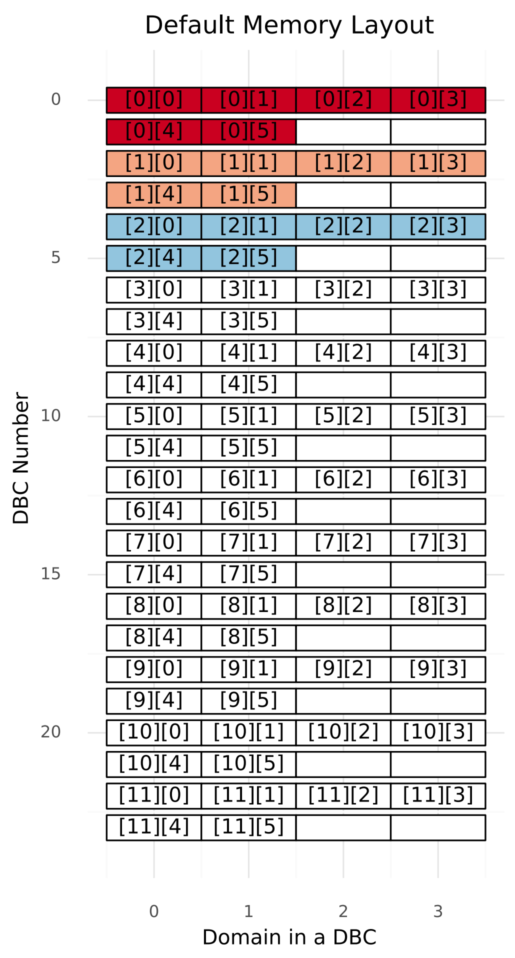
Definition 4.2 (2D stencil layout transformation)**.**
Let s be the size of a stencil, let r be the size of a DBC, and let $a \times b$ be dimensions of a two dimensional array. Define the the *2D stencil layout transformation* T as follows:

$$\begin{aligned}
&T : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z},\\
&T(v, w) = (x, y), \text{ where}\\
&x = b \cdot (v \bmod s) + (b \cdot s \cdot (\left\lfloor \frac{v}{s \cdot r} \right\rfloor)) + w,\\
&y = \left\lfloor \frac{v}{s} \right\rfloor \bmod r
\end{aligned}$$

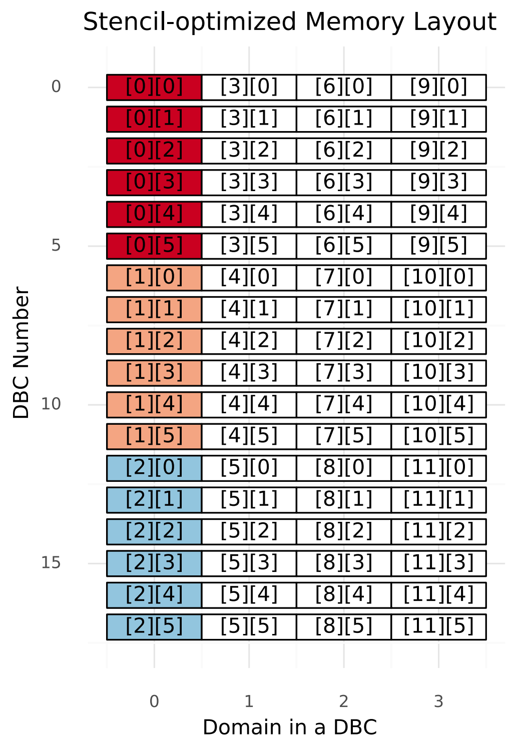
This transformation function provides the exact layout of a two-dimensional array in an RTM, where x represents the DBC and y represents the domain of the DBC. For higher-dimensional values, one can put each two-dimensional chunk one after another.

An important question is how to detect where this layout transformation might be beneficial. For this, similar to the schedule modification in section 4.3, take a pattern-based approach. First identify a stencil-like access pattern and then change the memory layout for this array according to the transformation function.

Similar to the scheduling algorithm, this is done this statement by statement. First of all, we consider only memory accesses for arrays with at least two dimensions. Algorithm 2 is then used to analyze the statement, and takes the statement name as *stmtId*, the list of memory accesses with at least two dimensions as *memAccesses*, the schedule tree as *scheduleTree* and the size of a DBC as *dbc*. We start by grouping the memory access maps of a statement into the same integer map by name (compare line 2). This is exactly the same as line 2 in algorithm 1. Each of these groups is checked in-



(a) This shows how `A` from figure 4.9 is stored in the RTM with the original memory layout.



(b) This shows the transformed memory layout for `A` that is optimized for the stencil memory access.

Figure 4.11: This shows the default and the transformed memory layout of the array `A` from figure 4.9.

Algorithm 2 This shows the algorithm that detects the stencil pattern in the memory accesses and transforms the latter accordingly.

```
 1: procedure TRANSFORMSTENCILMEMACCESS($stmtId$, $memAccesses$, $scheduleTree$, $dbc$)
 2:   $G$ = group $memAccesses$ of $stmtId$ by name
 3:   $innerBand$ = get the innermost band of $stmtId$ from $scheduleTree$
 4:   for $M \in G$ do
 5:     if $M$ has equal memory access indices then
 6:       $L = \text{lexmin}(M)$
 7:       $D = \Delta((L \circ M^{-1})^{-1})$
 8:       $D$ = fix the dim of $D$ where $innerBand$ occurs in $M$ to zero
 9:       $mName$ = get array name from $M$
10:       if not $D$ is singleton and $mName$ is not transformed then
11:         $pDims$ = get the nonzero dims of $D$
12:         for $pDim \in pDims$ do
13:           $stencilSize$ = get the stencil size of $pDim$ in $D$
14:           if $\exists stencilSize$ then
15:             transform all $memAccesses$ of the array $mName$ according to $pDim$,
                ↪ $dbc$ and $stencilSize$
16:           end if
17:         end for
18:       end if
19:     end if
20:   end for
21: end procedure
```

dividually for optimization possibilities (compare line 4). Let us have a look at an accompanying example. For this, consider the code in figure 4.9, which has a single statement. For this statement, we look at the loop iteration in line 4 where the group `A` is processed. The group M for `A` looks like this:

$$\begin{aligned} M := &\{\, \mathrm{S}(i,j) \to \mathrm{A}(i,j+1) \,\} \cup \{\, \mathrm{S}(i,j) \to \mathrm{A}(i+1,j) \,\} \cup \\ \cup &\{\, \mathrm{S}(i,j) \to \mathrm{A}(i,j) \,\} \cup \{\, \mathrm{S}(i,j) \to \mathrm{A}(i-1,j) \,\} \cup \\ \cup &\{\, \mathrm{S}(i,j) \to \mathrm{A}(i,j-1) \,\} \end{aligned}$$

For a stencil to be selected for optimization, the entire group has to have equal memory access indices (see line 5). Otherwise it is not clear whether the layout transformation is actually beneficial, so the algorithm proceeds only in this case. For the array `A` in our example, the group has equal memory access indices, as all accesses point to `A[i][j]` with only a constant offset.

In the the next step, let $L := \text{lexmin}(M)$ and calculate the delta set $D := \Delta((L \circ M^{-1})^{-1})$ (lines 6 and 7). The delta set describes the different memory accesses executed for the same statement. For our example, we get:

$$L := \{\, \mathrm{S}(i,j) \to \mathrm{A}(i-1,j) \,\},$$

$$D := \{\, \mathrm{A}(1,1) \,\} \cup \{\, \mathrm{A}(2,0) \,\} \cup \{\, \mathrm{A}(1,0) \,\} \cup \{\, \mathrm{A}(0,0) \,\} \cup \{\, \mathrm{A}(1,-1) \,\}.$$

Next, fix all dimensions of D that are related to the innermost band/loop surrounding the statement to zero (see line 8). These are the accesses where the memory pattern does not provide any benefits, as these are memory accesses that refer to values that are located in the same DBC. In our example, the innermost loop is the j loop, and as we can see from M, this means we have to fix the second dimension of each element of D to zero. D becomes:

$$\{\, \mathrm{A}(2,0) \,\} \cup \{\, \mathrm{A}(1,0) \,\} \cup \{\, \mathrm{A}(0,0) \,\}.$$

If D contains more than one element after that (cf. line 10), it can benefit from the transformation. As we can see, this is the case in our example. Next, find the non-zero dimensions of D (cf. line 11). In our example, only the first dimension of D is not equal to zero.

Next, check if the distances between the ascendingly sorted memory accesses of the nonzero dimension are constant (see line 13). If that is the case, we have found a stencil and get the stencil from the number of elements in D. When looking at the different values in our example, we get $[0, 1, 2]$, which has the constant distance between subsequent entries of 1 and thus is a stencil, of size three in this case.

