A Compiler for Symbolic Code Generation for Tightly Coupled Processor Arrays

Ein Compiler zur symbolischen Code-Erzeugung für eng gekoppelte Rechenfelder

Der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur
Erlangung des Doktorgrades Dr.-Ing.

vorgelegt von

Michael Witterauf
Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg
Tag der mündlichen Prüfung: 18. Oktober 2021

Vorsitzender des Promotionsorgans: Prof. Dr.-Ing. Knut Graichen

Gutachter: Prof. Dr.-Ing. Jürgen Teich
Prof. Christian Lengauer, Ph. D.
A tightly coupled processor array is a type of systolic array, a term with its etymology in cardiology: The systole is the stage of the heart’s rhythm when the heart contracts and pumps blood. After the systole follows the diastole, the stage when the heart relaxes. In the following partial electrocardiogram, the diastole is drawn black, the systole red:

This rhythmic pumping of blood through the body is metaphorically similar to the “rhythmic pumping” of data through a systolic array, where in each time step, each processing element processes its inputs and “pumps” the results to its neighbors.

«Je n’ai fait celle-ci plus longue que parce que je n’ai pas eu le loisir de la faire plus courte.»

(Loosely: “I would have written a shorter letter, but I didn’t have the time.”)

— Blaise Pascale, 1657, in Lettres Provinciales
This dissertation presents symbolic loop compilation, the first full-fledged approach to symbolically map loop nests onto TCPAs (tightly coupled processor arrays), a class of loop accelerators that consist of a grid of PEs (processing elements). It is:

- **Full-fledged** because it covers all steps of compilation, including space-time mapping, code generation, and generation of configuration data for all involved hardware components. A full-fledged compiler is paramount because manual mapping for accelerators, such as TCPAs, is difficult, tedious, and, most of all, error-prone.

- **Symbolic** because symbolic loop compilation assumes the loop bounds and number of allocated PEs to be unknown during compile time, thus allowing them to be chosen at run time. This flexibility benefits resource-aware applications where the number of PEs is known only at run time.

Symbolic loop compilation is a hybrid static/dynamic approach with two phases: At compile time, all involved NP-hard problems (such as resource-constrained modulo scheduling) are solved symbolically, resulting in a so-called *symbolic configuration*, which is a space-efficient intermediate representation parameterized in the loop bounds and number of PEs. This phase is called *symbolic mapping*. Because it takes place at compile time, there is ample time to solve the involved NP-hard problems. At run time, for each requested accelerated execution of a loop program with given loop bounds and number of allocated PEs, concrete PE programs and configuration data are generated from the symbolic configuration according to these parameter values. This phase is called *instantiation*.

In the context of these two phases, this dissertation presents the following contributions:

- **Symbolic modulo scheduling** is a technique for solving resource-constrained modulo scheduling for multi-dimensional loop nests when the loop bounds and number of available PEs are unknown. We show that a latency-minimal solution can be found if the number of PEs is known beforehand and a near latency-minimal solution if it is not.

- **Polyhedral syntax trees** are a space-efficient, parameterized representation of a set of PE program variants from which the necessary concrete PE programs are generated at run time.

- **Instantiation** includes methods to generate concrete programs and configuration data from a symbolic configuration in a manner whose time complexity is not proportional to the loop bounds or number of allocated PEs.
- **Run-time requirement enforcement for loops** is a technique that utilizes the flexibility granted by symbolic loop compilation to enforce requirements on non-functional properties by dynamically adapting the mapping before execution. An example is to allocate a number of PEs that satisfies a given latency bound.

In summary, the methods presented in this dissertation enable, for the first time, the full-fledged symbolic compilation of loop nests onto TCPAs. Without these methods, a given loop nest would need to be fully recompiled each time the loop bounds or number of available PEs change, which would render run-time mapping impractical and even conventional compilation overly time- and space-consuming.
Acknowledgments

This dissertation was only possible because of Prof. Dr.-Ing. Jürgen Teich, to whom I want to express my gratitude in particular. Not only did he provide me with the opportunity to research such an interesting topic in-depth, but he was also always eager to make time for discussions about scientific problems, research directions, and technical challenges. Furthermore, his feedback on my papers was invaluable. I would also like to thank Prof. Christian Lengauer, Ph. D. for co-examining my dissertation, especially considering the admittedly very formal content.

Special thanks go to PD Dr.-Ing. Frank Hannig, who always had an open ear for problems, scientific or otherwise, and whose guidance helped me on more than one occasion. Finally, I want to thank all my TCPA colleagues, but in particular Marcel Brand and Dominik Walter, whose detailed feedback only improved my work and who I could always count on to have exciting and fruitful discussions.
# Contents

1 Introduction ....................................................... 1
  1.1 Tightly coupled processor arrays .......................... 1
  1.2 Run-time requirement enforcement ......................... 2
  1.3 The need for symbolic loop compilation ................. 3
  1.4 Contributions .............................................. 3
  1.5 Publications ............................................... 5
  1.6 Related work .............................................. 6
  1.7 Roadmap through the thesis ................................ 8

2 Tightly Coupled Processor Arrays ............................... 11
  2.1 Architecture ............................................... 12
    2.1.1 Processing elements ............................... 12
    2.1.2 Interconnect network ............................ 19
    2.1.3 Global controllers ............................... 21
    2.1.4 I/O buffers ........................................ 22
    2.1.5 TCPA synthesis flow ................................ 25
  2.2 TCPA configuration data .................................. 25
    2.2.1 Processing elements ............................... 26
    2.2.2 Interconnect network ............................ 27
    2.2.3 Global controller ............................... 28
    2.2.4 I/O buffers—Address generators ................. 28
    2.2.5 Configuration flow ................................ 29
  2.3 Integration .................................................. 29
    2.3.1 Multi-core systems with a shared bus ............ 29
    2.3.2 Tiled MPSoCs with a network-on-chip: Invasive Computing 30
    2.3.3 Driver .............................................. 32
  2.4 Related work ............................................... 32

3 Fundamentals of Loop Parallelization and Scheduling in the Polyhedral Model 33
  3.1 Uniform dependence algorithms .......................... 34
  3.2 Presburger sets ............................................ 40
  3.3 Dependence graphs ....................................... 42
  3.4 Space-time mapping ....................................... 46
    3.4.1 Choice of allocation and scheduling techniques .. 49
    3.4.2 Symbolic tiling ..................................... 50
    3.4.3 Processing element allocation .................... 56
    3.4.4 Modulo scheduling .................................. 59
"The free lunch is over," C++-expert Herb Sutter proclaimed in his famous 2005 article of the same name [Sut05], foreseeing a revolution of parallel programming. The metaphor “free lunch” refers to Dennard scaling [DGY*74], a projection that the possible operating frequency of a chip inversely scales with the ever-decreasing transistor size. Software developers would reap these benefits in the form of free speedups by simply upgrading to new processors. While transistor sizes keep shrinking to this day, Dennard scaling came to a rough stop around the mid-2000s because the leakage of individual transistors became significant. Not only did this make further increasing operating frequencies not viable due to overheating, but it also became impossible to utilize the entirety of a chip at the same time. This phenomenon has been termed the “utilization wall” [VSG*10].

1.1 Tightly coupled processor arrays

Much time has passed since 2005, but as of 2021, the utilization wall still looms menacingly above chip development; less and less of a chip can be active at the same time. The parts of a chip that remain inactive at a time are known as dark silicon [EBA*11]. A commonly accepted solution to counter dark silicon is heterogenization: By building systems-on-chip with low-energy, application-specific accelerators that are only active when required, simultaneously active chip parts are kept to a minimum. An important question arises: Which kinds of hardware accelerators are worth investigating?

From an application-centric viewpoint, for-loops are an excellent candidate. Not only do they represent the majority of computation in most scientific, signal processing, economic, and many more applications, but they also naturally exhibit a large degree of parallelism across multiple levels. Dedicated loop accelerators exploit this vast parallelism to provide loop acceleration that is both fast and energy-efficient. In this dissertation, we focus on one class of such loop accelerators: TCPAs (tightly coupled processor arrays) [DKH*09; HLB*14], which we introduce in-depth in Chapter 2. Explained briefly, TCPAs provide a grid of weakly programmable PEs (weakly means without a stack, cache, interrupts, …) [KHK*06], paired with peripheral components specifically designed to accelerate loops. In contrast to most other loop accelerators, TCPAs provide comprehensive loop acceleration by accelerating not only the computation of a loop, but also its control and communication. This enables more rigorous off-loading of loops
and ensures that all these parts of loop execution are performed in a timing-predictable manner. Predictability, as the next section explains, is becoming increasingly important in today’s system design.

### 1.2 Run-time requirement enforcement

The most common use for an energy-efficient loop accelerator such as a TCPA is within an embedded system. With each year, an increasing number of traditionally non-computerized “things” are turned into embedded systems. Many of these embedded systems—prime examples include industrial machinery and automobiles—require non-functional execution properties, such as execution time, throughput, or reliability, to be bounded. For example, an airbag controller must react in a timely manner. A robot vision pipeline must sustain its frame rate. A satellite must operate reliably in space. In addition, other non-functional properties, such as energy consumption, are desired to be optimized. For example, a smartphone video application should, as a secondary goal, minimize energy consumption to maximize user satisfaction. Predicting non-functional properties, however, is hindered by uncertainty, of which there are two major sources: complex hardware architectures and unknown input sizes.

Especially modern multi-core processors exacerbate execution uncertainty because their microarchitectural state is too complex. For example, caches accelerate programs on average, but widen the interval between worst-case (no code is cached) and best-case (all code is cached) execution times. The number of possible states of a cache is tremendous, making modeling them accurately almost futile. Other examples include out-of-order execution and deep pipelining.

TCPAs, on the other hand, are designed with timing predictability in mind. Given the loop bounds and number of PEs, some non-functional properties, such as the execution latency, can be evaluated accurately. Still, input sizes—and hence loop bounds—are generally not known until execution. For example, a feature detection algorithm might output a different number of features for different inputs, or the input image size is selected by the user. To satisfy given requirements (bounds) on non-functional properties, a mapping of the loop onto the processor array that satisfies them for the given loop bounds must therefore be determined at run time. This process is called run-time requirement enforcement [TGR’16]. Consider the following example: An application wants to use a TCPA to accelerate a loop nest and specifies a certain deadline. From the deadline and an accurate formula for the execution latency, the number of PEs to allocate for execution that both satisfies the deadline and minimizes energy consumption can be determined. In general: the more PEs, the faster the execution, but the more energy is consumed; the fewer PEs, the slower the execution, but the less energy is consumed. This dynamic mapping lies at the heart of run-time requirement enforcement. However, as it turns out, leaving the loop bounds and number of allocated PEs unknown until run time challenges traditional compilation approaches.
1.3 The need for symbolic loop compilation

To enable the low-energy, predictable, and parallelized acceleration of loops, the components of a TCPA execute in a tightly synchronized manner, which requires the necessary configuration data to be specific to the given combination of loop bounds and number of allocated PEs. Generating this configuration data, including programs, from a loop specification by hand is difficult, time-consuming, and error-prone, ultimately escalating development costs of applications—a loop compiler is crucial. However, the steps for automatically generating the necessary configuration data and programs involve multiple NP-hard problems, such as resource-constrained modulo scheduling. We face a conundrum: The configuration data depends on parameters only known at run time—the loop bounds and number of PEs—but space and time constraints prohibit run-time compilation because embedded systems in particular only have limited memory and computational power.

This thesis is the culmination of research toward a solution known as symbolic loop compilation, where the compilation of a loop is split into two distinct phases, as visualized in one picture in Figure 1.1: symbolic mapping, performed at compile time, and instantiation, performed at run time [WHT19; WWW+]. Symbolic mapping frontloads all intractable NP-hard problems to the compile time—leaving ample time to solve them optimally—and generates data structures called symbolic configurations. A symbolic configuration contains all necessary configuration data and programs in a space-efficient form parameterized on the loop bounds and number of PEs. At run time, instantiation generates a concrete configuration stream that can be directly used to reconfigure the target TCPA in order to accelerate the given loop according to any given values of the parameters. The time complexity of instantiation is not proportional to the number of PEs or the loop bounds and therefore scales well with increasing loop bounds and TCPA size.

Symbolic loop compilation not only enables run-time requirement enforcement, but also dynamic resource management in general. For example, if multiple applications already share a TCPA, a new application cannot know beforehand how many PEs are available. Using symbolic loop compilation, the application can adequately react. As such, symbolic loop compilation is a powerful tool for application developers.