Thus, we can apply the aforementioned transformation (see line 15). Assuming that the RTM has a DBC size of four, we get the following layout mapping for our example:

$$\begin{aligned}
&T : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z},\\
&T(v,w) = (x,y), \text{ where}\\
&x = 6 \cdot (v \bmod 3) + (6 \cdot 3 \cdot (\left\lfloor \frac{v}{3 \cdot 4} \right\rfloor)) + w,\\
&y = \left\lfloor \frac{v}{3} \right\rfloor \bmod 4
\end{aligned}$$

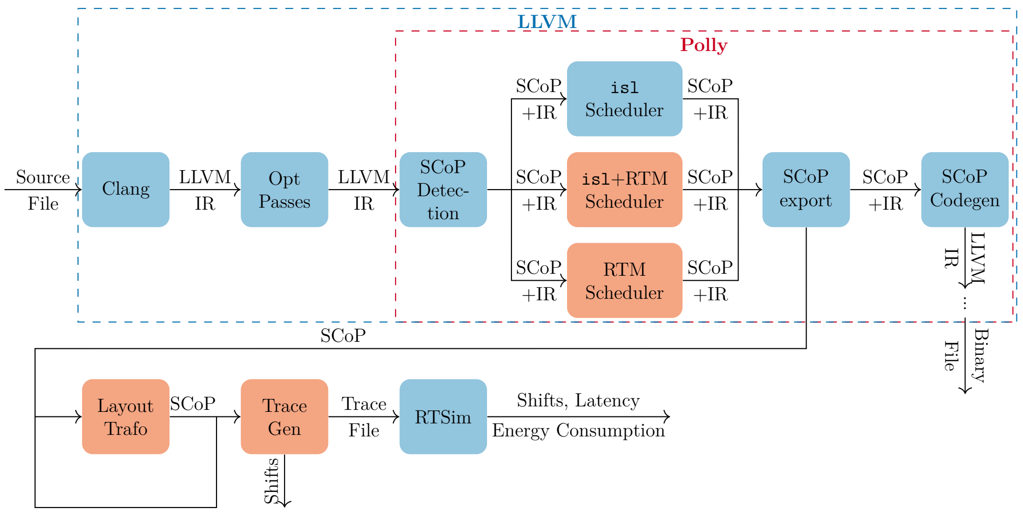


Figure 4.12: This shows the overall workflow from a source file with a SCoP to the results presented in the analysis. The parts that are contributed by this thesis are highlighted by a different background.

4.5 Integration into the LLVM Framework

This section describes the implementation of the preceeding algorithms into an existing polyhedral optimization workflow. There are different implementations of the polyhedral model available, most noticeably Graphite [43] for GCC, and Polly [24] for the LLVM framework [32]. For this work, Polly was chosen as the LLVM framework is easily customizable, and Polly internally uses the `isl` library [56] that makes the suggested algorithms easy to implement.

A high-level overview of the overall workflow is shown in figure 4.12. It shows that Polly itself is implemented as a part of the LLVM optimizer toolchain, starting with the SCoP detection and some normalization passes which are not depicted. After these passes, the original implementation of Polly allows to call the `isl` scheduler combined with some post-processing like tiling if desired. For this work, this pass is modified to switch between three modes. The first option is to call the `isl` scheduler, but disables any further optimizations like tiling. The second is to call algorithm 1 that implements the alternation transformation. The last option combines the `isl` scheduler with the alternation transformation. This is the only modification to the existing LLVM workflow provided by this thesis. After the optimization, the rest of Polly, including the SCoP export and the code generation, continues, leading to a binary after other LLVM optimization passes. The memory layout transformation targeting stencils from section 4.4 is implemented using the exported SCoP from Polly as an optional post-processing step. After that, a trace generation that uses the polyhedral model to generate the array memory accesses in the correct order is used to generate a trace file. This trace file can be used to simulate the memory accesses on RTSim [28], a cycle-accurate racetrack memory simulator. This is necessary as RTMs are still in development and not yet available. This is also the reason why the layout transformation is implemented as a post-processing step and not in Polly itself as the important details on how to specify the exact layout on the RTM are not yet known in the compiler.

5 Evaluation

This chapter presents the evaluation of the optimization techniques for RTM shifts that were introduced in chapter 4.

The evaluations and qualitative comparison of different configurations should answer the following research questions:

RQ1. *Do existing cost-model based polyhedral scheduling algorithms provide any benefit in reducing the shifts?* We have seen in section 4.3 that the cost-model based algorithms have some limitations, but they could be useful for loop fusion or interchange that may reduce RTM shifts.

RQ2. *Does the pattern-based schedule transformation always reduce the shifts?* This question points into two directions: First, whether the optimization provides benefits as expected. Second, if it is actually monotonous in the sense that it does not increase the shifts rather than reduce it.

RQ3. *Does the pattern-based layout transformation always reduce the shifts?* The same as for the previous question applies to this one as well. Do the transformations work as expected? Are there cases where the transformation increases the RTM shifts?

RQ4. *Which combinations of the aformentioned optimizations work best?* All of the three optimizations can be applied independently. Hence, it is a question whether they complement each other when applied together or not.

RQ5. *Does the shift reduction result in reduced latency and/or energy consumption?* The reduction in shifts is motivated in previous research by reduction in the memory latency and the energy consumption. This needs to be checked for the aforementioned optimizations as well.

In the following, whenever *average* is spelled, the arithmetic mean and the correspoding standard deviation are meant. For all diagrams, smaller values are better.

5.1 Benchmarks and Setup

Number of DBCs	1024 × 1024
Domains per DBC	64
Write energy [pJ]	576.2
Read energy [pJ]	447.3
Shift energy [pJ]	420.5
Read latency [ns]	12.82
Write latency [ns]	17.57
Shift latency [ns]	11.14

Table 5.1: The RTM parameters used for the evaluation. The RTM uses 32 racetracks per DBC, which makes a total of 256 Megabytes. The latency and energy number are extracted from the circuit-level memory simulator Destiny [38]. The per-access and per-shift latency and energy numbers also include the latency/energy of the peripheral circuitry.

To answer the research questions above, it is necessary to evaluate the novel optimizations with standard benchmarks. A benchmark suite that is commonly used with polyhedral optimizations is Polybench [65]. In the version 4.2.1 used in this thesis, it provides 30 kernels. The kernel `deriche` was excluded from the evaluation in this thesis as it hit a limitation in the trace generation (see figure 4.12) that made it unable to evaluate it properly. In addition to the 29 kernels from Polybench, 3 representative kernels from the COSMO atmospheric model [1] are included in the evaluation. COSMO spells Consortium for Small-Scale Modelling and is a numerical atmospheric model that can be used for weather forecasts and modelling of the climate on a large scale. The three representative kernels are named `diffusion`, `fastwave`, and `advection`. As the layout transformation from section 4.4 targets stencils, it is important to know which of the kernels are classified as stencils. In total, there are 9, 6 from Polybench (`adi, fdtd-2d, heat-3d, jacobi-1d, jacobi-2d`, and `seidel-2d`, according to [65]) and all three COSMO kernels (`diffusion, fastwave`, and `advection`). The other kernels of Polybench consist of statistical calculations, linear algebra operations

(BLAS and more complex kernels), linear algebra solvers, and some uncategorized kernels.

The experiments were run using the pipeline presented in section 4.5. For the latency and energy results, the values in table 5.1 were used. For the racetrack memory, 64 domains were used per track. The RTM was simulated as a scratchpad memory where it is assumed that the arrays are already in the memory, and the ports of each DBC point to the first entry in its DBC.

5.2 Cost-model Based Scheduling Techniques

The `isl` scheduler [60] combines both the Feautrier scheduler [18, 19] and the Pluto scheduler [9, 10]. Hence, it can be used to evaluate both cost-model based scheduling algorithms for RTM shifts. As the cost-model of these algorithms targets other goals, these are not expected to reduce the shifts, but they might still be beneficial due to loop reordering or loop fusion. The `isl` scheduler provides 10 different boolean parameters [55, p. 194ff.] that can influence the output of the scheduler. Hence, each combination of those can potentially output a different schedule. However, not all of these schedules need to be distinct. In fact, the number of different schedules is two to three magnitudes smaller and ranges from 2 to 26, with a median of 5.0 schedules. All of these distinct schedules were evaluated. The results can be found in figure 5.1. It shows the number of shifts for the best- and the worst-performing schedule. The baseline is the number of shifts in the original kernel. In 23 out of the 32 kernels, none of the computed schedules reduces the number of shifts by more than 5 %, and in the remaining 9 that are the 9 bottom ones in figure 5.1, 4 of them (`advection, syr2k, gemver`, and `threemm`) also have schedules where the shifts are increased by the worst configuration. This already hints that, only in very limited cases, the goals of the `isl` scheduler, namely optimizing for parallelism and temporal locality, incidentally reduce RTM shifts. By examining the schedules for each kernel of the 9 kernels, it is found that in all cases except `syrk`, `syr2k`, and `threemm`, the reduction in shifts is achieved through loop fusion. This fuses accesses to the same location occuring in different loop nests together, hence avoids shifts. All those schedules are computed by the modified Pluto algorithm, except for the best version of the `cholesky` kernel, where the Feautrier scheduler found the best schedule. The other three, `syrk`, `syr2k`, and `threemm`, do not benefit from loop fusion, but from loop interchange, which also groups together accesses to the same memory location. For `syrk`, `syr2k`, the optimal schedule is found by the Feautrier scheduler, while for `threemm`, Pluto interchanged the loops.