1.4 Contributions

Condensed into one sentence:

Symbolic loop compilation is the first full-fledged, hybrid static/dynamic parallelization approach to compile $n$-dimensional loop nests with unknown bounds onto an unknown number of PEs (processing elements) of a TCPA (tightly coupled processor array).
1 Introduction

Any loop size $N \times M$ TCPA

App 1: $2 \times 5$

App 2: $1 \times 3$

App 3: $? \times ?$

Any available allocation size: $R \times C$

Figure 1.1: Visual summary of symbolic loop compilation. Using symbolic mapping (Chapter 5), an $n$-dimensional loop nest with unknown bounds is compiled into a symbolic configuration at compile time. This entails symbolic modulo scheduling (Chapter 4) because existing modulo scheduling techniques do not apply. A symbolic configuration contains a polyhedral syntax tree, a space-efficient program representation (Chapter 5). From the compiled symbolic configuration, concrete configuration data and PE programs are generated using instantiation (Chapter 6).

Previous approaches for TCPAs (see Section 1.6) either were not full-fledged (for example, missing code generation) or were static, not allowing for unknown loop bounds or an unknown number of PEs. The former case entails manual processing, decreasing developer productivity, while the latter case requires to recompile a given loop each time the loop bounds or number of available PEs changes, making run-time mapping impractical. Approaches for other accelerator architectures, such as CGRAs (coarse-grained reconfigurable arrays), usually utilize only the instruction-level parallelism of the inner-most level of an $n$-dimensional loop nest, eschewing much of the parallelism the loop nest may offer (see Appendix C).

In more detail, the highlights of this thesis’ contributions are:

- **Symbolic modulo scheduling, which enables modulo scheduling despite an unknown number of PEs** (Chapter 4): Having parametric loop bounds and number of PEs poses challenges for latency-minimal scheduling of $n$-dimensional loop nests because conventional techniques for solving resource-constraint modulo scheduling fail to apply. We remedy this by introducing symbolic modulo scheduling, which is able to statically determine symbolic loop schedules that are latency-minimal if the number of PEs is known beforehand and near latency-minimal if the number is not known beforehand.

- **Register allocation and communication routing for symbolic loop schedules** (Section 4.8): Having symbolic loop schedules affects other parts of the
1.5 Publications

For a full list of my publications, refer to the bibliography. This dissertation focuses on the results summarized in the following four papers:

- “Modulo scheduling of symbolically tiled loops for tightly coupled processor arrays” [WTH+16] presents symbolic modulo scheduling, a technique to modulo-schedule \( n \)-dimensional loop programs despite not knowing the loop bounds and number of available PEs.

- “Polyhedral Fragments: An Efficient Representation for Symbolically Generating Code for Processor Arrays” [WHT19] introduces polyhedral syntax trees as a parameterized, space-efficient representation of loop programs, as well as how to generate them from modulo-scheduled reduced dependence graphs.

- “Symbolic loop compilation for tightly coupled processor arrays” [WWW+19] builds upon the previous two papers and presents instantiation (the generation of configuration
data from a symbolic configuration, given values for the loop bounds and allocated number of PEs), as well as register allocation and propagation channel routing despite the number of PEs being unknown at compile time.

- “Run-time Requirement Enforcement for Loop Programs on Processor Arrays” [WT18] formalizes run-time requirement enforcement for loops and presents two approaches for reducing the number of mappings at compile time an enforcer needs to consider at run time.

In these publications, I developed and formalized the method (symbolic modulo scheduling, instantiation, run-time requirement enforcement of loops) respectively data structure (polyhedral syntax trees) for solving the problem at hand after conceptualizing with and gaining inspiration from my co-authors. I further conducted the corresponding experiments, as well as collected and interpreted the results after discussion with my co-authors. Finally, I wrote both the first content-complete draft of each of these papers, and was responsible for the final editing for consistency after my co-authors had provided their feedback.

1.6 Related work

For related work regarding massively parallel architectures similar to TCPAs and the corresponding mapping methods, we refer to Appendix C.

In previous work, Boppu et al. describe techniques for static loop compilation and code generation for TCPAs [BHT13; BHT14; Bop15]. While this thesis builds in part upon that work, it improves upon it significantly in the following aspects:

- Dynamic mapping: The biggest difference of Boppu’s work is that both the loop bounds and number of allocated PE need to be known at compile time. Hence, each different input size requires recompiling and thus re-solving NP-hard problems. Symbolic loop compilation, by contrast, only needs to perform instantiation in case of changed loop bounds or number of allocated PEs.

- Efficient resource usage: Register allocation is not formalized in [BHT13; BHT14] except for one type of registers, the general-purpose registers (see Section 2.1). Still, the general-purpose registers are not reused across disjoint subsets of the iteration space of a loop, possibly decreasing performance due to register pressure. Symbolic loop compilation leverages the advantages of the polyhedral model to reuse registers as much as possible (Section 4.9), not only for general-purpose registers, but for all types of registers of a TCPA. The same holds for interconnect routes and ports (Section 4.9.4), which were also not reused in Boppu’s work, posing a severe restriction because the number of interconnect ports is very limited.
• Complete resource constraints: The control signals required for synchronization of PE programs are not constrained and not reused in Boppu’s work. Symbolic loop compilation takes the actual number of available control signals into account during control signal allocation (Section 6.3) and reuses each as much as possible.

• Architecture: Boppu assumed VLIW PEs, while symbolic loop compilation as described in this thesis assumes a PE architecture called orthogonal instruction processing \[BHT^+17\] (see Section 2.1.1). The main difference is that with orthogonal instruction processing, the individual functional unit programs do not need to be folded into VLIWs, which would potentially result in code size bloat.

Before Boppu, Hannig was the first to describe how to automatically map loops onto TCPAs, then still known as WPPAs (weakly programmable processor arrays, [DKH+09; KHK+06]), within the context of the high-level synthesis tool PARO [Han09]. From an algorithm description in the functional language PAULA, PARO synthesizes a parallelized, application-specific accelerator in the form of a processor array with optimal performance according to an integer linear program (ILP) that describes resource and data dependency constraints, building upon the work of Teich et al. [TTZ96; TTZ97]. To map loops onto WPPAs instead of synthesizing a dedicated accelerator, Hannig proposes to add WPPA-specific constraints to the ILP—in particular, register and channel constraints. He further describes how to generate VLIW instructions from the ILP’s solution, but not how to combine these into PE programs. By contrast, this thesis details the generation of both complete PE programs and other configuration data (see Chapter 6). In addition, as with Boppu’s work, both the loop bounds and number of PEs of the target processor array are assumed static, that is, known at compile time.

Other previous work investigated symbolic hierarchical tiling and symbolic optimal scheduling of multi-dimensional loop nests \[TTH13; TTH14; TWT^+14; TWT^+15; Tan17\]^1. Hierarchical tiling divides the iterations of a loop into a hierarchy of so-called tiles. Each level in the hierarchy is either a sequential or a parallel level, where its parts (tiles or, on the lowest level, iterations) are symbolically scheduled to execute sequentially resp. in parallel. Choosing an appropriate combination of sequential and parallel levels allows trade-offs regarding required I/O-bandwidth and local memory size. However, in this thesis, we build upon one specific instance known as locally sequential, globally parallel (LSGP): Iterations within a tile are executed sequentially, while tiles are executed in parallel by being mapped one-to-one to PEs (see Section 3.4.2). Moreover, Tanase et al. had so far not described any specific way to generate PE programs or configuration data from the obtained symbolic schedules, making the approach not full-fledged. Finally, the method schedules iterations atomically, that is, there is no software pipelining (see Section 3.4.4), which reduces the achievable instruction-level parallelism and thus impacts performance negatively. Symbolic loop compilation, however, applies symbolic

^1Note that some of these publications were also co-authored by the author of this thesis; for details, refer to the bibliography.
modulo scheduling to achieve software pipelining (see Chapter 4).

We know of no other work with the same scope of this thesis—generation of all necessary configuration data, from periphery to programs, in a symbolic fashion. However, there are many related works concerning the individual steps, for which we refer to the related work sections within the individual chapters.

1.7 Roadmap through the thesis

Chapter 2 gives an overview of TCPAs, including their architecture, reconfigurability, and integration into a larger system. Afterward, we introduce the polyhedral model in Chapter 3. The polyhedral model is a formal model of for-loops that allows the use of mathematical tools, such as linear algebra, to greatly facilitate analyses and transformations.

The remainder of the thesis is structured according to Figure 1.2, which depicts the compile flow from the specification of a for-loop in a high-level language, such as C++ or X10, to the execution on a given target TCPA. The compile flow is partitioned into three parts: loop extraction, symbolic mapping, and instantiation. Both loop extraction and symbolic mapping are performed during compile time, while instantiation is performed at run time.

Loop extraction, as the name implies, extracts annotated for-loops from a high-level language to be compiled onto the target TCPA. For example, in C++, an annotated loop might be:

```c
#pragma tcpa
for (auto i = 0U; i < N; ++i) {
  // ...
}
```

Note that the bounds can be specified using parameters, such as N in this example. The extracted loop is first transformed into a UDA (uniform dependence algorithm), a type of recurrence algorithm, (see Section 3.1) and then into a RDG (reduced dependence graph) (see Section 3.3), which is the intermediate representation used throughout symbolic mapping. This transformation consists of the following steps: single-assignment conversion [Fea91], embedding [Han09], and localization [TR91; Tei93]. This thesis only describes its contributions, that is, symbolic mapping and instantiation, in detail; for more information on loop extraction, refer to the cited works.

Symbolic mapping generates a set of symbolic configurations from the RDG generated during loop extraction\(^2\). Note that there is a set of symbolic configurations because there are usually multiple schedule candidates that could be latency-minimal depending on the concrete parameter values. Symbolic mapping consists of symbolic modulo scheduling, described in Chapter 4, and the subsequent generation of symbolic configurations.

\(^2\)An RDG may stem from other sources. For example, a high-level synthesis tool like PARO allows the description of algorithms using a functional language and transformation into an RDG [HRD08].
Figure 1.2: Compile flow for symbolic loop compilation onto TCPAs, including run-time requirement enforcement.
described in Chapter 5. At this point, the functional correctness of symbolic mapping may be verified using a cycle-accurate simulator. The automatic generation of this simulator from an architecture description is described in [WHT17], but details are omitted in this thesis for brevity.

Finally, at run time, a so-called enforcer performs run-time requirement enforcement and selects a mapping according to given requirements, such as latency bounds, and secondary objectives, such as minimizing energy consumption. The mapping includes the number of PEs to allocate, as well as the symbolic configuration to instantiate. (If run-time requirement enforcement is not used, then the symbolic configuration associated with the latency-minimal schedule is selected.) Then, instantiation generates concrete program and configuration data from the selected symbolic configuration according to the then-known loop bounds and number of allocated PEs. Instantiation is described in-depth in Chapter 6, run-time requirement enforcement of loops in Chapter 7.

Chapter 8 concludes this thesis by summarizing its contributions and giving an outlining future work that may further improve symbolic loop compilation.

**Remark.** Appendix A gives an overview of the formal notation used throughout this thesis.

---

3 Instantiation can also be performed during compile time if all parameters happen to be already known then, unburdening the run-time system even more.
Tightly Coupled Processor Arrays

Construction of a compiler necessitates deep knowledge of the target architecture, in the case of this thesis TCPAs (tightly coupled processor arrays) [DKH+09; HLB+14]. TCPAs are a class of processor arrays specially designed for timing-predictable high-performance loop acceleration with tight energy budgets, for example in embedded systems. Juxtaposing these seemingly contradictory goals demands TCPAs to deviate from conventional accelerator designs in salient ways; highlights include:

Comprehensive loop parallelization  TCPAs accelerate all parts of a loop nest to exploit all levels of parallelism it exhibits. In contrast, other accelerators (in particular CGRAs) often only accelerate up to the iteration level and exclude loop- and task-level parallelism. Comprehensive loop parallelization requires novel mapping methods, making it the main motivator for this thesis.

Systolic streaming  Because accessing off-chip main memory and long wires increase energy consumption, TCPAs are instead operating similarly to systolic arrays [KL78]. Systolic refers to the rhythmic, recurring contraction of the heart muscle; in a processor array, systolic describes a type of streaming where data is “pumped” from processing element to processing element using only local connections. Consequently, memory accesses only take place at the borders of the array. To facilitate this type of local data propagation, TCPAs offer a flexible, reconfigurable interconnect [KHK+06].

Zero-overhead looping  Static control flow in a loop involves updating a multi-dimensional counter and checking whether it satisfies loop-dependent conditions. This wastes resources when performed by processing elements themselves. Also, due to the regularity of loops, control flow after parallelization is mostly shared between processing elements. TCPAs avoid this overhead by computing it centrally in a global controller and propagating the generated control signals systolically.

Orthogonal instruction processing  Instruction-level parallelism in processors is often implemented using VLIWs (very long instruction words). However, VLIWs can lead to bloated programs because the individual programs of the functional units, each with possibly different control flow, must be folded together. Instead, TCPAs
use OIP (orthogonal instruction processing) [BHT+17], where each functional unit has its own instruction memory and sequencer.

TCPAs are also highly configurable using synthesis-time parameters, which define the concrete architecture of a TCPA. Detailing the architecture and synthesis flow from an architecture description to synthesizable HDL code is the focus of the following section.

2.1 Architecture

The physical layout of a TCPA is illustrated in Figure 2.1, the component hierarchy in Figure 2.2. At the top level, a TCPA contains a processor array—a grid of interconnected, simple PEs—that is connected to the surrounding periphery composed of global controllers for synchronization and I/O buffers for energy-efficient external data access. Each PE is embedded into an interconnect wrapper connected to its four neighbors (or I/O buffers at the borders) to form a reconfigurable, circuit-switched interconnect network. Finally, an I/O controller is responsible for the timely draining and refilling of the I/O buffers to external data sinks or from external data sources.

The signals propagated between PEs are either data signals or control signals; this distinction is called the semantics of a signal. Data signals have a configurable width of \( w \) bits, while control signals are always 1 bit. This separation is reflected both in the PEs, which contain registers for both, and the interconnect network, which provides separate layers for both.

For further insight, the next sections elaborate on each component’s architecture and contribution to achieving the three goals underlying the design of TCPAs (to reiterate: high performance, timing predictability, and energy efficiency).

2.1.1 Processing elements

The processor array comprises a configurable number of \( R \times C \) simple PEs arranged in a grid with \( R \) rows and \( C \) columns. As depicted in Figure 2.3, each PE \( p := (r, c) \in \{0 \ldots R - 1\} \times \{0 \ldots C - 1\} \) contains only two parts: a set of functional units connected to a pair of register files semantically separated into data and control. This simplicity aids in achieving the three goals: (1) Omitting sources of unnecessary overhead and thus undesired uncertainty, such as caches, interrupt handling, and multi-threading, makes loop processing on PEs timing-predictable, (2) keeping data and instructions local instead of reading from energy-hungry global memories improves energy efficiency, and (3) providing multiple functional units to exploit instruction-level parallelism offers high performance potential.
Figure 2.1: Top-level block diagram of a TCPA’s components. At the center resides the processor array, a grid of PEs embedded into interconnect wrappers, each of which is connected to its four neighbors to form the interconnect network. Surrounding the processor array, there is the periphery consisting of up to four global controllers (GCs) in the corners and of four I/O buffers at the borders. Each global controller is connected to its corresponding corner PE to propagate control signals into the processor array. Each I/O buffer is connected to the PE directly facing it. The I/O controller generates DMA transfers that are processed by an external DMA controller.
Figure 2.2: An $R \times C$ TCPA consists of a processor array and periphery. The processor array contains a grid of $R \times C$ PEs, embedded one-to-one into interconnect wrappers. Each interconnect wrapper is connected to its four neighbors and together the wrappers form the TCPA's interconnect network. A PE consists of two parts: a set of functional units and a pair of register files, one for data and one for control values. The periphery consists of one to four global controllers, responsible for synchronization, and of four I/O buffers, responsible for external data access. Each I/O buffer is divided into modules, each of which contains multiple memory banks and address generators. Managing the data transfers from and to a TCPA is the responsibility of the single-instance I/O controller.
2.1 Architecture

Data and control register file

A PE contains two register files, one for data and one for control signals. Both provide the following types of registers, where the mnemonics of data registers are suffixed with \( d \) and control registers with \( c \):

- **General-purpose registers** \( r^d_n \) and \( r^c_n \) correspond to conventional data registers found in most processors and store a single word value. These registers are used either for temporary storage, such as intermediate values between operations within a loop iteration, or for storing preloaded values, such as the coefficients of an FIR filter or convolution.

- **Input registers** \( i^d_n \) and \( i^c_n \) provide access to the PE’s input ports (see Section 2.1.2). For each input port, there is one corresponding input register. They differ in their realization: The data input registers \( i^d_n \) are FIFOs, whereas the control input registers \( i^c_n \) are shift registers. This difference is due to the tight synchronization required for propagating control signals—they must arrive in the correct cycle no matter if or how often they are read. The maximum depth of input registers is configurable at synthesis time.

- **Output registers** \( o^d_n \) and \( o^c_n \) are registers whose output is connected to the output ports of the PE; consequently, there is one output register per output port (see Section 2.1.2). Output registers are also useful in cases where a value is output and read back later.

- **Feedback (shift) registers** \( f^d_n \) and \( f^c_n \) are cyclic shift registers with a reconfigurable depth that rotate when they are accessed. For example, assume a feedback shift register with a depth of 3 and the values \((1, 2, 3)\). Reading the register would yield 3 and rotate the values into the order \((3, 1, 2)\). Similarly, writing the value 4 to the register would yield \((4, 1, 2)\). Feedback shift registers are used for recurring data, such as loop-carried dependences, or data to be propagated between image lines. The maximum depth of feedback shift registers is configurable at synthesis time.

- **Virtual registers** \( v^d_n \) and \( v^c_n \) allow simultaneous write access to multiple registers. How many registers a virtual register can write to at once is a synthesis-time parameter; which registers are written to is reconfigurable at run time. For example, using a virtual register, an intermediate result can be propagated within the same iteration \((r^d_0)\), to a later iteration \((f^d_0)\), and to another processing element \((o^d_0)\) in the same cycle, without requiring multiple instructions.

While the number of registers of each register type is individually configurable at synthesis time, all registers of the same semantics share the same configurable width.
Figure 2.3: An OIP PE consists of a set of functional units and local memory in the form of two semantically separated sets of registers denoted data and control. Each set contains multiple kinds of registers: general-purpose registers for storing PE-internal intermediate data, feedback registers for storing recurring data, virtual registers for writing to multiple registers at once, and finally input and output registers that are connected to the surrounding interconnect wrapper to allow inter-PE communication. (A “?” shows that the number of available registers of a given kind is configurable at synthesis time.) Each functional unit receives up to two input operands and produces one result; depending on which register file the functional unit is connected to, it is either classified as a data (depicted as rectangles) or control functional unit (depicted as rounded rectangles). To allow for synchronization between functional units, all functional units share the same flags and control signals, logically stored with the control registers.
2.1 Architecture

Functional units

The set \( FU = \{fu_1, fu_2, \ldots \} \) of functional units a PE contains is configurable at synthesis time. The most common functional units are adders, multipliers, dividers (all three as either integer or floating-point), logic units (Boolean bit operations), comparators, and bit shifters. In addition, custom FUs (functional units) may be implemented, such as:

- Anytime dividers, adders, and multipliers [BWH+19; BWB+20]: functional units that realize anytime instruction processing to provide support for approximate computing. Anytime instructions encode the number of calculated most significant bits of the operation within the instruction to allow for programmable accuracy. For example, the instruction \( \text{div2 rd0 rd1 rd2} \) only computes the two most significant bits of the result. Not computing the full result can improve energy efficiency and the latency of the operation.

- Vector units that work on multiple values in parallel. For example, instead of adding one 32-bit value to another 32-bit value, the vector unit may add four 8-bit values to another four 8-bit values at once. This type of single instruction, multiple data (SIMD) processing exploits data-level parallelism inherent to many algorithms, such as found, for example, in image processing. These units improve energy efficiency because control flow is shared among multiple operations instead of repeated for each individual datum.

- Functional units that implement activation functions: CNNs (convolutional neural networks) often apply an activation function to each result pixel of a convolution that is not efficiently realizable in software, such as ReLU or the hyperbolic tangent function [ABH+20]. By providing efficient hardware implementations of these functions, performance and energy efficiency can be improved.

Each FU is assigned a mnemonic identifier that is unique within the context of a PE. For example, in

\[ FU = \{\text{add0}, \text{add1}, \text{mul} \}, \]

\( \text{add0} \) and \( \text{add1} \) are two adders, and \( \text{mul} \) is a multiplier.

Each FU \( fu \) implements a specific instruction set\(^1\) \( IS_{fu} = \{\text{instr}_1, \text{instr}_2, \ldots \} \). Although an FU implementation may define a custom instruction format, it is usually based on three-address code:

\[ \text{mnemonic} \text{ destination operand1 operand2} \]

The instruction \( \text{add od0 rd1 id2} \), for example, adds the value of \( \text{rd1} \) to the value of \( \text{id2} \) and stores the result in \( \text{od0} \). Three-address code aligns well with the fact that each FU is connected to up to two read ports and one write port of one of the two register files (see

\(^1\)The instruction set of functional units may be parameterized.
Figure 2.3). For example, an adder $\text{add0}$ may have the following instruction set, where each instruction is represented using an instruction template:

$$IS_{\text{add0}} = \{ \text{add } rd \text{ rs } rt, \text{addi } rd \text{ rs } imm, \text{sub } rd \text{ rs } rt, \text{subi } rd \text{ rs } imm, \ldots \},$$

where $rd$, $rs$, $rt$, and $imm$ are operand placeholders for register operands and immediate operands, respectively. (During code generation, the actual operands are matched to these placeholders to select an adequate instruction.) A functional unit instruction has two properties of note: the pipeline interval $\pi_{\text{instr}}$, which is the delay necessary between starting $\text{instr}$ and starting the next instruction, and the latency $l_{\text{instr}}$, which is the duration in cycles after which the result of the instruction will have been written back to the destination register. Additionally, a functional unit may provide specific flags that are set by these instructions; for example, adders offer carry, negative, zero, and overflow flags, while comparators offer less-than and equal-to flags.

As stated above, the PEs implement OIP (orthogonal instruction processing), where in contrast to VLIW processors, each FU has its own instruction memory and branch unit. This separation of functional unit programs can reduce program size and thus the required size of instruction memories [BHT’17]. Despite this logical separation, synchronization of control flow between all functional units is achieved by sharing flags and control signals, both accessed through the control register file, among all FUs. For example, one functional unit may branch on the carry flag of another functional unit or, as another example, all functional units may wait on a control signal from the global controller to synchronize globally.

To achieve this, each OIP instruction is a pair of a functional instruction (as described above) and a branch instruction that controls the branch unit. There are three basic branch instructions:

1. Conditional multi-way branch instructions:

$$\text{jmp } target_0, \ldots, target_{2^N-1} \text{ if } cond_0, \ldots, cond_{N-1}$$

A target denotes an address within the instruction memory. A condition $cond$ can either be a control signal, such as $ic0$, or a functional unit flag, such as $\text{add0.}c$ when branching on the carry flag of functional unit $\text{add0}$. A multi-way branch sets the program counter to one of the targets as follows: The binary values of the 1-bit conditions are concatenated into an $N$-bit vector and interpreted as the index into the vector of targets. For example,

$$\text{jmp } 12, 15, 3, 5 \text{ if } ic0, \text{add0.}c$$

sets the program counter to 12 if both $ic0$ and $\text{add0.}c$ are 0, to 15 if $ic0$ is 1 and $\text{add0.}c$ is 0, to 3 if $ic0$ is 0 and $\text{add0.}c$ is 1, and to 5 if both are 1.

Note that while the number of simultaneous conditions in a condition branch instruction is a synthesis-time parameter, having only one is the most common choice. This is equivalent to a conventional processor's branch instructions.
2. Unconditional branch instructions:

\[ \text{jmp target} \]

An unconditional branch sets the program counter to \textit{target}.

3. Next instructions:

\[ \text{next} \]

A next instruction increments the program counter, passing control flow sequentially to the next instruction.

Consequently, the instruction memory of a FU contains pairs of functional and branch instructions. When executing an instruction, the next pair addressed by the program counter is read from the instruction memory. Then, both the functional instruction (e.g., \text{add od0 rd1 rd2}) and the branch instruction (e.g., \text{next}) are executed in parallel. When the functional instruction has finished execution (after \(l_{\text{instr}}\) cycles), the next instruction is read from the instruction memory, addressed by the program counter as updated by the just executed branch instruction.