The observations are confirmed when aggregating all the relative shifts across all benches for each of the 1024 possible configurations. Of these 1024 possible configuration values, there are 16 configurations with the exact same arithmetic mean of 0.99 ± 0.29 relative shifts compared to the identity. Thus,

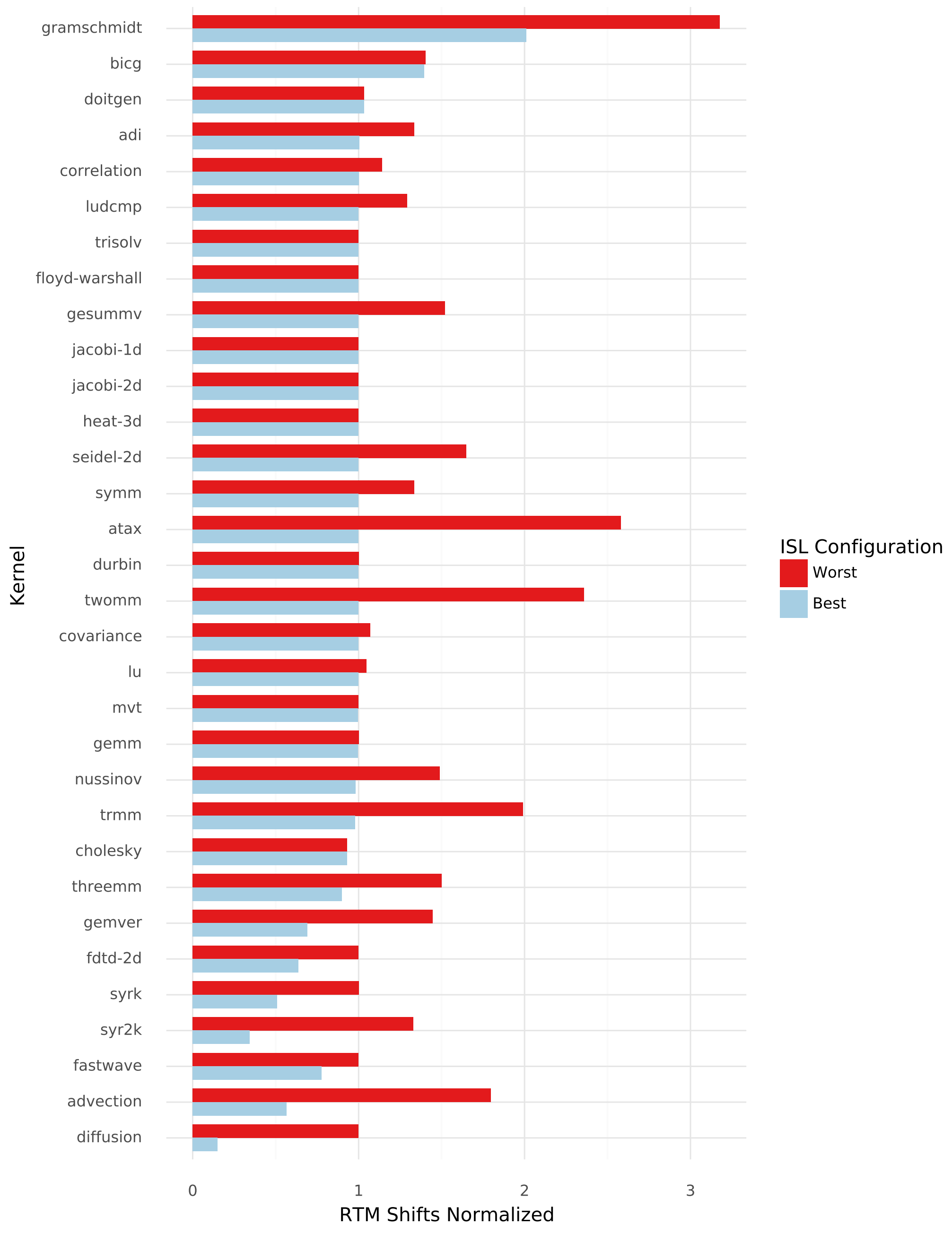


Figure 5.1: This shows the best and the worst schedule in terms of shifts that was calculated by the `isl` scheduler each. The shifts are normalized to the shifts in each of the original kernels, respectively.

there is only a tiny improvement achieved overall. The worst configuration has, on average, 1.29 ± 0.58 relative shifts compared to the identity. The best configurations all contain the option to use the Pluto scheduler, while the worst all use the Feautrier scheduler.

For the 16 configurations leading to the best schedule on average, it is interesting to analyze the common options among these configurations. One option is to use the Pluto scheduler. The second common option in all these 16 configurations is the one that reverts a modification to the Pluto scheduler that exists in the `isl` implementation [60, p. 45ff.] by disabling incremental scheduling of the strongly connected components of the dependency graph. This is particularly interesting because the modified Pluto used in `isl` aims at increasing the number of coincident loops. However, for the shifts this modification seems to have a slight disadvantage. The option that splits the two different groups of configurations into two equal-sized groups of 8 is the one that can disable loop coalescing through bounding the loop coefficients. Apparently, this does not have an influence on the shifts right now, but it might affect the novel pattern-based optimizer, as that aims to optimize single loops.

There is one last observation when considering the kernels `bicg` and `gramschmidt`. For both, every single configuration of the `isl` scheduler makes the shifts worse by at least 39.3 % and 100 % respectively. When looking at the schedule of the best configuration in this case, it becomes clear why this is the reason: For both, loops from the original code are split into several loops, which is the opposite of loop fusion. This further strengthens the observation from that loop fusion is a very useful operation for keeping the RTM shifts low.

Overall, to answer *RQ1*, the analysis of the results show that the impact of existing cost-model based schedulers on RTM shifts is largely arbitrary, and provide neither a clear advantage nor a clear disadvantage. However, as the relatively large standard deviation of 0.29, which is more than one third of the mean, shows, it is highly unpredictable whether the schedule will improve or worsen the number of shifts. That said, detailed analysis of the best cases shows a trend in the result as well. The majority of large improvements is achieved through loop fusion using the Pluto configuration of the `isl` scheduler. Hence, for code where one knows that loop fusion is possible, the `isl` scheduler running the Pluto algorithm might be an option to consider for reducing RTM shifts.

5.3 Pattern-based Schedule Transformation

The results of the pattern-based schedule optimization are presented in Figure 5.2. For the two configurations where the name contains *islb* (for *isl best*), it was chosen to use the overall best configuration of `isl`, that is to