### 2.1.2 Interconnect network

The PEs in a TCPA are connected via a circuit-switched interconnect network whose topology is reconfigurable at run time [KHK+06]. To realize this reconfigurability, each PE \( p = (r, c)^T \) is embedded into an \textit{interconnect wrapper} \( w = (r, c)^T \) that acts as a switch between its up to four neighbors and the input/output ports of the PE within. Figure 2.4 shows their physical layout: For each semantic layer (data and control) and for each cardinal direction (north, south, west, east), an interconnect wrapper has a configurable number of output and input ports called \textit{directional ports}. Each directional port is connected to the corresponding port in the neighboring interconnect wrapper. For example, the first east output port of \( w \) is connected to the first west input port of the eastern neighbor \( w' = (r, c + 1)^T \). Directional ports at the border of the processor array that are not connected to another interconnect wrapper are instead connected to an I/O buffer. Additionally, there is a configurable number of internal input and output ports connected to the corresponding registers of the embedded PE, simply called \textit{PE ports}. To summarize, a port \textit{port} is a quadruple \((\text{semantics, location, direction, } i)\), where

- \( \text{semantics} \in \{\text{data, control}\} \)
- \( \text{location} \in \{\text{north, east, south, west, } \text{pe}\} \)
- \( \text{direction} \in \{\text{input, output}\} \)
- \( i \in \mathbb{N}_0 \).

If the context is clear, we omit the \textit{location} and \textit{semantics} of a port and write, for example, \text{north}_0 for short. In situations where ports of different wrappers need to be distinguished, we write \((w, \text{port})\) to identify port \textit{port} of wrapper \( w \).
Figure 2.4: Physical layout of an interconnect wrapper containing a PE. Each interconnect wrapper has up to four neighbors—north, south, west, east—, which is why there are directional ports located at the four sides of the wrapper. In particular, there are a configurable number of input and output ports, both for data signals (ports depicted as rectangles) and control signals (ports depicted as rounded rectangles). These directional ports can be either connected to each other (for creating varying topologies) or to the input and output ports located at the PE, also for data and control signals. Note that two ports can only be connected if they are configured to be adjacent at synthesis time. In the figure, three adjacencies are illustrated using gray lines. The PE ports are each connected to an input or output register of the PE.
In the context of an interconnect wrapper, each port has an orientation: either it is a source or a sink port. As a short-hand, we denote source ports with a right-facing triangle, e.g., \( \text{north} \triangleright \), and sink ports with a left-facing triangle, e.g., \( \text{north} \triangleleft \). A port is a source port if it provides data—that is, all directional input ports and the PE output ports—and a sink port if it consumes data—that is, all directional output ports and the PE input ports. At run time, two ports can be connected if they have the same semantics, but opposing orientation, and have been configured to be adjacent at synthesis time. Adjacency for an interconnect wrapper \( \mathbf{w} \) is configured using a semantics-specific adjacency matrix

\[
A_{r,c}^{\text{semantics}} = (a_{i,j}) \in \{0, 1\}^{\#\text{sinks} \times \#\text{sources}},
\]

where each sink port \( \triangleleft i \) is adjacent to source port \( \triangleright j \) if \( a_{i,j} = 1 \). For example, a crossbar in the data layer, as shown in Figure 2.5, is constructed using the adjacency matrix with only 1s:

\[
A_{r,c}^{\text{data}} = (a_{i,j} = 1).
\]

By utilizing this synthesis configurability and run-time reconfigurability, PEs can be connected using various topologies, including, for example, two-dimensional tori, meshes, and three-dimensional hypercubes.

### 2.1.3 Global controllers

A TCPA may be configured with up to four global controllers, one per corner, each responsible for synchronizing a single loop program. Consequently, the maximum number of concurrently running loops and thus the degree of task parallelism of a TCPA is equal to the number of global controllers. Each global controller is connected to control ports\(^2\) of its corresponding corner PE (compare Figure 2.1), which further

\(^2\text{Usually, a global controller is connected to the north control ports for the northern two corners and the south control ports for the southern two corners.}\)
Tightly Coupled Processor Arrays

Figure 2.6: A global controller contains two components: an iteration space scanner that sequentially scans the iteration space $I$ of a loop and a control signal generator that generates control signals from the current iteration $i \in I$ according to the control flow of the loop.

The control signals are generated according to the static control flow of the executing loop, which, in the polyhedral model, only depends on the current iteration vector $i$. This is reflected in the architecture of a global controller as illustrated by the block diagram in Figure 2.6: There is an iteration space scanner with a configurable maximum dimension of $n$ that sequentially scans a rectangular iteration space $I \subset \mathbb{Z}^n$, keeping track of $i \in I$. From the iteration vector $i$, a control signal generator generates a configurable number of control signals. Each control signal is 1 if $i$ is in a given subset of $I$. Hence, the control signal generator implements containment checks for (not necessarily convex) polyhedral sets.

Remark. Both the maximum dimension as well as the maximum number of control signals determine the complexity of the loops that the global controller supports and must therefore be carefully chosen. For example, a convolution layer of a CNN is a 6-dimensional loop, meaning the allocated global controller must be configured with $n \geq 6$ to support it. Similarly, a loop program with many complex iteration-dependent conditions requires the global controller to support a large number of concurrent control signals.

2.1.4 I/O buffers

Memory design in a dedicated accelerator requires special attention: Not only is the required high bandwidth to exploit all of a loop’s potential parallelism often the bottleneck for performance, but especially dynamic memories (such as DDR memory) usually have the worst energy proportionality in a system [MSG+12], while transfers over shared buses and NoCs (network-on-chips) may introduce undesired uncertainty [CFG+10]. To mitigate all three concerns, which are in direct opposition to the three design goals stated in the beginning, the synchronous data accesses of a TCPA are decoupled from the rest of the system by four I/O buffers, one at each border. In particular, the concerns regarding memory design are addressed as follows: Adding an additional level to the memory hierarchy and synchronously generating addresses helps reduce uncertainty; providing a sufficient number of access ports helps to satisfy the I/O bandwidth requirement; and,
2.1 Architecture

Figure 2.7: There is an I/O buffer on each of the four borders of the processor array. Each is connected to a wide internal data bus that is in turn connected to an external bus via an interface unit that decouples the system bus (for example, AMBA or AXI) from the TCPA internal bus.

keeping the I/O buffers small and local to the processor array helps to improve energy efficiency. Of course, trade-offs must be made, which inspired the following design choices, and which can be fine-tuned at synthesis time using the given parameters.

As shown in Figure 2.7, each of the four I/O buffers is connected to both sides of the system: to the processor array via the corresponding border PEs and to the rest of the system via an internal data bus. The internal data bus connects all four I/O buffers to an interface unit facing the rest of the system, usually through a bus such as AMBA/AXI (ARM) or QPI (Intel). Another option is connecting to a serialization/deserialization unit that interfaces with multiple high-speed Gigabit transceivers (such as Xilinx’ Aurora). The configurable width of the internal data bus in words aims to provide sufficient bandwidth to fill and drain buffers in time without stalling the TCPA. Obviously, attaining this goal also depends on the external connection, which must be able to sustain a sufficient bandwidth and, if timing predictability is required, provide means for predictable transfers, such as TDMA (time division multiple access). To flexibly yet cost-efficiently connect the I/O buffers to the PEs of the processor array, each I/O buffer is divided into modules (as shown in Figure 2.8).

Modules, Partitions, and Banks

Each I/O buffer consists of a configurable number of modules, each containing a configurable number of partitions and a configurable number of memory banks that allow random access. A partition provides independent access to a reconfigurable subset of the banks, which constitute the actual memory of the module.

Within the context of this dissertation, not all features of an I/O buffer are required,

3Requiring this is reasonable because timing predictability must be tackled as a system-wide property anyway.
4In actuality, I/O buffers can be connected much more flexibly to PEs. Furthermore, they offer features such as a line buffer mode.
which is why we consider a simplified view, as illustrated in Figure 2.8. The north and south I/O buffers have as many modules as there are columns \( C \) of PEs; the west and east I/O buffers have as many modules as there are rows \( R \) of PEs. Each module has \( P \) partitions, where each partition of a module is connected to a pair of an input and output data port of the adjacent border PE. For example, partition 1 of module 3 of the north I/O buffer is connected to both north\(_{1}^{C}\) (PE output port to partition input port) and north\(_{1}^{C}\) (partition output port to PE input port) of PE \( p = (0, 3)\). This allows each border PE \( P \) simultaneous data accesses.

Within each partition, access to a subset of the banks is arbitrated by an address generator. An address generator generates affine addresses\(^5\) from the current iteration \( i \) according to

\[
\text{address} = C \cdot i + c,
\]

where \( C \in \mathbb{Z}^n \) are reconfigurable address coefficients and \( c \in \mathbb{Z} \) is a reconfigurable base address. Because the banks within a module form a contiguous address space, the generated addresses may span multiple banks. Therefore, by reconfiguring an address generator accordingly, each partition can allow access to not only one bank, but a subset of banks.

\textbf{Remark.} The above organization and address generation scheme can not only be used for making a partition offer configurable memory space, but can also be used for double

\(^5\)For a comparison of different alternative address generation schemes, refer to [Bop15].
2.2 TCPA configuration data

buffering. Double buffering helps to hide transfer times by filling or draining one bank while working on another and then switching between these two.

The current iteration vector $i$ is generated by an internal iteration space scanner similar to the one in a global controller. In addition, an enable signal is generated according to

$$
enable = \begin{cases} 
1 & \text{if } i \in I_{io} \subseteq I \\
0 & \text{otherwise}
\end{cases},
$$

where $I_{io}$ is a subset of the iteration space where the generated addresses are considered valid (that is, where the connected PE is expected to access data).

2.1.5 TCPA synthesis flow

See Appendix B for an overview of the synthesis flow from architecture description to target generation.

2.2 TCPA configuration data

While this thesis describes methods to compile loop programs for a TCPA symbolically—that is, to re-emphasize, in a way that leaves the loop bounds and number of allocated PEs adjustable until run time—, the resulting symbolic configuration of a compiled loop cannot be used to configure a synthesized TCPA directly. Once bounds and number of allocated PEs are known, it must rather be instantiated into concrete run-time configuration data that contains loop-specific data for all involved hardware components:

- **Processing elements**: Each allocated PE is configured with a compact, software-pipelined OIP program containing the instructions of the loop’s iterations that are mapped to the PE. Additionally, it is configured according to the corresponding loop schedule: the feedback registers with the temporal distances of loop-carried dependences and the control input registers with the appropriate synchronization delay between PEs.

- **Interconnect network**: The allocated interconnect network region is configured with the topology required for the propagation of inter-processor dependences, as well as for accessing the I/O buffers.

- **Global controller**: The global controller is configured with the loop’s static control flow to generate control signals for synchronizing the PEs and for orchestrating the local control flow within each PE program.
• **I/O buffers and address generators**: The address generator of each allocated I/O partition is configured according to the data accesses within the program of the connected PE and the memory layout of the associated input or output array.

How these high-level concepts correlate to the low-level configuration data is explained in the next sections, mostly based on previous work [Bop15].

### 2.2.1 Processing elements

A PE contains two types of sub-components that require run-time configuration. First, the functional units are programmable, requiring a program, and second, the register files include reconfigurable (feedback) shift registers, requiring their depth specified.

#### Program

As explained in Section 2.1.1, the PEs use OIP to reduce the average size of programs and thus the required size of instruction memories by having each functional unit execute an individual functional unit program. Therefore, an OIP program assigns a functional unit program to each functional unit of a subset $\mathcal{FU} = \{fu_1', fu_2', \ldots\}$ of a PE’s functional units $\mathcal{FU}$. Each functional unit program is a sequence of OIP instructions, resulting in the following syntactical structure of an OIP program:

```
program "<name>" {
  fu_1': {
    instruction "<label>" {
      <functional instruction>
      <branch instruction>
    }
    // ...
  }
  fu_2': {
    // ...
  }
  // ...
}
```

As explained previously, each instruction of a functional unit program is a pair of a *functional instruction*, specifying the operation performed by the functional unit, and a *branch instruction*, specifying how the branch unit determines the address to be executed next. As is conventional, the targets of a branch instruction can be specified using labels. For example,

```
instruction "propagate" {
  move od0 rd0
```
jmp propagate, next if ic0
}

keeps propagating rd0 to od0 until a control signal from ic0 is received. Then, the
control flow passes to the next instruction, indicated by the special label next. Note
that FU programs are orthogonal and therefore labels are only valid within the scope of
a single FU program.