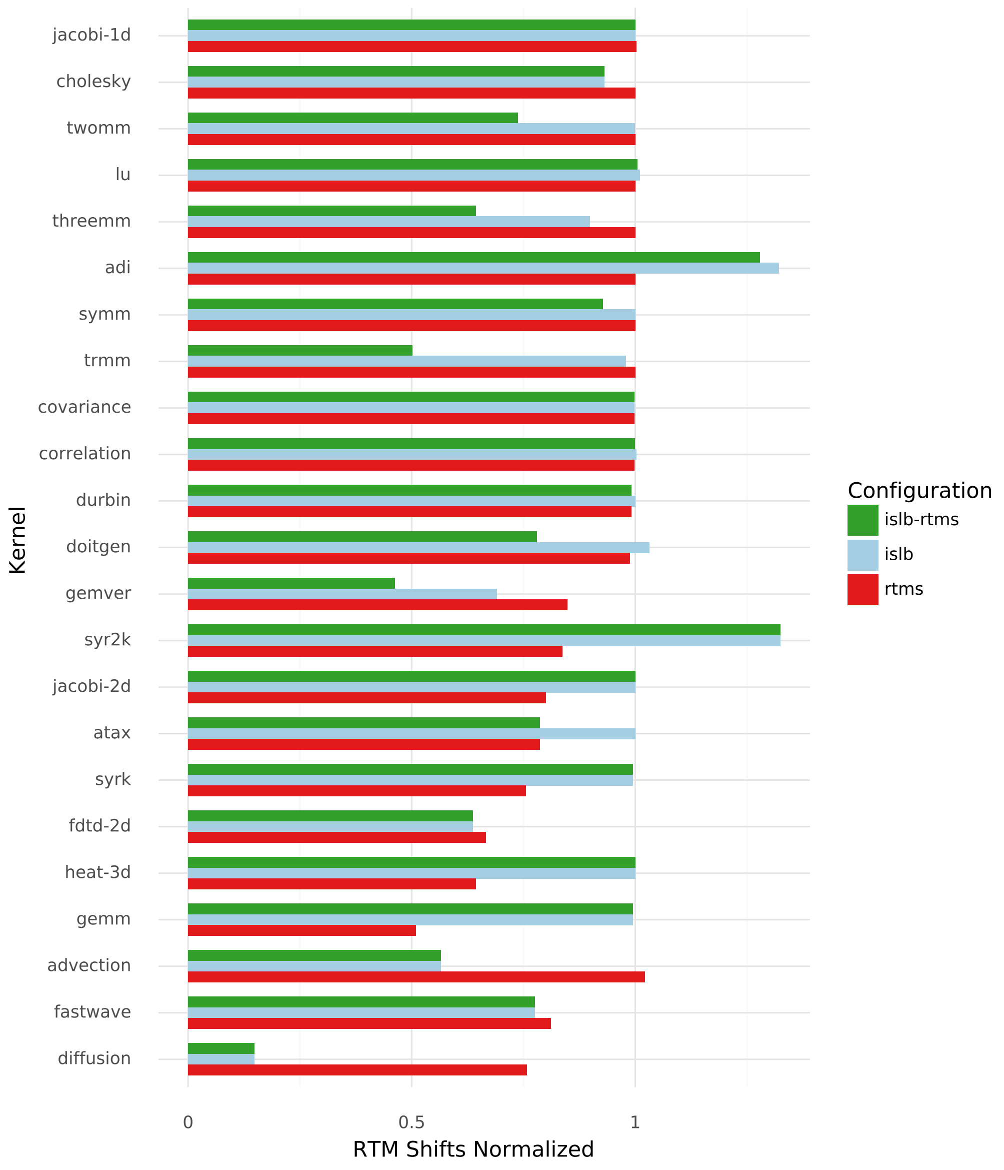


Figure 5.2: This shows changes in shifts using only the pattern-based schedule optimizer (*rtms*), the best configuration of the `isl` scheduler which runs the Pluto algorithm (*islb*), and the combination of the two (*islb-rtms*). The nine cases where the pattern-based schedule optimizer did not perform any transformation (which are `floyd-warshall`, `gesummv`, `nussinov`, `seidel-2d`, `trisolv`, `mvt`, `ludcmp`, `bicg`, `gramschmidt`) are not depicted. The shifts are normalized to the shifts in each of the original kernels, respectively.

use Pluto with both incremental scheduling and loop coalescing disabled, the latter due to the fact that loop coalescing would be a disadvantage to the schedule optimization approach, as it reduces the number of loops

But let us first look at the results of the pattern optimizer in a stand-alone mode. There are 6 kernels where the opimizer alternates at least one loop, but achieves a change in the shifts of under 2 %, 10 kernels where the reduction is more than 10 %, 15 kernels where it detects no optimization possibility, including the 9 kernels that are not shown in figure 5.2, and one where it increases the shifts by 2.2 %. We see the best reduction in shifts for the `gemm` kernel by 49.1 %, which is not surprising as the alternation transformation was first suggested to improve shifts for exactly this type of kernels.

The 10 kernels with considerable shifts reduction are from the following Polybench groups [65]: BLAS routines (`gemm, syrk, syr2k, gemver`), linear algebra kernels (`atax`) and stencils (`heat-3d, fdtd-2d, diffusion, jacobi-2d, fastwave`). These show that the idea of the optimization is effective as these groups are the ones with repeated, yet very simple memory accesses. But one would expect to see an improvement of the shifts in kernels like `twomm` and `threemm` as well, as these are only a combination of multiple matrix-matrix-multiplications. However, here no alternation transformation is applied. An analysis of the code of the two shows that this is due to the loop structure where too many loops are fused and the loops that could be alternated reside outside the first filter node in the schedule, hence they are not selected as candidates for alternation as this would also affect other statements. This shows that a purely pattern-based schedule optimization is limited to cases where the code has a specific structure, and even small changes in this structure forbid the optimization. Hence, it would be desirable to start from a schedule that is not as variable as provided by human code.

For this, the `isl` scheduler (see above for the configuration) is executed in front of the pattern-based schedule transformation. On the one hand, it shows promising results. In 5 kernels, namely `trmm, doitgen, symm, twomm, threemm`, the `isl` scheduler enables shift reduction. The pattern optimizer already affected `trmm` and `doitgen`, but with no substantial improvements in shifts. In the other three cases, the pattern optimizer is able to provide an alternation which was not possible without the `isl` scheduler. On the other hand, in 8 of the 10 kernels where the pattern optimizer alone is effective in reducing the shifts, the `isl` scheduler prevents the pattern-based optimization that was possible without it. This is in some kernels offset by the fact that the `isl` scheduler can provide a reduction in the shifts through loop fusion. For the remaining 2, there is one, `atax`, where the `isl` scheduler does not change anything, while for `gemver`, combining both optimizers reduces the shifts more than each of them alone.

This demonstrates again that applying a cost-model based scheduler upfront whose model does not fit the shifts optimization goal is a double-edged sword and can lead to both fewer or more shifts depending on the kernel.

This is further confirmed by the average of the relative shifts, which are 0.919 ± 0.134, and 0.934 ± 0.319 for the the pattern-based optimizer standalone and the combination of the two respectively, as the standard deviation for the former is much smaller than for the latter. This means that whether the `isl` scheduler improves or deteriorates the shifts is much more unpredictable than for the pattern-based schedule optimization.

This result analysis helps in responding to *RQ2*. There is not even a single kernel that is negatively affected by the alternation transformation, 10 kernels benefit significantly from it, including the `gemm` kernel which originated this idea. Hence, the alternation transformation provides a simple yet useful pattern-based schedule optimization that can help to mitigate shifts in racetrack memory.

A partial answer to *RQ4* is already possible: Combining the two different schedule optimization approaches can help, but also hinder the optimization. It is mostly beneficial for the kernels that contain matrix multiplications, namely `twomm` and `threemm`, where one expected the alternation transformation to be beneficial, but it actually was not because the structure of the original schedule prevented the optimization.

5.4 Layout Transformation

The results of the memory layout transformation are shown in figure 5.3. It only presents the kernels that were affected by the layout transformation. For all other kernels and schedules, no layout transformation was applied, hence the number of shifts did not change compared to the previous sections. First of all, it is noticeable that the layout transformation is detected for five of six Polybench kernels (`seidel-2d, jacobi-2d, heat-3d, fdtd-2d, adi`) that are categorized as stencils [65]. The only one that is missing is the `jacobi-1d`, which is to be expected, as this operates on one-dimensional arrays for which the layout transformation does not provide any benefits. All three COSMO kernels (`diffusion, fastwave, advection`) are also included, and finally, there is a stencil-like access detected in `nussinov`.

The first important observation is that the layout transformation standalone (named *lt* in figure 5.3) is effective, as it reduces the shifts in all nine cases. When taking all 32 kernels into account, the average relative shifts are 0.874 ± 0.243 for *lt* which shows that the effect of the layout transformation, at least in this kernel selection, is even better than the schedule transformations. The complete coverage of all the stencil kernels together with the average reduction in shifts by 12.6 % allows to answer *RQ3* positively.

It is interesting to see how the layout transformation interacts with the

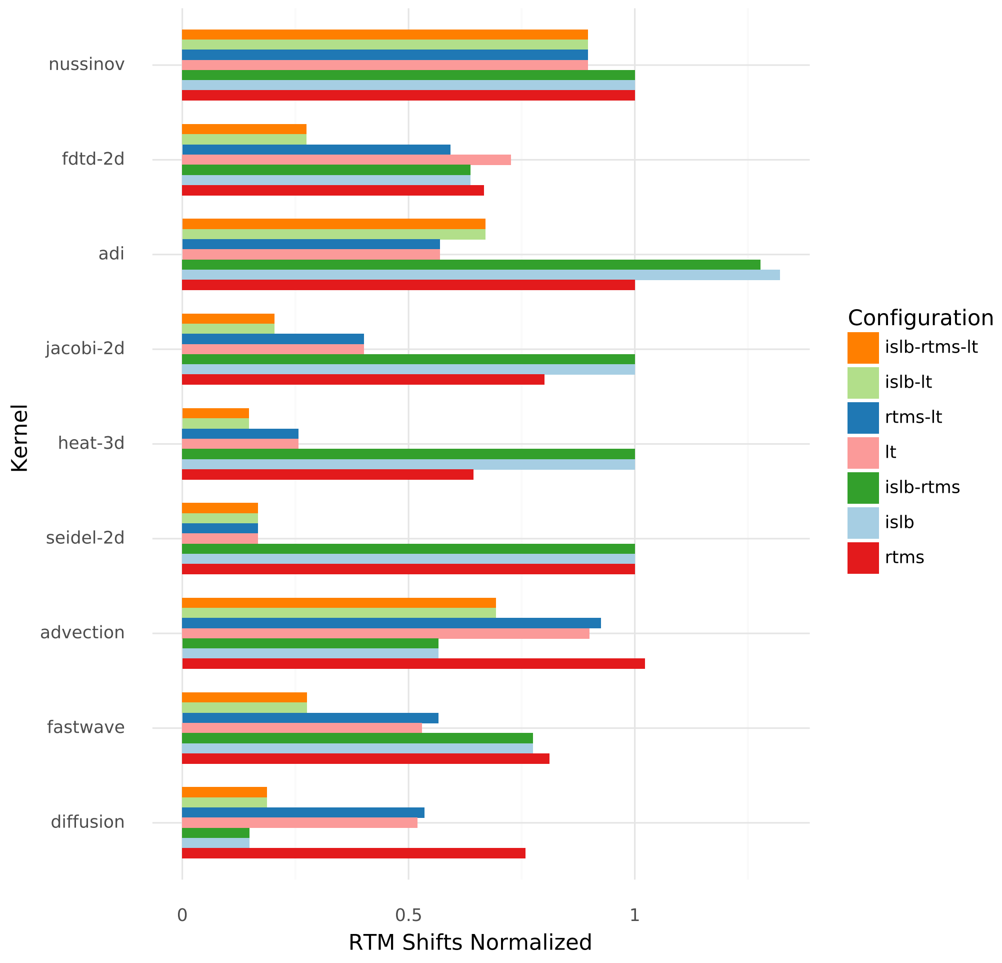


Figure 5.3: This depicts the results for all possible combinations of the three optimizations for the nine kernels where the layout transformation had an effect, i.e., it detected a stencil-like memory access. The shifts are normalized to the shifts in each of the original kernels, respectively. The layout transformation was applied in those configurations that include the name *lt*.

other optimizations. First, one can observe that when the optimization is combined with the `isl` scheduler (named *islb-lt* in figure 5.3), the shifts get reduce even further in 6 cases. Only for `seidel-2d` and `nussinov`, nothing changes. For the `adi` kernel, the shifts increase. This is not surprising as the `adi` kernel is the only one which is also badly affected by the `isl` scheduler without the layout transformation as well.