Remark. PE programs must be both compact (because the instruction memory per
functional unit is limited) and heavily software-pipelined (to exploit as much parallelism
as possible, making the computation fast and energy-efficient). Both challenges are
addressed in this thesis: the latter using symbolic modulo scheduling in Chapter 4, the
former during compact program instantiation in Chapter 6.

Feedback and shift register depths
The data feedback registers fdn of a PE are used to store intermediate results communi-
cated along uniform loop-carried dependences. Uniform means that each loop-carried
dependence has a fixed spatial distance within the loop’s iteration space and recurs every
iteration. In other words, in each iteration, a new value is produced that is consumed a
fixed number of iterations later. This temporal distance—which depends on the execution
order of the iterations—corresponds to the depth of the allocated feedback register,
which then allows all produced intermediate values until they are consumed and pro-
vides the stored values at the correct point in time. For example, if every iteration a new
value is produced that is consumed after 16 iterations with a length of 1 cycle each, the
allocated feedback register is reconfigured with a depth of 16.

The control input registers icn, on the other hand, are used to delay control signals
generated by the global controller. Delaying is necessary because not all allocated PEs
start at the same time, but rather in a staggered fashion. The depths of the control input
registers are configured accordingly.

Remark. In contrast to the control input registers, the data input registers are organized
as FIFOs and therefore do not need their depths configured. The enable signals generated
by any involved address generators ensure that the data in the FIFOs is consistent with
the executing PE program.

2.2.2 Interconnect network
Within the context of symbolic loop compilation, the main responsibilities of the inter-
connect network are to (a) propagate loop-carried dependences from PE to PE, and (b)
connect PEs performing I/O to the I/O buffers. For (a), each loop-carried dependence
in a loop program is assigned a set of propagation channels (see Section 4.9.4), while for
(b), a set of input channels and output channels is allocated (see Section 6.5). Since each
interconnect wrapper is connected to its direct neighbors, routes connecting PEs can be established as long as the configured adjacencies are respected; for example

$$( \mathbf{w} = (0, 0)^\top, \text{pe}_0^{\bowtie} ) \rightarrow ( \mathbf{w}, \text{east}_0^{\bowtie} ) \rightarrow ( \mathbf{w}', (0, 1)^\top, \text{west}_0^{\bowtie} ) \rightarrow ( \mathbf{w}, \text{pe}_0^{\bowtie} ),$$

which is also illustrated in Figure 2.9. This example requires $\text{pe}_0^{\bowtie}$ to be adjacent to $\text{east}_0^{\bowtie}$ in PE $\mathbf{(0, 0)^\top}$ and $\text{west}_0^{\bowtie}$ to be adjacent to $\text{pe}_0^{\bowtie}$ in PE $\mathbf{(0, 1)^\top}$. Finding these routes is complicated by the unknown number of PEs during symbolic compilation; a solution is provided in Section 4.9.4.

### 2.2.3 Global controller

To synchronize the PEs executing a loop, the global controller assigned to its orchestration generates control signals $c_s_i$ according to the loop’s control flow, which is inferred from a sequential scanning of the iteration space$^6$. Accordingly, the configuration of the global controller has two parts: The first consists of the iteration space and the loop’s initiation interval $\pi$ (the period with which iterations are started, see Section 3.4.4 and Chapter 4). The second consists of the one-domains $O_i$ of the allocated control signals $c_s_i$. Each one-domain $O_i \subset I$ is the not necessarily convex subset of iterations where the control signal $c_s_i$ shall take the value 1 and is determined from the loop’s control flow (see Section 6.3 about control signal allocation).

### 2.2.4 I/O buffers—Address generators

The configuration of the I/O buffers is derived from the input and output accesses of the mapped loop program (see Section 6.5). An access $x[\phi(i)]$, $\forall i \in I_x$ of an $m$-dimensional I/O variable $x$ is characterized by an access domain $I_x$ — the subset of iterations where the access takes place — and an affine indexing function $\phi(i) : I_x \mapsto \mathbb{Z}^m$ that maps an

---

$^6$To be specific, the global controller is configured according to the *intra-tile schedule* after tiling; see Section 3.4.
2.3 Integration

iteration $i$ to an index vector (see Section 3.1). Each PE that executes at least one iteration within $I_x$ requires individual, usually concurrent access to read or write $x$ and, therefore, the allocation of an individual I/O partition within an I/O module connected to the PE. The address generator within an allocated partition is configured with the access domain $I_x$ and affine indexing function $\phi(i)$ of the corresponding access $x[\phi(i)]$. Furthermore, it is configured with a start delay derived from the loop schedule that synchronizes the generated addresses to the program of the connected PE.

2.2.5 Configuration flow

See Appendix B for an overview of the flow from a symbolic configuration to the run-time configuration of a synthesized TCPA.

2.3 Integration

TCPAs are loop accelerators and therefore only part of a larger system, making it imperative to look at how to integrate TCPAs with other components into a system. A TCPA requires two types of bidirectional communication with the rest of the system: (1) transferring large amounts of input and output data from data sources (usually memories or sensors) and to data sinks (usually memories); (2) controlling the TCPA, which includes configuring it, starting it, and otherwise controlling it. Here, two scenarios are elaborated: Multi-core systems where the components are connected with a shared bus, and tiled multi-processor systems-on-chip (MPSoC) where each component occupies a tile within a grid of tiles connected using an NoC.

2.3.1 Multi-core systems with a shared bus

Conventional multi-core systems provide a shared system bus that all components connect to, be it multi-core processors, memories, or accelerators such as GPUs or in this case, TCPAs. Figure 2.10 highlights the bus’ role for both types of communication mentioned above: High-bandwidth DMA is used to transfer input and output data to and from the TCPA, while memory-mapped registers are used to control and configure it. (Note that sensors, such as cameras, microphones, or radars, may also be directly connected to a TCPA to save bandwidth on the bus and facilitate achieving predictability.)

One example of such a multi-core system is the Zynq SoC (system-on-chip) family developed by Xilinx. For instance, the Zynq-7000 SoC features a dual-core ARM Cortex A9 processor coupled to an FPGA, which is connected to the system bus using the AXI protocol. If a TCPA is loaded onto the integrated FPGA, loop acceleration facilities are provided to the two ARM cores. The TCPA can then interface over the AXI bus with the ARM processor, main memory, and any connected peripherals.
Figure 2.10: A multi-core system where a TCPA is integrated by connecting it to a system bus, which is used for both transferring I/O data and controlling the TCPA. Additionally, sensors may be connected directly to a TCPA to unburden the bus.

### 2.3.2 Tiled MPSoCs with a network-on-chip: Invasive Computing

Tiled MPSoCs aim to mitigate the complexity caused by the ever-increasing number of transistors, and thus components, per chip: As illustrated in Figure 2.11, each component is embedded into a tile, all of which are connected with an NoC. Decoupling components this way makes systems scalable and easily extensible with more and new components, for example one or multiple TCPAs. In such a system, a TCPA tile contains not only the accelerator itself, but also a complementary CPU responsible for directly controlling the TCPA; other tiles may then control the TCPA indirectly through this CPU. Data, on the other hand, can be directly streamed in from other tiles over the NoC, such as dedicated memory tiles.

Still, managing an increasing number of resources across multiple running applications is a difficult problem, imposing many challenges on all parts of a system (compiler, operating system, hardware, …). One contender to solve these challenges comprehensively is Invasive Computing, a paradigm for resource-aware programming [Tei08; THH+11] where applications cycle through three phases:

1. **invade**: During the **invade** phase, an application may request resources, such as processors, TCPAs, or memory, from the run-time system and if successful, obtains a **claim** to these resources. The application then has exclusive access to the resources within its claim. Exclusivity facilitates ensuring non-functional properties, such as timing predictability in particular.

2. **infect**: During the **infect** phase, the application loads programs and the necessary data onto the resources within its claim and executes the programs.

3. **retreat**: During the **retreat** phase, the application returns the resources within its claim to the run-time system, making them available to other applications again.

TCPAs cooperate very well with Invasive Computing because they allow fine-granular allocation of PEs. Multiple applications from the MPSoC (multi-processor system-on-
chip) may claim regions of PEs on a TCPA and dynamically react to the number of PEs they are actually granted. To this effect, Invasive TCPAs provide some additional facilities [LHT11; LNH+11; Lar15]. To allow for efficient invasion of a TCPA, each PE of an invasive TCPA additionally contains an invasion controller that keeps track of the current invasion state and forwards invasion requests to its neighbors cycle by cycle. There has been research regarding different designs of invasion controllers and different linear and rectangular invasion strategies [AMA+09; LHT11; LNH+11].

Using these facilities together with Invasive Computing and symbolic loop compilation allows, for example, to implement adaptive fault tolerance [WTT+15; LTT+15]: The degree of modular redundancy (e.g., dual or triple) is changed dynamically according to measurements of the environment, such as the radiation, to achieve a given required mean time to failure. The requested modular redundancy is realized by replicating a
given loop program accordingly across the invasive TCPA \cite{TWT15}; for example, for triple modular redundancy, thrice the PEs are allocated and the loop is executed three times in lock-step. Configurable voters then check the results of the replicas against each other and signal detected faults to the TCPA driver (see next section). This is an example of run-time requirement enforcement, which is further discussed in Chapter 7.

2.3.3 Driver

Regardless of the manner of integrating a TCPA into a system, many of the required functionalities are shared, such as instantiation of symbolic configurations, configuring the TCPA hardware, keeping track of allocated PEs, and so on. To improve maintainability, to provide a common interface, and to decouple these core functionalities from applications and the operating system, they are collected within a driver. However, many functionalities also differ, such as the method of communication with other components in the system, the programming interface (for instance, direct vs. Invasive Computing), and so on. These are implemented on top of the driver on a per-system basis.

2.4 Related work

For an extensive overview of architectures similar to TCPAs and how they are programmed, see Appendix C.
Efficiently utilizing the extensive number of resources in a TCPA requires a compiler to perform meticulous loop parallelization. Parallelization is a transformation that rearranges the operations in a given loop into an order that maximizes the parallelism between them while maintaining semantic equivalency with the original loop. Semantic equivalency is achieved by preserving the data dependences between operations, making dependence analysis a crucial prerequisite to loop parallelization.

Dependence analysis faces unique challenges within loops since they model iterative control flow. Each statement in a loop body is executed repeatedly, requiring the compiler to detect the dependences between statement instances, not only their syntactical representations (such as nodes in an abstract syntax tree). The polyhedral model [FL11; Len93] has been developed over the last few decades with that in mind: Each statement instance is identified by an integer point called iteration (vector) within the iteration space of a loop, which is based on so-called integer polyhedra [GR07]. The close relation between polyhedra and systems of inequalities enables the use of formal methods such as linear algebra and integer linear programming, thus facilitating analyses, such as dependence analysis, and transformations, such as parallelization.

However, the polyhedral model does not prescribe how to model the body, that is, the statements themselves, of a loop. We chose a representation that emphasizes the advantages of systolic architectures: UDAs (uniform dependence algorithms), derived from systems of uniform recurrence equations [KMW67], not only have favorable properties for parallelization—for example, there are only flow dependences—but also for mapping in general—for example, uniform dependences exclusively result in local communication between neighboring PEs. Furthermore, each UDA has a graph representation called a RDG (reduced dependence graph) that is a practical intermediate representation we use throughout the thesis.

Roadmap. Section 3.1 situates UDAs within the class of so-called recurrence algorithms and explains the related basic concepts. Section 3.2 formalizes Presburger sets, the notation used throughout this dissertation to describe parametric iteration, index, and condition spaces in the polyhedral model. The definition of RDGs is presented in Section 3.3. Finally, Section 3.4 about allocation and scheduling of reduced dependence graphs
prepares the foundation of how UDAs are mapped onto TCPAs. Chapters 4 and 5 build upon that foundation to symbolically schedule UDAs and generate a symbolic TCPA configuration.

3.1 Uniform dependence algorithms

UDAs (uniform dependence algorithms) belong to the class of algorithms called recurrence algorithms. Recurrence algorithms model iterative computations as a set of recurrence relations between multi-dimensional indexed variables, imposing a mathematical structure that facilitates analysis and transformation.

Definition 3.1. An $m$-dimensional indexed variable $x$ is a structured arrangement of variable instances on an index space $I_x \subseteq \mathbb{Z}^m$. Each index vector $i \in I_x$ corresponds to an individual instance, denoted $x[i]$ or $x[i_1, i_2, \ldots, i_m]$, assuming $i = (i_1, i_2, \ldots, i_m)^T$. The elements $i_1, i_2, \ldots$ of the index vector are called index variables.