Second, when combining the layout transformation with the alternation transformation (named *rtms-lt* in figure 5.3), there are five kernels (`seidel-2d, nussinov, jacobi-2d, heat-3, adi`) where nothing changes compared to *lt*, three `advection, fastwave, diffusion`, where the shifts get slightly worse, and only for `fdtd-2d` there is an improvement. The last one can be explained by the fact that somehow the alternation transformation allows the stencil detection to optimize a third array.

When all three optimizations are used (*islb-rtms-lt*), the results are the same as for the combination of the layout transformation with the `isl` scheduler. This is expected from the combination of the `isl` scheduler with the alternation transformation (*islb-rtms*) as in there, the `isl` scheduler prevented all alternations in all kernels affected by the layout transformation except `adi`. This can be seen in figure 5.3 by the fact that *islb* and *islb-rtms* are equal for all kernels except `adi`.

5.5 Summary of the Shift Analysis

The results of all configurations across all benchmarks are shown in figure 5.4 and summarized in table 5.2. The table indicates that, by the arithmetic mean of all kernels, the combination of all three optimizations leads to the best results; this answers *RQ4*. This is backed by the fact that there are 17 kernels in total that have an improvement in shifts by at least 0.05 %. This is mostly driven by the combination of the two optimizations (*rtms-lt*) proposed in this thesis, which improves 14 kernels and has the second-best arithmetic mean among all kernels. Furthermore, in this case, for no kernel, the shifts increase significantly by the optimizations. When we look only at the shifts among the kernels that were improved, the combination of the `isl` scheduler with the layout transformation performs best (*islb-lt*), however, this improvement is achieved in only 12 kernels. Overall, the results show the following:

- The layout transformation seems to have the most significant effect of all three techniques, however, it is also the one that can be applied to the least of the kernel.

- The two proposed optimizations in this work either reduce the number of RTM shifts or leave them unchanged.

Configuration	∅ All	#Aff.	#Ben.	∅ Ben.	#Mal.
rtms	0.919 ± 0.134	17	10	0.742 ± 0.106	**0**
islb	0.991 ± 0.289	22	7	0.664 ± 0.263	4
islb-rtms	0.934 ± 0.319	24	12	0.658 ± 0.220	4
lt	0.874 ± 0.243	9	9	0.552 ± 0.257	**0**
rtms-lt	0.832 ± 0.246	20	14	0.617 ± 0.236	**0**
islb-lt	0.868 ± 0.394	26	12	$\mathbf{0.503 \pm 0.320}$	3
islb-rtms-lt	$\mathbf{0.812 \pm 0.400}$	**28**	**17**	0.546 ± 0.287	3

Table 5.2: This table shows the summarized results. The numbers of affected kernels (#Aff.) are the ones where there occured any changed that altered the number of shifts, even if it is only minor. Beneficial (#Ben.) are those where the shifts are reduced by at least 5 %, whereas maleficial (#Mal.) refer to those where the shifts are increased by at least that number. ∅ All and ∅ Ben. refer to the arithmetic mean and standard deviation of the relative shifts of all kernels or the beneficial ones, respectively.

- The `isl` scheduler, while sometimes highly effective through loop fusion, behaves unpredictable, as expected.
- Loop fusion is an operation that strongly reduces the number of shifts.
- The combination of all optimizations produces the best results.

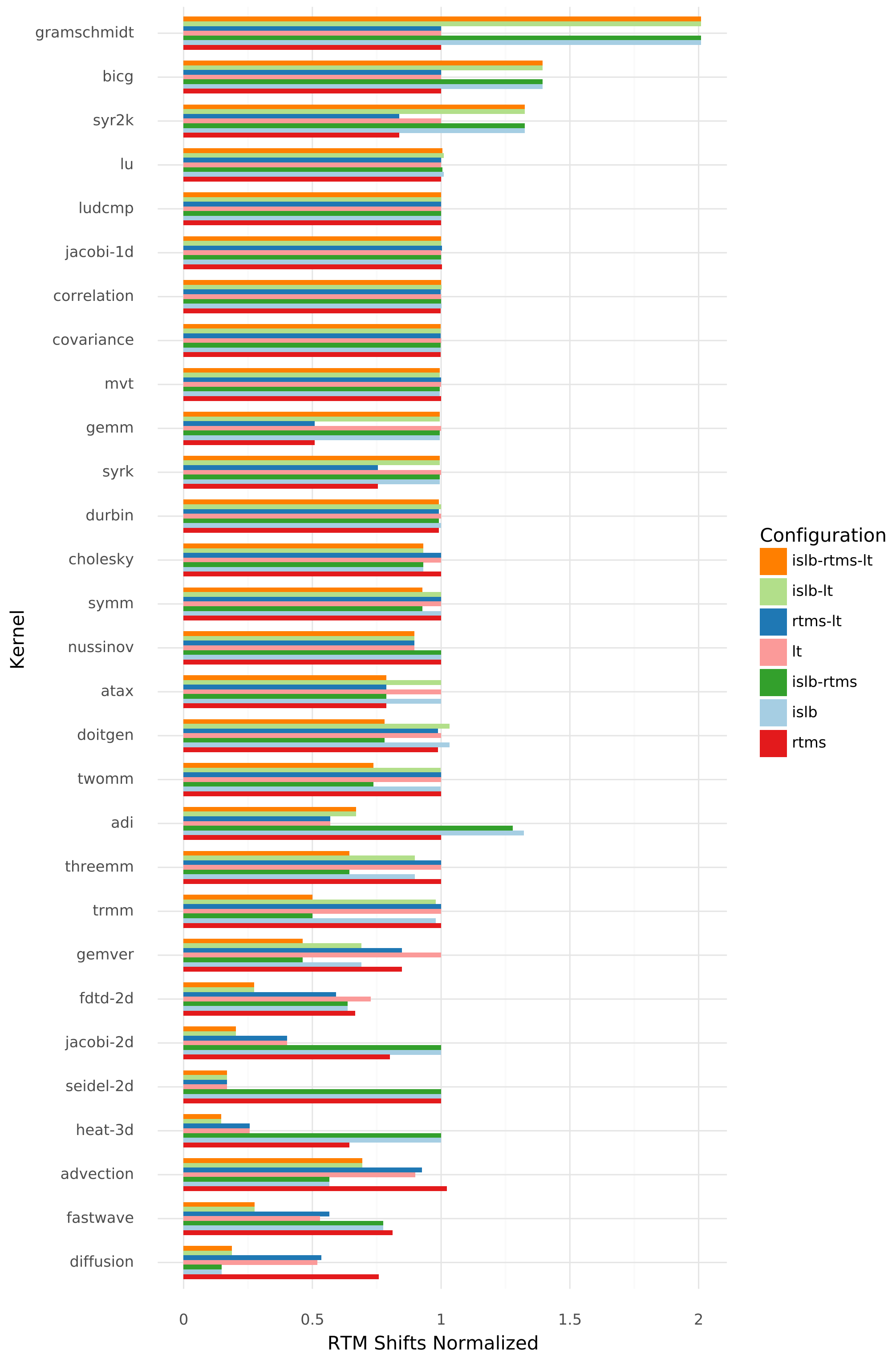


Figure 5.4: This shows the RTM shifts normalized to the identity schedule of all kernels, excluding `floyd-warshall`, `gesummv`, and `trisolv`, as those were not modified in any configuration, and all configurations.

Config	∅ Shifts	∅ Latency	∅ Energy
rtms	0.919 ± 0.134	0.958 ± 0.070	0.956 ± 0.073
islb	0.991 ± 0.289	0.980 ± 0.119	0.979 ± 0.126
islb-rtms	0.934 ± 0.319	0.959 ± 0.126	0.957 ± 0.134
lt	0.874 ± 0.243	0.930 ± 0.136	0.926 ± 0.142
rtms-lt	0.832 ± 0.246	0.911 ± 0.135	0.906 ± 0.141
islb-lt	0.868 ± 0.394	0.911 ± 0.189	0.907 ± 0.199
islb-rtms-lt	$\mathbf{0.812 \pm 0.400}$	$\mathbf{0.891 \pm 0.187}$	$\mathbf{0.885 \pm 0.197}$

Table 5.3: This table shows the arithmetic mean and standard deviation of each configuration for the relative shifts, latency and energy consumption among all kernels.