Example. An audio signal may be modeled as a one-dimensional indexed variable $u$, where each instance $u[i]$ represents the amplitude of the $i$-th sample; an RGB image may be modeled as a 3-dimensional indexed variable $rgb$, where each instance $rgb[x, y, c]$ represents the channel value of the pixel at position $(x, y)$ and channel $c$. As a special case, we model scalar variables (that is, single-instance variables) as zero-dimensional indexed variables, for example $s[]$, and omit the brackets for clarity, simply writing $s$.

Recurrence algorithms are a generalization of systems of uniform recurrence equations (SUREs), which were introduced in the seminal work of Karp, Miller and Winograd to study the parallelism inherent in iterative computations [KMW67].

Definition 3.2. An $n$-dimensional recurrence algorithm $\mathcal{R}$ is a set $\{S_1, S_2, \ldots\}$ of recurrence equations quantified over an iteration space $I \subseteq \mathbb{Z}^n$ that interrelate the instances of a set $X$ of indexed variables. Each equation is of the form

$$S_i: x_i[\psi_i(i)] = f_i(\ldots, y_{ij}[\phi_{ij}(i)], \ldots) \text{ if } i \in I_i, \quad 1 \leq j \leq \text{arity}(f_i),$$

(3.1)

where $x_i \in X$ is the indexed variable with instances defined by $S_i$ as the value of the function $f_i$, and $y_{ij} \in X$ are the indexed variables with instances used by that definition as arguments; we say $x_i$ depends on $y_{ij}$. The vector $i$ is called the iteration (vector), its elements iteration variables. The equation is only defined for iterations $i \in I \cap I_i \neq \emptyset$, where $I_i \subseteq \mathbb{Z}^n$ is called the condition space of the equation. Both $\psi_i: \mathbb{Z}^n \mapsto \mathbb{Z}^{m_{\psi_i}}$ and $\phi_{ij}: \mathbb{Z}^n \mapsto \mathbb{Z}^{m_{\phi_{ij}}}$ are called indexing functions and map the iteration vector to an index.

---

1In this thesis, we differentiate between indexed variables and arrays: An indexed variable is a mathematical, abstract construct, whereas an array is a concrete data structure with physical size and type.
vector of the corresponding dimension $m_{x_i}$ resp. $m_{y_{i,j}}$. No instance of an indexed variable may be defined more than once:

$$S_i \neq S_j \land x_i = x_j \implies \forall i \in I_i \cap I_j, j \in I_j \cap I : \psi_i(i) = \psi_j(j)$$

This is analogous to the array single-assignment form.\(^2\)

Three reasons make recurrence algorithms particularly amenable to automatic parallelization. First, the array single-assignment property ensures that each use of a variable instance has exactly one corresponding definition, which facilitates dependence analysis. Second, recurrence algorithms are referentially transparent: the meaning of an instance of an equation $S$ does not depend on the surrounding context.\(^3\) Referential transparency hence ensures that, safe for satisfying data dependences, the execution order can be freely chosen [Kes96]. For recurrence algorithms, this is because each recurrence equation merely defines a set of functional dependencies of indexed variables without any prescribed execution order. Finally, both indexing functions and iteration spaces are mathematical objects—functions and sets, respectively—enabling mathematical reasoning.

**Remark.** The iteration space, the condition spaces, and the indexing functions of a recurrence algorithm in Definition 3.2 may depend on parameters, which represent integer values that may remain unknown prior to execution, but remain constant during. Technically, they define a family of recurrence algorithms, one for each value combination of the parameters. For example, the parameterized iteration space

$$I(N) := \{ i | 0 \leq i < N \}$$

defines a different iteration space for each $N$:

$$I(0) = \{ \}, I(1) = \{0\}, I(2) = \{0, 1\}, \ldots$$

However, to avoid cluttering the notation, instead of $\mathcal{R}(N)$ and $I(N)$, we simply write $\mathcal{R}$ and $I$, treating the parameters as a part of the definition. This aligns with our goal to symbolically compile recurrence algorithms—we are not interested in specific parameter values until instantiation.

We will see in Section 3.4 that such a compilation involves defining an order of computing the right-hand side functions $f_i$ and assigning the results to the left-hand side indexed variables such that each instance has been computed before its use. Any such order is called a schedule. Apart from scheduling, a mapping of operations to processor elements and functional units must also be defined. Both tasks are together known as space-time mapping and will be described in Section 3.4.

---

\(^2\) The array single-assignment form is an extension of the static single-assignment form used in most compilers where the $\phi$-function considers data flow between individual array elements [KS98].

\(^3\) Compare this to statements in sequential languages such as C, where the meaning of a statement may change, for example, because the value of a global variable depends on the context.
The following example introduces the notational conventions we use for recurrence algorithms and motivates putting restrictions on the indexing functions.

**Example.** The one-dimensional recurrence algorithm below defines the computation of the arithmetic mean $y$ of the pixel values of a two-dimensional grayscale image $img$ along the circumference of a circle with radius $r$ using $N$ samples, where $i$ denotes the current sample:

$$S_1: \quad p[i] = \text{img}[\text{round}(r \cdot \cos \frac{2\pi}{N} i), \text{round}(r \cdot \sin \frac{2\pi}{N} i)]$$

$$S_2: \quad x[i] = p[i] \quad \text{if } i = 0$$

$$S_3: \quad x[i] = x[i - 1] + p[i] \quad \text{if } i > 0$$

$$S_4: \quad y = x[N - 1]/N \quad \text{if } i = N - 1$$

The iteration space is $I = \{i = (i) \mid 0 \leq i < N\}$ with a single parameter $N$. The set of indexed variables is $X = \{p, x, y\}$.

Each equation in this example can be given in the notation of Equation (3.1), for example $S_3$:

$$S_3: \quad x_3[\psi_3(i)] = f_3(y_{3,1}[\phi_{3,1}(i)], y_{3,2}[\phi_{3,2}(i)]) \text{ if } i > 0, \text{ where}$$

$$x_3 := x, \psi_3(i) := i,$$

$$f_3(a, b) := a + b,$$

$$y_{3,1} := x, \phi_{3,1}(i) := i - 1,$$

$$y_{3,2} := p, \phi_{3,2}(i) := i$$

While denser and arguably less intuitive, all constituents in this notation can be unambiguously referenced and therefore used during compilation. We will use this detailed notation, for example, when constructing reduced dependence graphs (see Section 3.3).

The above example also illustrates three important points. First, multiple equations may define the same variable, but at disjoint instances. Here, $x[0]$ is defined by $S_2$, while $x[i]$ for $1 \leq i < N$ is defined by $S_3$. Second, if an instance is not defined by the algorithm, it is an implicitly assumed input; Similarly, if it is defined, but not used, it is an implicitly assumed output. Here, all used instances of $img$ are inputs, and $y$ is the single output. Third, allowing arbitrary indexing functions, such as the sine, cosine, and rounding in $S_1$, is not practical: Both inversion and composition are necessary for analyzing the data flow (for example: which pixels from $img$ does $r[3]$ depend on?), but may be difficult or impossible for some functions. By contrast, affine expressions such as $i - 1$ are easy to invert, easy to compose, and also much easier to represent and compile to regular structures of processor arrays by a compiler. \(\triangle\)

Restrictions on the indexing functions are therefore desirable. Depending on the kind of restrictions, different sub-classes of recurrence algorithms emerge. Zehendner
[Zeh96] gives a comprehensive survey of different classes, but here we focus on two that are particularly well suited for mapping to TCPAs: ADAs (affine dependence algorithms), where all indexing functions are affine functions, and UDAs, where all indexing functions are translations.

Definition 3.3. An \( n \)-dimensional affine dependence algorithm \( \mathfrak{A} \) is an \( n \)-dimensional recurrence algorithm where all indexing functions are affine in the iteration vector. In particular, \( \psi_i(i) \coloneqq Q'_i i + d'_i \) and \( \phi_{i,j}(i) \coloneqq Q_{i,j} i - d_{i,j} \):

\[
S_i: x_i[Q'_i i + d'_i] = f_i(\ldots, y_{i,j}[Q_{i,j} i - d_{i,j}], \ldots) \text{ if } i \in I_i \quad 1 \leq j \leq \text{arity}(f_i)
\]

We call \( Q'_i \) and \( Q_{i,j} \) indexing matrices, \( d'_i \) and \( d_{i,j} \) indexing offsets. Only \( d'_i \) and \( d_{i,j} \) may contain parameters.

ADAs appear under various names in the literature, most commonly as piecewise linear algorithms [TR91] and systems of affine recurrence equations [YC88; WS94; MP01].

Example (FIR filter). A finite impulse response filter (FIR filter) is used in signal processing to implement, for example, low-pass, high-pass or averaging filtering on an incoming signal. Mathematically, it is defined as

\[
Y_i = \sum_{j=1}^{N} A_j \cdot U_{i-j},
\]

where \( N \) is the number of so-called taps of the filter. The transformed signal is \( Y \), while \( A \) represents the filter coefficients and \( U \) the incoming signal. The following ADA implements a FIR filter for a signal segment of \( T \) samples:

\[
\begin{align*}
S_1: & \quad u[i, j] = U[i] \quad \text{if } j = 0 \\
S_2: & \quad u[i, j] = 0 \quad \text{if } j > 0 \land i < j \\
S_3: & \quad u[i, j] = u[i - j, 0] \quad \text{if } j > 0 \land i \geq j \\
S_4: & \quad x[i, j] = A[j] \cdot u[i, j] \\
S_5: & \quad y[i, j] = x[i, j] \quad \text{if } j = 0 \\
S_6: & \quad y[i, j] = y[i, j - 1] + x[i, j] \quad \text{if } j > 0 \\
S_7: & \quad Y[i] = y[i, j] \quad \text{if } j = N - 1
\end{align*}
\]

The iteration space is \( I = \{(i, j)^T \mid 0 \leq i < N \land 0 \leq j < T\} \). This ADA was formulated such that the signal samples \( U \) are "read in" when \( j = 0 \) (compare \( S_1 \)) and then recur for \( j > 0 \) (compare \( S_3 \)).\(^4\) The indexing matrix and offset of \( u[0, i - j] \) in equation \( S_3 \) are

\[
Q_{3,1} = \begin{pmatrix} 1 & -1 \\ 0 & 0 \end{pmatrix}, \quad \text{and } d_{3,1} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.
\]

\(^4\)The idea is that inputs and outputs are only read in or written out at the borders of a processor array.
The distance between the use of $u[0, i - j]$ in $S_3$ and the corresponding definition in $S_1$ is defined by
\[
\begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i - j \\ 0 \end{pmatrix} = \begin{pmatrix} j \\ j \end{pmatrix}.
\]
Thus, it grows with $j$ and depends on the current iteration $i$. If the iterations in $i$-direction were to be assigned to multiple PEs during parallelization, this might result in non-local communication with increasing hop counts, which is not possible in TCPAs with only connections between direct neighbors. Compare this to the indexing function of $y[i, j - 1]$ on the right-hand side of $S_6$:
\[
Q_{6,1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \text{and} \quad d_{6,1} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}
\]
Here, the magnitude of this dependence is constant regardless of the current iteration—it is always $d_{6,1}$. This convenient characteristic stems from $Q_{6,1}$ being the identity matrix, making the indexing function a translation; we call such dependences uniform. \(\triangle\)

Restricting recurrence algorithmsto only allowing translations as indexing functions yields uniform dependence algorithms.

**Definition 3.4.** An $n$-dimensional uniform dependence algorithm $\mathfrak{U}$ is an $n$-dimensional ADA where all indexing matrices are the identity matrix:
\[
S_i: x_i[i + d'_i] = f_i(\ldots, y_{i,j}[i - d_{i,j}], \ldots) \text{ if } i \in I_i, \quad 1 \leq j \leq \text{arity}(f_i),
\]
where $d_{i,j}$ denotes the dependence vector of the uniform dependence between $x_i$ and $y_{i,j}$. We use $D$ to denote the set of all dependence vectors occurring in equations of a UDA.

We will see that uniform dependences will lead to regular communications when introducing their mapping to TCPAs in Section 4.9.4. UDAs and similar algorithm classes have therefore been extensively studied in the literature, beginning with the seminal work on systems of uniform recurrence equations (SUREs) by Karp, Miller, and Winograd [KMW67] that inspired the field of recurrence algorithms. Other similar classes include regular iterative algorithms [Rao85] and piecewise regular algorithms [Thi88].