5.6 Energy and Latency Results

The average latency of each configuration is provided in table 5.3. The summarized results of the latency compared with the RTM shifts for each kernel can be found in figure 5.6. Both indicate the same trend: The reduction in shifts corresponds to a reduction in latency, and an increase in shifts also increases the latency. However, it is also shown that their relationship is not directly proportional. The change in the RTM shifts tends to be more extreme than the change in the latency. However, the difference in the reduction differs greatly between kernels.

For example, when looking at the configuration *rtms* for the kernels `gemm` and `heat-3d`, for the former, the shifts are reduced by 49.1 %, while for the latter, the shifts are reduce for 35.7 %. On the other side, the reduction in latency by 21.8 % and 22.0 % is slightly better for the `heat-3d` kernel in this case. Examining the schedule of both suggests that the number of-per access shifts in `heat-3d` is higher than those in the `gemm` kernel in the unmodified schedule.

Another interesting kernel is again `gramschmidt`. There, the shifts are increased by the `isl` scheduler (configuration *islb*) by around 100 %, while the access latency is only increased by 17 %. This is due to the fact that in the `gramschmidt` kernel, there are a lot more accesses to the same memory locations than there are shifts. Thus, there are a lot of accesses that do not require shifting at all in both cases. Hence, the latency does not suffer as much from the increase in shifts as it does in other kernels. Overall, the latency results confirm that the combination of the `isl` scheduler with both the pattern-based schedule and layout transformation yields the best result.

The summarized results of the energy consumption compared with the RTM shifts for each kernel can be found in figure 5.6, and the arithmetic means of the energy results are provided in table 5.3. Overall, the reduction in energy consumption behaves similarly to the reduction in latency.

To summarize, one can answer *RQ5* positively as both the energy and latency are reduced when the RTM shifts are reduced, but the effect is not directly proportional due to the reasons explaind above.

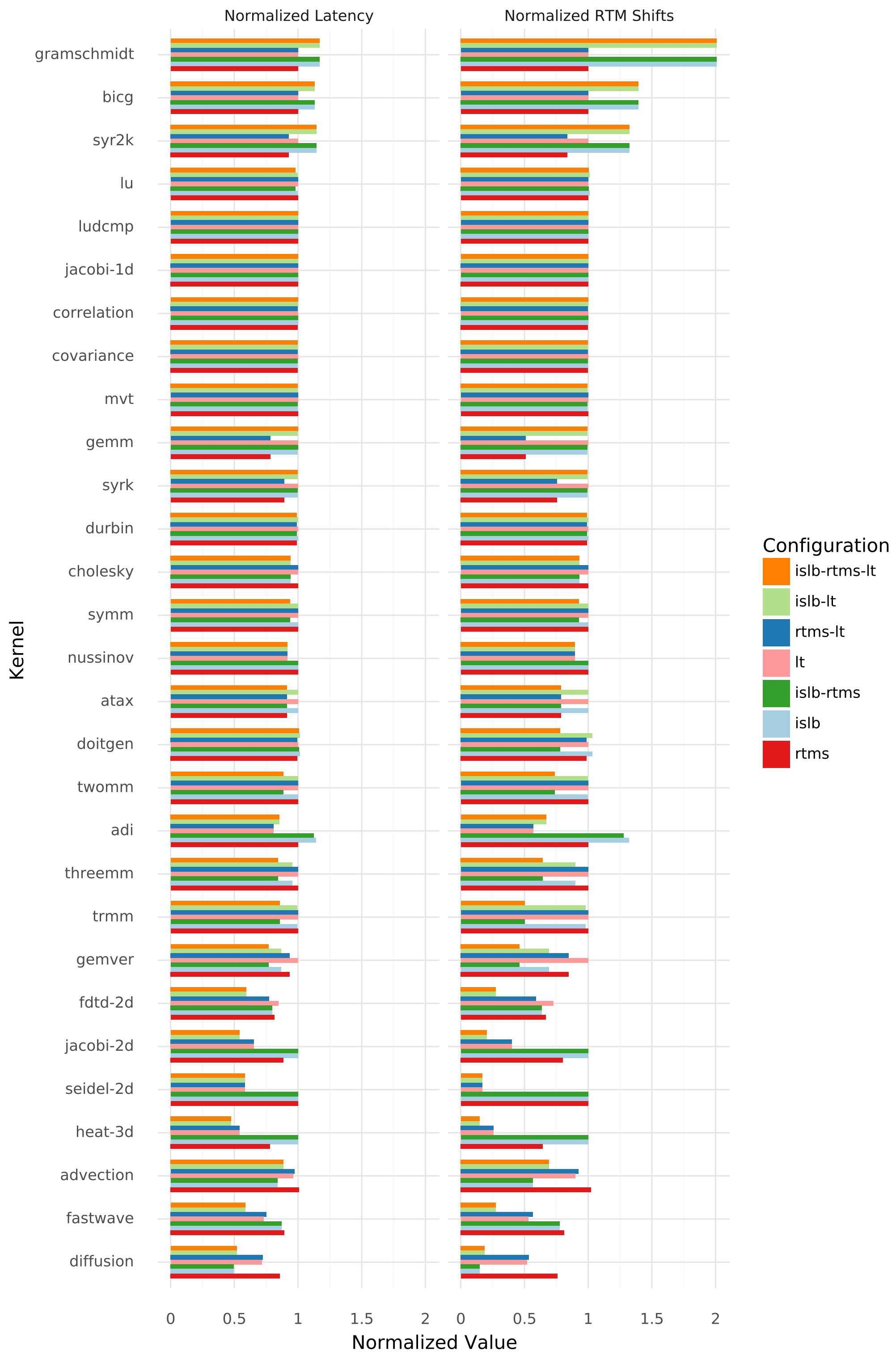


Figure 5.5: This shows the latency results next to the RTM shifts normalized to the identity schedule of all kernels excluding `floyd-warshall, gesummv`, and `trisolv`, as those shifts did not change in any configuration.

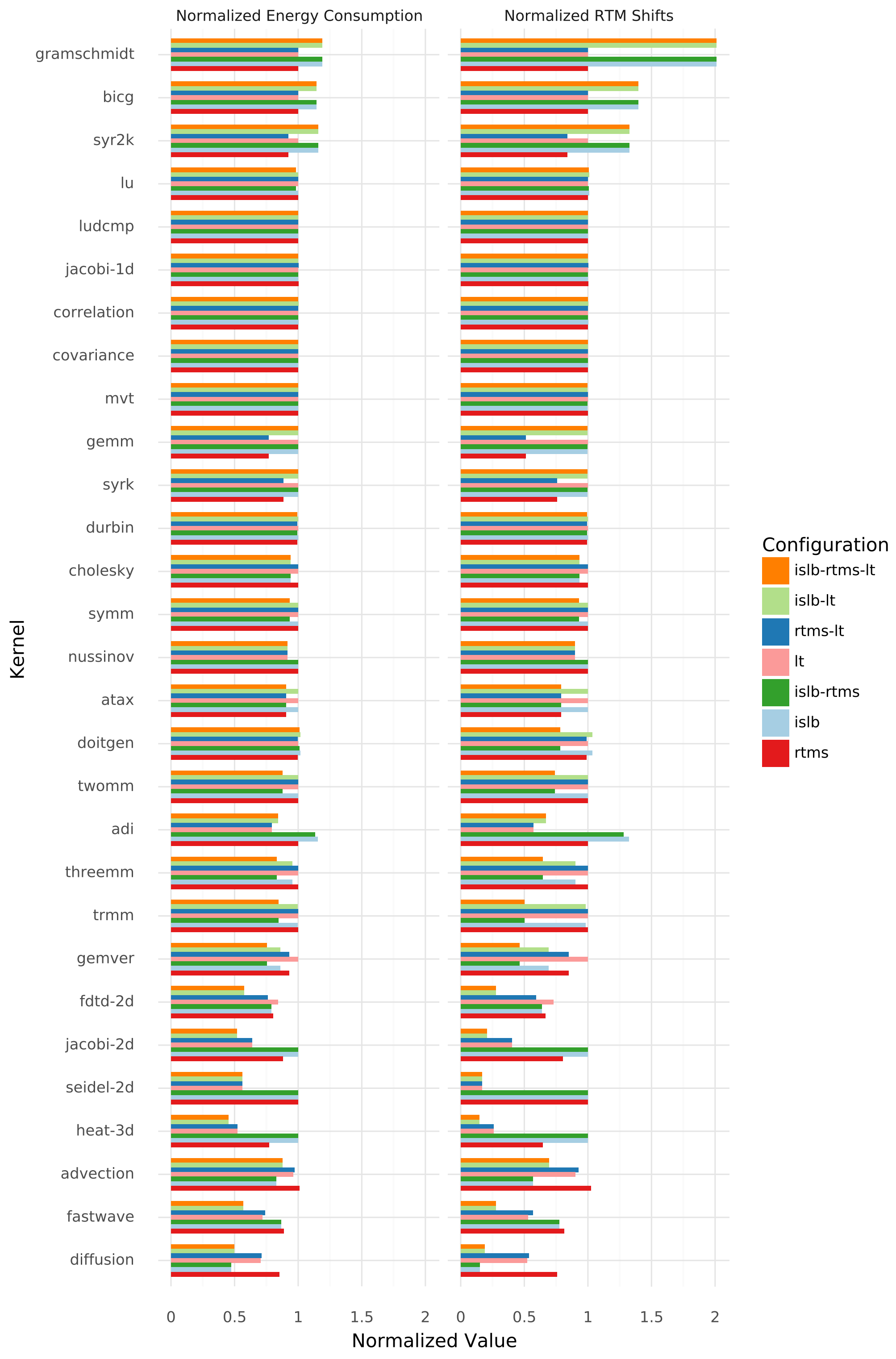


Figure 5.6: This shows the energy consumption results next to the RTM shifts normalized to the identity schedule of all kernels excluding `floyd-warshall`, `gesummv`, and `trisolv`, as those shifts did not change in any configuration.

6 Conclusion & Future Work

This section summarizes the thesis and discusses some future work and proposals that could lead to further improvements.

The main contribution of this thesis is the first automatic framework that optimizes the static control parts in programs for minimal-offset locality in racetrack memories. The framework is built on top of the polyhedral optimizer Polly and is integrated into the mainstream LLVM framework. It offers support for both schedule and memory layout transformations to reduce the number of RTM shifts and can combine those with the existing `isl` scheduler. The evaluations on the polybench and COSMO kernels showed that the framework can achieve a significant reduction in RTM shifts, providing an important step forwards to integrating RTMs into mainstream computational systems. More specifically, both the layout and schedule transformations in this thesis guarantee that the RTM shifts are either reduced or the program is left unmodified. However, the best results were achieved combining both transformations with the `isl` scheduler calling Pluto, as this covered the most kernels and achieved a reduction in shifts of 18.8 % on average. Nonetheless, this framework is only the first step in the right direction. In the following, some possibilities for future work and suggestions to improve the performance of the framework are presented.

First, both pattern detection algorithms in their current form are not optimized for execution time. Some non-representative measurements show that the current implementation increases the compilation time. However, this was not investigated thoroughly as this was not the focus of this thesis. One idea would be to switch the implementation to the loop tactics framework [11] which is meant for pattern-based optimization like the ones in this thesis. Unfortunately, this was too recently published to adapt it.

Second, the implementation can also potentially benefit from using the MLIR compiler infrastructure [33] that specifically aims at building a reusable and extensible compiler infrastructure with a focus on heterogeneous hardware. The latter is exactly what is needed for a use-case specific optimization like the reduction of shifts in racetrack memories. Furthermore, MLIR provides an affine dialect[1] that is supposed to make the polyhedral transformations easy to implement.

Third, as explained earlier, the iterative scheduling approach was discared, largely due to two reasons: First of all, it often results in complex schedules that are not suitable for postprocessing, which is currently necessary to achieve the alternation transformation, as explained in section 4.3. However, one could possibly embed the alternation transformation by splitting statements that might benefit from the alternation transformation into two different statements with distinct domains. The second reason not to use the iterative scheduling was that it is currently impractical, as one needs to actually compute the metrics, in this case the shifts, for all schedules that are explored. This is currently very time consuming, as of now, there exists no model that can estimate the shifts. However, very recent work for a polyhedral cache miss estimation model [25] has shown that the polyhedral model is capable of estimations for memory models, and there exist first analytical models for the overall RTM shifts [29, 30] that could provide a starting point for a generalized estimation of shifts. It might also be possible to apply machine learning algorithms to model the shift costs of programs, as this was also done for execution time [21].

Finally, this work specifically focused on software-based optimizations to mitigate the RTM shifts in arrays, as there existed no prior work to do this automatically. However, it did not focus on other optimizations goals that are traditional part of polyhedral scheduling, the most important one being parallelism. But there are transformations that are possible for cost-model based schedulers that can mitigate the shifts (cf. chapter 4), and chapter 5 shows that, by chance, these can coincide with the goals of the `isl` scheduler, thus achieving both minial-offset locality and parallelism. Hence, to support both goals explicitely and not by chance, one could try to improve the current `isl` scheduler by changing the cost-model to include the shifts as well. The work on consecutivity [59] provides a similar modification to the `isl` scheduler, although as explained in section 3.1.1, this modification is not the one required for RTMs. But it certainly seems possible, though not obvious, to improve the `isl` scheduler for RTMs while still optimizing for parallelism, tiling and locality. This would require a more complex design of racetrack memories, as the memory architecture needs to support parallel accesses to memory regions. This, however, is thoroughly covered in this recent review on RTMs [7].

[1] `https://mlir.llvm.org/docs/Dialects/Affine/`, visited on 2020-04-19.

Bibliography

[1] Michael Baldauf, Axel Seifert, Jochen Förstner, Detlev Majewski, Matthias Raschendorfer, and Thorsten Reinhardt. “Operational Convective-Scale Numerical Weather Prediction with the COSMO Model: Description and Sensitivities”. In: *Monthly Weather Review* 139.12 (2011), pp. 3887–3905.

[2] Ana Balevic and Bart Kienhuis. “A data parallel view on polyhedral process networks”. In: *SCOPES*. ACM, 2011, pp. 38–47.

[3] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. “Tiling stencil computations to maximize parallelism”. In: *SC*. IEEE/ACM, 2012, p. 40.

[4] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. “Automatic C-to-CUDA Code Generation for Affine Programs”. In: *CC*. Vol. 6011. Lecture Notes in Computer Science. Springer, 2010, pp. 244–263.

[5] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. “The Polyhedral Model Is More Widely Applicable Than You Think”. In: *CC*. Vol. 6011. Lecture Notes in Computer Science. Springer, 2010, pp. 283–303.

[6] Sabpreet Bhatti, Rachid Sbiaa, Atsufumi Hirohata, Hideo Ohno, Shunsuke Fukami, and S.N. Piramanayagam. “Spintronics based random access memory: a review”. In: *Materials Today* 20.9 (2017), pp. 530–548.

[7] Robin Bläsing, Asif Ali Khan, Panagiotis Ch. Filippou, Chirag Garg, Fazal Hameed, Jeronimo Castrillon, and Stuart S. P. Parkin. “Magnetic Racetrack Memory: From Physics to the Cusp of Applications within a Decade”. In: *Proceedings of the IEEE* (Mar. 2020), pp. 1–19.

[8] U. Bondhugula, V. Bandishti, A. Cohen, G. Potron, and N. Vasilache. “Tiling and optimizing time-iterated computations over periodic domains”. In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 2014, pp. 39–50.

[9] Uday Bondhugula, Muthu Manikandan Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model". In: *CC*. Vol. 4959. Lecture Notes in Computer Science. Springer, 2008, pp. 132–146.

[10] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A practical automatic polyhedral parallelizer and locality optimizer". In: *PLDI*. ACM, 2008, pp. 101–113.

[11] Lorenzo Chelini, Oleksandr Zinenko, Tobias Grosser, and Henk Corporaal. "Declarative Loop Tactics for Domain-specific Optimization". In: *TACO* 16.4 (2020), 55:1–55:25.

[12] Xianzhang Chen, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Penglin Dai, and Weiwen Jiang. "Optimizing data placement for reducing shift operations on domain wall memories". In: *DAC*. ACM, 2015, 139:1–139:6.

[13] Xianzhang Chen, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Chun Jason Xue, Weiwen Jiang, and Yuangang Wang. "Efficient Data Placement for Improving Data Access Performance on Domain-Wall Memory". In: *IEEE Trans. VLSI Syst.* 24.10 (2016), pp. 3094–3104.

[14] Daichi Chiba, Gen Yamada, Tomohiro Koyama, Kohei Ueda, Hironobu Tanigawa, Shunsuke Fukami, Tetsuhiro Suzuki, Norikazu Ohshima, Nobuyuki Ishiwata, Yoshinobu Nakatani, and Teruo Ono. "Control of Multiple Magnetic Domain Walls by Current in a Co/Ni Nano-Wire". In: *Applied Physics Express* 3.7 (July 2010), p. 073004.

[15] Jason Cong and Jie Wang. "PolySA: polyhedral-based systolic array auto-compilation". In: *ICCAD*. ACM, 2018, p. 117.

[16] Andi Drebes, Lorenzo Chelini, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, Tobias Grosser, Kanishkan Vadivel, and Nicolas Vasilache. *TC-CIM: Empowering Tensor Comprehensions for Computing-In-Memory*. IMPACT 2020 - 10th International Workshop on Polyhedral Compilation Techniques. Jan. 2020.

[17] Paul Feautrier. "Dataflow analysis of array and scalar references". In: *International Journal of Parallel Programming* 20.1 (1991), pp. 23–53.

[18] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. I. One-dimensional time". In: *International Journal of Parallel Programming* 21.5 (1992), pp. 313–347.

[19] Paul Feautrier. "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time". In: *International Journal of Parallel Programming* 21.6 (1992), pp. 389–420.

[20] Stefan Ganser, Armin Größlinger, Norbert Siegmund, Sven Apel, and Christian Lengauer. “Iterative Schedule Optimization for Parallelization in the Polyhedron Model”. In: *TACO* 14.3 (2017), 23:1–23:26.