Any ADA may be transformed into a UDA using localization [Thi89], where affine dependences are transformed such that they are propagated locally from iteration to iteration.

**Example** (FIR filter, continuing from p. 37). Localizing the FIR filter (including $A$) from
3.1 Uniform dependence algorithms

the previous example yields the following UDA:

\[ S_1: \quad a[i, j] = A[i, j] \quad \text{if } i = 0 \]
\[ S_2: \quad a[i, j] = a[i - 1, j] \quad \text{if } i > 0 \]
\[ S_3: \quad u[i, j] = U[i, j] \quad \text{if } j = 0 \]
\[ S_4: \quad u[i, j] = 0 \quad \text{if } j > 0 \land i = 0 \]
\[ S_5: \quad u[i, j] = u[i - 1, j - 1] \quad \text{if } j > 0 \land i > 0 \]
\[ S_6: \quad x[i, j] = a[i, j] \cdot u[i, j] \]
\[ S_7: \quad y[i, j] = x[i, j] \quad \text{if } j = 0 \]
\[ S_8: \quad y[i, j] = y[i, j - 1] + x[i, j] \quad \text{if } j > 0 \]
\[ S_9: \quad Y[i, j] = y[i, j] \quad \text{if } j = N - 1 \]

The previously affine dependence \( u[0, i - j] \) has been localized to be propagated along \( d_{5,1} = (1, 1)^T \) in \( S_5 \). Due to the definition of UDAs, all indexed variables in \( X \) need to be \( n \)-dimensionally indexed. Note that although \( A, U, \) and \( Y \) represent 1-dimensional in- and output data, they are indexed 2-dimensionally in the above example. This restriction obfuscates the original dimensionality and necessitates additional steps to ensure the correct mapping between the 2-dimensional indexing and the actual 1-dimensional data. △

Therefore, the following definition lifts some restrictions of UDAs to form a practical compromise between fully affine and exclusively uniform indexing.

**Definition 3.5** (adapted from [WHT19]). An \( n \)-dimensional uniform dependence algorithm with affinely indexed inputs and outputs \( \mathcal{U} \) is an \( n \)-dimensional ADA where each indexed variable \( x \in X \) satisfies exactly one of the following conditions:

1. If instances of \( x \) are only used, but none are defined (intuitively: \( x \) only appears on the right-hand side), \( x \) is an input variable. Input variables may be indexed affinely.

2. If instances of \( x \) are only defined, but none are used (intuitively: \( x \) only appears on the left-hand side), \( x \) is an output variable. Output variables may be indexed affinely.

3. If all used instances of \( x \) are defined (intuitively: \( x \) appears on both sides), \( x \) is an internal variable. When defined, internal variables may only be indexed using the identity; when used, internal variables may only be indexed uniformly.

These conditions partition \( X \) into the set of input variables \( X_{in} \), the set of output variables \( X_{out} \), and the set of internal variables \( X_{var} \).

Any ADA can be transformed into this form using localization, embedding, and equation splitting. For brevity, UDA henceforth always refers to this definition.
Example (Matrix multiplication). The following 3-dimensional UDA describes the matrix-matrix multiplication $C = A \cdot B$ of two square matrices $A$ and $B$:

- $S_1: a[i, j, k] = A[i, k]$ if $j = 0$
- $S_2: b[i, j, k] = B[k, j]$ if $i = 0$
- $S_3: a[i, j, k] = a[i, j - 1, k]$ if $j > 0$
- $S_4: b[i, j, k] = b[i - 1, j, k]$ if $i > 0$
- $S_5: z[i, j, k] = a[i, j, k] \cdot b[i, j, k]$
- $S_6: c[i, j, k] = c[i, j, k - 1] + z[i, j, k]$ if $k > 0$
- $S_7: c[i, j, k] = z[i, j, k]$ if $k = 0$
- $S_8: C[i, j] = c[i, j, k]$ if $k = N - 1$

Its iteration space is $\mathcal{I} = \{(i, j, k)^T \in \mathbb{Z}^3 \mid 0 \leq i, j, k < N\}$. The input variables are $X_{in} = \{A, B\}$, the output variables $X_{out} = \{C\}$, and the internal variables $X_{var} = \{a, b, z, c\}$. Note that all definitions of internal variables ($S_1$ to $S_7$) have full dimensionality $n = 3$, but the input and output variables are indexed affinely.

The next section introduces Presburger sets, a formalization of the notation we used so far to specify polyhedral iteration, condition, and index spaces. Presburger sets are subsets of $\mathbb{Z}^n$ with certain properties, such as closedness under intersection and union, that further facilitate the analysis and transformation of UDAs.

Remark. The algorithm classes as introduced here do not include data-dependent conditions. However, there are classes that model them, for example, dynamic piecewise linear algorithms as introduced in [Han09]. While the methods in this thesis are presented without explicit mentioning of data-dependent conditions, Section 8.1.1 provides some ideas to seamlessly integrate them into the presented compilation flow.

3.2 Presburger sets

In the polyhedral model, iteration, index, and condition spaces are based on integer polyhedra. These are closely tied to systems of inequalities and thus integer linear programming, enabling the use of powerful algorithms and tools.

Definition 3.6. An $n$-dimensional (convex) integer polyhedron $\mathcal{I} \subseteq \mathbb{Z}^n$ is the set of integer points that solves an $n$-dimensional system of $m$ integral linear inequalities:

$$\mathcal{I} := \{i \in \mathbb{Z}^n \mid Ai \geq b\}, \quad A \in \mathbb{Z}^{m \times n}, b \in \mathbb{Z}^m$$

A more generalized notion is represented by linearly bounded lattices, which can, for example, represent “holes” in the iteration spaces arising from step sizes greater than 1.
3.2 Presburger sets

Definition 3.7 ([Tei93]). An \( n \)-dimensional linearly bounded lattice \( L \) is the set of integer points

\[
L := \{ i \in \mathbb{Z}^n \mid i = Mx + c \land Ax \geq b \},
\]

where \( x \in \mathbb{Z}^\ell \), \( M \in \mathbb{Z}^{n \times \ell} \), \( A \in \mathbb{Z}^{m \times \ell} \), and \( b \in \mathbb{Z}^m \). The set \( \{ x \in \mathbb{Z}^\ell \mid Ax \geq b \} \) is a convex \( \mathbb{Z} \)-polyhedron that is affinely transformed by a lattice \( Mx + c \) to form the linearly bounded lattice\(^5\).

Linearly bounded lattices are not closed under union, difference, and complement, operations we will make frequent use of. Instead, we model iteration, index, and condition spaces as \textit{unions of linearly bounded lattices}; Rajopadhye et al. proved that these are closed under intersection, union, difference, complement, image, and preimage [GR07]. It was also shown that they are equivalent to Presburger sets, which are integer sets induced by Presburger formulae.

Definition 3.8. The \textit{Presburger language} is generated by the following context-free grammar:

\[
\langle \text{term} \rangle \colon= \text{constant} \mid \text{variable} \mid \text{constant} \cdot \text{variable} \mid \langle \text{term} \rangle + \langle \text{term} \rangle \\
\langle \text{relop} \rangle \colon= = \mid \neq \mid \geq \mid \leq \mid < \mid > \\
\langle \text{formula} \rangle \colon= \langle \text{formula} \rangle \land \langle \text{formula} \rangle \mid \langle \text{formula} \rangle \lor \langle \text{formula} \rangle \mid \overline{\langle \text{formula} \rangle} \\
\mid \exists \text{variable}: \langle \text{formula} \rangle \mid \forall \text{variable}: \langle \text{formula} \rangle \mid \langle \text{term} \rangle \langle \text{relop} \rangle \langle \text{term} \rangle
\]

All operators have their usual semantics from first-order logic. Besides the operators, there are two kinds of terminal symbols: a constant is any fixed-value integer, while a variable is a name for an unknown integer.

To model iteration spaces with Presburger formulas, we classify the variables used in a Presburger formula into three distinct sets: the \( n \) iteration variables (which form the iteration vector \( i \in \mathbb{Z}^n \)), parameters, and helper variables (which may only occur bound in a quantification). We then define a Presburger predicate \( P(i) \mapsto \{ \text{true}, \text{false} \} \) to yield the truth value of a Presburger formula containing the iteration variables and any number of parameters and helper variables. A (parameterized) iteration space is then given by \( I := \{ i \mid P(i) \} \). The following examples illustrate this notation and introduce some “syntactic sugar” we use throughout this thesis.

Example. The following 2-dimensional, rectangular iteration space is visualized in Figure 3.1a. The lower and upper bounds are combined using the conjunction \((\land)\):

\[
I_1 = \{ i = (i, j)^T \mid 0 \leq i \land i < N \land 0 \leq j \land j < N \} \\
\equiv \{ i = (i, j)^T \mid 0 \leq i < N \land 0 \leq j < N \} \equiv \{ i = (i, j)^T \mid 0 \leq i, j < N \}
\]

\(^5\)Linearly bounded lattices are a superset of so-called \( \mathbb{Z} \)-polyhedra [Le 95]. A \( \mathbb{Z} \)-polyhedron is the \textit{intersection} of an integer polyhedron with a lattice.
Each element of the set, that is each iteration, is represented by a node drawn at the corresponding coordinate of the 2-dimensional Cartesian coordinate system.

Some algorithms use “triangular” iteration spaces, which can be described by relating multiple iteration variables in a single inequality. For example, the following set is visualized in Figure 3.1b:

\[ I_2 = \{ i = (i, j)^T | 0 \leq i, j < N \land j \leq i \}. \]

By using the disjunction (\( \lor \)), non-convex sets can be described. Suppose, for example, an equation describing outputs at the “right” and “bottom” borders of the iteration space, as visualized in Figure 3.1c:

\[ I_3 = \{ i = (i, j)^T | j = N - 1 \lor i = N - 1 \}. \]

Using linearly bounded lattices without the union operation, such an equation would need to be split into two.

Different step sizes or “holes” in the iteration space can be described by using existential quantification. Suppose, for example, that one equation of a UDA is defined in all iterations with an even \( i \), as visualized in Figure 3.1d:

\[ I_4 = \{ i = (i, j)^T | \exists a: 2a = i \land 0 \leq i, j < N \}. \]

Remark. A mature C implementation of parameterized Presburger sets and operations on them, including parametric integer programming, is provided by isl, the integer set library [Ver10].

### 3.3 Dependence graphs

The dependence relations between indexed variables as defined and used by equations of a recurrence algorithm induce a graph-like structure. Graphs provide a rich mathematical apparatus, making them easy to store, analyze, manipulate, and ultimately an excellent intermediate representation in compilers. In particular, throughout this thesis, we represent UDAs as RDGs (reduced dependence graphs). An RDG is a compact representation of a UDA that makes dependences between the uses and definition of indexed variables more explicit.

**Definition 3.9** (Based on [Rao85; Han09]). The RDG \( G \) of a UDA \( \mathcal{U} \) is a directed multi-graph \( (V, E, I) \) where \( V \) is the set of nodes, \( E \subseteq V \times V \) is the set of edges, and \( I \) is the iteration space of \( \mathcal{U} \). There is a node for each input variable, a node for each output variable, and a node for each equation in \( \mathcal{U} \). There is an edge \( e = (v, w) \) from a node \( v \) to another not necessarily distinct node \( w \) if at least one variable instance defined by \( v \) is used by \( w \), or if \( w \) represents an output variable and \( v \) an equation that defines at least one instance of that output variable.
3.3 Dependence graphs

Figure 3.1: Visualization of Presburger sets induced by the given Presburger formulae.
Edges in the RDG represent the dependences of a UDA explicitly because there is an edge for each definition a use may refer to.

**Example** (Fibonacci). The following 1-dimensional UDA with iteration space \( I = \{(i) \mid 0 \leq i < F \land F \geq 3\} \) specifies the computation of the \( F \)-th number in the Fibonacci sequence:

\[
\begin{align*}
S_1: f[i] &= 1 & \text{if } i \leq 1 \\
S_2: f[i] &= f[i-1] + f[i-2] & \text{if } i \geq 2 \\
S_3: f_{out}[i] &= f[i] & \text{if } i = F - 1
\end{align*}
\]

Consider the use \( f[i-1] \) in equation \( S_2 \). If \( i = 2 \), then instance \( f[1] \) is used, which is defined by \( S_1 \); however, if \( i = 3 \), then instance \( f[2] \) is used, which is defined by \( S_2 \). This means the dependences between equations—and thus operations—are only given implicitly by the UDA. By contrast, this use results in two edges in the RDG, making the two possible sources explicit.