[21] Stefan Ganser, Armin Größlinger, Norbert Siegmund, Sven Apel, and Christian Lengauer. “Speeding up Iterative Polyhedral Schedule Optimization with Surrogate Performance Models”. In: *TACO* 15.4 (2019), 56:1–56:27.

[22] Roman Gareev, Tobias Grosser, and Michael Kruse. “High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach”. In: *TACO* 15.3 (2018), 34:1–34:27.

[23] Tobias Grosser. “Enabling Polyhedral Optimizations in LLVM”. Diploma thesis. 2011.

[24] Tobias Grosser, Armin Größlinger, and Christian Lengauer. “Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation”. In: *Parallel Processing Letters* 22.4 (2012).

[25] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. “A fast analytical model of fully associative caches”. In: *PLDI*. ACM, 2019, pp. 816–829.

[26] Masamitsu Hayashi, Luc Thomas, Rai Moriya, Charles Rettner, and Stuart S. P. Parkin. “Current-Controlled Magnetic Domain-Wall Nanowire Shift Register”. In: *Science* 320.5873 (2008), pp. 209–211.

[27] Asif Ali Khan, Andres Goens, Fazal Hameed, and Jeronimo Castrillon. “Generalized Data Placement Strategies for Racetrack Memories”. In: *CoRR* abs/1912.03507 (2019).

[28] Asif Ali Khan, Fazal Hameed, Robin Blaesing, Stuart Parkin, and Jeronimo Castrillon. “RTSim: A Cycle-Accurate Simulator for Racetrack Memories”. In: *Computer Architecture Letters* 18.1 (2019), pp. 43–46.

[29] Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart S. P. Parkin, and Jeronimo Castrillon. “ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0”. In: *ACM Trans. Archit. Code Optim.* 16.4 (Dec. 2019).

[30] Asif Ali Khan, Norman A. Rink, Fazal Hameed, and Jeronimo Castrillon. “Optimizing tensor contractions for embedded devices with racetrack memory scratch-pads”. In: *LCTES*. ACM, 2019, pp. 5–18.

[31] Emre Kultursay, Mahmut T. Kandemir, Anand Sivasubramaniam, and Onur Mutlu. “Evaluating STT-RAM as an energy-efficient main memory alternative”. In: *ISPASS*. IEEE Computer Society, 2013, pp. 256–267.

[32] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *CGO*. IEEE Computer Society, 2004, pp. 75–88.

[33] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: A Compiler Infrastructure for the End of Moore's Law". In: *CoRR* abs/2002.11054 (2020).

[34] Stan Y. Liao, Srinivas Devadas, Kurt Keutzer, Steven W. K. Tjiang, and Albert R. Wang. "Storage Assignment to Decrease Code Size". In: *ACM Trans. Program. Lang. Syst.* 18.3 (1996), pp. 235–253.

[35] Haiyu Mao, Chao Zhang, Guangyu Sun, and Jiwu Shu. "Exploring data placement in racetrack memory based scratchpad memory". In: *NVMSA*. IEEE, 2015, pp. 1–5.

[36] Vadim Maslov. "Lazy Array Data-Flow Dependence Analysis". In: *POPL*. ACM Press, 1994, pp. 311–325.

[37] Ioan Mihai Miron, Thomas Moore, Helga Szambolics, Liliana Daniela Buda-Prejbeanu, Stéphane Auffret, Bernard Rodmacq, Stefania Pizzini, Jan Vogel, Marlio Bonfim, Alain Schuhl, and Gilles Gaudin. "Fast current-induced domain-wall motion controlled by the Rashba effect". In: *Nature Materials* 10.6 (2011), pp. 419–423.

[38] Sparsh Mittal, Rujia Wang, and Jeffrey Vetter. "DESTINY: A Comprehensive Tool with 3D and Multi-Level Cell Memory Modeling Capability". In: *Journal of Low Power Electronics and Applications* 7.3 (Sept. 2017), p. 23.

[39] Joonas Multanen, Pekka Jääskeläinen, Asif Ali Khan, Fazal Hameed, and Jeronimo Castrillon. "SHRIMP: Efficient Instruction Delivery with Domain Wall Memory". In: *ISLPED*. IEEE, 2019, pp. 1–6.

[40] Stuart S. P. Parkin, Masamitsu Hayashi, and Luc Thomas. "Magnetic Domain-Wall Racetrack Memory". In: *Science* 320.5873 (2008), pp. 190–194.

[41] Stuart SP Parkin. *Shiftable magnetic shift register and method of using the same*. Dec. 2004.

[42] Stuart Parkin and See-Hun Yang. "Memory on the racetrack". In: *Nature Nanotechnology* 10.3 (2015), pp. 195–198.

[43] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. "GRAPHITE: Polyhedral analyses and optimizations for GCC". In: *Proceedings of the 2006 GCC Developers Summit*. Citeseer. 2006, p. 2006.

[44] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. "Iterative optimization in the polyhedral model: part ii, multidimensional time". In: *PLDI*. ACM, 2008, pp. 90–100.

[45] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. "Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time". In: *CGO*. IEEE Computer Society, 2007, pp. 144–156.

[46] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. "Polyhedral-based data reuse optimization for configurable computing". In: *FPGA*. ACM, 2013, pp. 29–38.

[47] Benoît Pradelle, Benoît Meister, Muthu Manikandan Baskaran, Jonathan Springer, and Richard Lethin. "Polyhedral Optimization of TensorFlow Computation Graphs". In: *ESPT/VPA@SC*. Vol. 11027. Lecture Notes in Computer Science. Springer, 2017, pp. 74–89.

[48] William Pugh. "The Omega test: a fast and practical integer programming algorithm for dependence analysis". In: *SC*. ACM, 1991, pp. 4–13.

[49] William Pugh and David Wonnacott. "An Exact Method for Analysis of Value-based Array Data Dependences". In: *LCPC*. Vol. 768. Lecture Notes in Computer Science. Springer, 1993, pp. 546–566.

[50] Kwang-Su Ryu, Luc Thomas, See-Hun Yang, and Stuart Parkin. "Chiral spin torque at magnetic domain walls". In: *Nature Nanotechnology* 8.7 (2013), pp. 527–533.

[51] Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[52] Jun Shirako, Akihiro Hayashi, and Vivek Sarkar. "Optimized two-level parallelization for GPU accelerators using the polyhedral model". In: *CC*. ACM, 2017, pp. 22–33.

[53] Rangharajan Venkatesan, Vivek Joy Kozhikkottu, Charles Augustine, Arijit Raychowdhury, Kaushik Roy, and Anand Raghunathan. "TapeCache: a high density, energy efficient cache based on domain wall memory". In: *ISLPED*. ACM, 2012, pp. 185–190.

[54] Rangharajan Venkatesan, Shankar Ganesh Ramasubramanian, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. "STAG: Spintronic-Tape Architecture for GPGPU cache hierarchies". In: *ISCA*. IEEE Computer Society, 2014, pp. 253–264.

[55] Sven Verdoolaege. *Integer Set Library: Manual*. English. Version isl-0.21. INRIA. Mar. 26, 2019. 254 pp. URL: `http://isl.gforge.inria.fr/isl-0.21.tar.gz` (visited on 03/21/2020).

[56] Sven Verdoolaege. "*isl*: An Integer Set Library for the Polyhedral Model". In: *ICMS*. Vol. 6327. Lecture Notes in Computer Science. Springer, 2010, pp. 299–302.

[57] Sven Verdoolaege. *Presburger formulas and polyhedral compilation*. English. Version v0.02. Polly Labs and KU Leuven. Jan. 15, 2016. 172 pp. URL: `https://lirias.kuleuven.be/retrieve/361209` (visited on 04/28/2020).

[58] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. "Schedule Trees". In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Ed. by Sanjay Rajopadhye and Sven Verdoolaege. Vienna, Austria, Jan. 2014.

[59] Sven Verdoolaege and Alexandre Isoard. "Extending Pluto-Style Polyhedral Scheduling with Consecutivity". In: 2018.

[60] Sven Verdoolaege and Gerda Janssens. *Scheduling for PPCG*. Tech. rep. CW 706. Department of Computer Science, KU Leuven, June 2017.

[61] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. "On Demand Parametric Array Dataflow Analysis". In: *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*. Ed. by Armin Größlinger and Louis-Noël Pouchet. Berlin, Germany, Jan. 2013, pp. 23–36.

[62] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming-Jinn Tsai. "Metal-Oxide RRAM". In: *Proceedings of the IEEE* 100.6 (2012), pp. 1951–1970.

[63] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. "Phase Change Memory". In: *Proceedings of the IEEE* 98.12 (2010), pp. 2201–2227.

[64] See-Hun Yang, Kwang-Su Ryu, and Stuart Parkin. "Domain-wall velocities of up to 750 m s-1 driven by exchange-coupling torque in synthetic antiferromagnets". In: *Nature Nanotechnology* 10.3 (2015), pp. 221–226.

[65] Tomofumi Yuki and Louis-Noël Pouchet. *PolyBench 4.2.1 (pre-release)*. English. Version 4.2.1-beta. Ohio State University. May 20, 2016. 14 pp. URL: `https://sourceforge.net/projects/polybench/files/polybench-c-4.2.1-beta.tar.gz` (visited on 04/04/2020).