This example raises another question: How can scalar constants, such as the right-hand side of equation \( S_1 \), be modeled? Considering the 1 part of the equation’s function is impractical for PE code generation since the function has to be mapped onto a fixed set of an FU instruction, where constants are usually immediate operands. Therefore, we instead consider constants to be 0-dimensional input variables whose value is known a priori, and whose name is equal to that value. For the Fibonacci example, the set of input variables then is \( X_{in} = \{1\} \). For convenience, we call the subset of constants \( X_{constant} \).

In the following, we structure relevant information using node and edge annotations.

**Definition 3.10.** A node annotation \( a \) maps a node \( v \in V \) to a value \( z \) of arbitrary type. To disambiguate annotations from function application, we write \( a[v] \) to access the annotated value \( z \) and \( a[v] \leftarrow z \) to annotate a new value \( z \). An edge annotation is defined analogously.

Each node in an RDG is annotated with the indexed variable \( x \in X \) and domain \( I \), which is the set of indices of \( x \) the node represents. Each edge \( e \) is annotated with the corresponding indexing function \( \phi := (Q, d) \). For convenience, we classify and further annotate the nodes and edges of the RDG as follows:

- A node that represents an equation \( S \) is called an operation node and is additionally annotated with an operation \( op \) that realizes the function \( f \) of \( S \). Operation nodes must have at least one outgoing edge and at least one incoming edge for each argument\(^6\) of \( f \). Each incoming edge is annotated with the argument position \( pos \) within \( f \).

\(^6\)Some functions may have no arguments and the corresponding operation node therefore no incoming edges.
3.3 Dependence graphs

- A node that represents an non-constant input variable is called an input node. Inputs nodes have no incoming edges, but must have at least one outgoing edge. Similarly, a node that represents an output variable is called an output node. Output nodes have no outgoing edges, but must have at least one incoming edge.

- A node that represents a constant is called a constant node and is additionally annotated with its value $c$.

- An edge from an operation node $v$ to another not necessarily distinct operation node $w$ is called a dependence edge and represents the use of an instance defined by the UDA.

- An edge from an input node $v$ to an operation node $w$ is called an input edge and represents the use of an instance not defined by the UDA.

- An edge from an operation node $v$ to an output node $w$ is called an output edge and represents the definition of an instance not used by the UDA.

- An edge from a constant node to an operation node is called a constant edge and represents the use of a constant value in an operation.

Remark. Some works, for example [Han09], introduce more types of nodes and edges, particularly for use in run-time conditionals. In this thesis, however, we limit RDGs to the necessary types for the presented compilation techniques.

The construction of an RDG from a given UDA is detailed in Appendix D.

Example (Fibonacci, continuing from p. 44). Figure 3.2 shows the RDG of the Fibonacci UDA given in the previous example and introduces the conventions used in this thesis for diagrams of RDGs.

Example (FIR filter, continuing from p. 38). Figure 3.3 shows the RDG of the FIR filter example. In contrast to the Fibonacci example, there are two input nodes, one for $U$ and one for $A$.

Example (Matrix multiplication, continuing from p. 40). Figure 3.4 shows the RDG of the matrix multiplication example.

While we use RDGs as the formal representation of UDAs, we visualize the spatial and temporal aspects using an unfolded dependence graph (or just called dependence graph) to improve the reader’s comprehension in the following sections and chapters about symbolic space-time mapping. In the unfolded dependence graph of a UDA, there is a node $i$ for each iteration $i \in I$ and an edge $(i, i')$ between two nodes if there is a dependence vector $d$ in the set of dependence vectors $D$ of the UDA with the property $d = i' - i$. Figure 3.5 shows the unfolded dependence graph for the FIR example. It is conventional to display coordinate axes to emphasize the geometric nature of unfolded dependence graphs (that stems from the polyhedral model).
3 Fundamentals of Loop Parallelization and Scheduling in the Polyhedral Model

Figure 3.2: RDG for the Fibonacci example UDA. Input and output nodes are drawn as rounded rectangles, operation nodes as rectangles, and constant nodes as circles. Each node is labeled with its name (here \(v_0\) to \(v_4\)), followed by a colon and the name of the corresponding indexed variable or constant and the corresponding equation in brackets (e.g., \([S_1]\)) if applicable. Operation nodes are additionally labeled with their operation using the notation "\(op(#0, #1, \ldots)\)" to show their arity. Finally, input, output, and operation nodes are labeled with their domain \(I\). Solid edges are either input, constant, or dependence edges and have their argument position annotated as \#\(pos\); dashed edges are output edges. Except for constant edges, all edges have the corresponding indexing function \(\phi\) annotated, which means the indexing matrix \(Q\) and indexing vector \(d\) for input and output edges, and only the indexing vector \(d\) for dependence edges. Note that we omit displaying \(d\) if \(d = 0\).

3.4 Space-time mapping

A UDA only describes what to compute, but not how. In particular, given an RDG, each operation node \(v\) represents an operation instance \(v[i]\) for each \(i \in v[I]\). But on which PE and FU and when is each instance computed? The solution entails solving two problems, namely allocation—assigning the type and quantity of resources to the computation of the operation \(op[v]\)—and scheduling—assigning a start time of execution to each operation instance \(v[i]\). These two parts form the so-called space-time mapping of a UDA onto a processor array such as a TCPA. Since allocation and scheduling heavily influence non-functional properties and thus the quality of the mapping, they have been the subject of numerous publications; for an overview, see [Han09]. But among these large number of approaches, which ones are amenable to symbolic compilation onto TCPAs?

Remark. Due to the introductory nature of this chapter, it only introduces the fundamentals of space-time mapping in the polyhedral model and how they apply to TCPAs. Generating configuration data for the periphery, as well as generating and instantiating PE programs are novel contributions of this dissertation and the subject of Chapters 4 to 6.

46
3.4 Space-time mapping

Figure 3.3: Reduced dependence graph for the FIR filter example.
3 Fundamentals of Loop Parallelization and Scheduling in the Polyhedral Model

$$Q = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad d = 0$$

**Figure 3.4:** Reduced dependence graph for the matrix multiplication example.

**Figure 3.5:** Dependence graph of the Fibonacci example.
### 3.4 Space-time mapping

#### 3.4.1 Choice of allocation and scheduling techniques

Given an RDG, the allocation for processor arrays such as TCPAs consists of two parts: *Global allocation* distributes the iterations among PEs by allocating a subset $\tilde{P} \subseteq P$ of PEs and assigning each iteration $i \in I$ a PE $p(i) \in \tilde{P}$ to execute on. *Local allocation* assigns each operation node $v$ a functional unit $v[lu] \in FU$ and each (non-constant) edge $e$ a set of registers $e[\tilde{R}] \subset R$ to store or communicate data. An edge that induces inter-processor communication is also assigned an interconnect route $v[\rho]$. Finally, scheduling assigns each operation node instance $v[i]$ with $i \in v[I]$ a start time $t(v, i) \in \mathbb{Z}$ such that no data dependences are violated.

Global allocation techniques are motivated by the desire to mirror the regularity of a loop in the mapping to the target processor array. The first of two common techniques resulting in regular allocations, *projection*, maps all iterations along a given projection vector to the same PE [Kuh80]. However, similarly to how the size of a shadow depends on the size of the object blocking the light, the number of PEs obtained by projection depends on the size of the iteration space. This fact alone invalidates projection for use in our desired use-case: A run-time enforcer must be able to choose the number of PEs to adjust non-functional execution properties (see Chapter 7).

We therefore favor the other common technique, called *partitioning* or *tiling*, which covers the iteration space with no overlaps or gaps [IT88] using a single tile shape, such as an $n$-dimensional hyperrectangle or parallelogram. In contrast to projection, the number of PEs when tiling is independent of the iteration space size, instead depending on tiling parameter. We introduce *symbolic tiling* using hyperrectangles in more detail in Section 3.4.2. Of particular interest are the following two kinds of global allocation using tiling [Jai86]:

**Locally sequential, globally parallel (LSGP)** The number of allocated PEs is equal to the number of tiles. Each PE executes the iterations within a tile sequentially; the tiles are executed in parallel. This is also known as outer loop parallelization.

**Locally parallel, globally sequential (LGPS)** The number of allocated PEs is equal to the number of iterations within a tile. Iterations within a tile are executed in parallel by the PEs; the tiles are executed sequentially. This is also known as inner loop parallelization.

While LSGP has higher local memory requirements but lower inter-processor communication bandwidth requirements, LGPS has lower local memory requirements but higher inter-processor communication bandwidth requirements. While the compilation techniques presented here can incorporate both schemes [TWT+14], the remainder

---

7 This is especially important for symbolic mapping because it is far easier to represent a regular structure symbolically than an irregular one.

8 Mathematically, a tiling is a *monohedral tessellation*.

9 Combining both approaches into a hierarchical partitioning scheme called co-partitioning allows trade-offs between these requirements [EM97].
of this thesis focuses on LSGP, which is arguably more natural for TCPAs: Much of the communication induced by LSGP is processor-internal using local memories, mapping well to sequential assembly programs accessing registers.

Local allocation techniques for loops, on the other hand, are motivated by maximizing the instruction-level parallelism constrained by a fixed number of PE-local resources such as functional units and registers. Because of its finer grain and higher irregularity compared to global allocation (iteration-based vs. operation-based), local allocation is significantly more intertwined with scheduling, which is why they are usually combined. The state-of-the-art technique combining both is resource-constrained modulo scheduling [RG81]. Modulo scheduling is a realization of software pipelining [Cha81], which overlaps the execution of consecutive iterations to further increase instruction-level parallelism. We introduce the principle behind modulo scheduling in Section 3.4.4. However, existing modulo scheduling algorithms are insufficient for our goals—in particular, they cannot be performed symbolically on \( n \)-dimensional loops—which is why we present symbolic modulo scheduling in Chapter 4.

Having motivated our choices for allocation and scheduling, the following sections introduce the respective techniques in more detail, starting with symbolic tiling.

### 3.4.2 Symbolic tiling

Tiling partitions an iteration space into uniform tiles of a given size and shape [Xue00]. While there are many permissible shapes, the remainder of this thesis assumes rectangular tiling: The given iteration space is partitioned into \( n \)-dimensional, rectangular tiles, each of size \( t_0 \times t_1 \times \ldots t_{n-1} \), where the tile sizes \( p_i \) are called the tiling parameters. Compared to other shapes that tessellate the \( n \)-dimensional space, hyperrectangles minimize the correlation between the individual dimensions because their sides are parallel to the standard basis, an advantage that facilitates symbolic representation and analysis.

In this thesis, the tiling parameters are kept as symbols in the space-time mapping, in particular the schedule, until instantiation at runtime and hence we specify tiling to be symbolic for emphasis; mathematically, there is no difference between “tiling” and “symbolic tiling”. The idea to find symbolic schedules from tiling was first proposed for single-level rectangular tiling in [TTH13] and extended to multiple levels in [TWT+14]. The remainder of this section summarizes single-level symbolic tiling because it provides the foundation for symbolic modulo scheduling as presented in Chapter 4.

We assume that any dependence vector of a given UDA must be able to fit into a tile\(^{10}\):

\[
p_i \geq \max_{d=(d_0,\ldots,d_{n-1}) \in D} (d_i), \quad 0 \leq i < n,
\]

a constraint called short dependences [Xue00].

\(^{10}\)As explained later, this restricts inter-processor communication to direct neighbors by construction.
3.4 Space-time mapping

Mathematically, tiling is a decomposition of an iteration space $I \subset \mathbb{Z}^n$ into an intra-tile space $J \subset \mathbb{Z}^n$ and a tile space $K \subset \mathbb{Z}^n$ such that $I$ is a subset of their composition:

$$I \subset J \oplus K := \{ i \mid i = P \cdot k + j, k \in K, j \in J \},$$

where $P \in \mathbb{Z}^{n \times n}$ is called the tiling matrix and describes the shape of the tiles. From $J$ and $K$, we construct the corresponding tiled iteration space

$$I^* = \{ i^* = (j^T, k^T)^T \mid j \in J \land k \in K \} \subset \mathbb{Z}^{2n},$$

where $(j^T, k^T)^T$ denotes the concatenation of $j$ and $k$, making the tiled iteration space 2$n$-dimensional. For rectangular tiles, $P$ is the diagonal matrix $\text{diag}(p_0, p_1, \ldots, p_{n-1})$ and accordingly

$$J = \{ j = (j_0, j_1, \ldots, j_{n-1})^T \mid 0 \leq i < n : 0 \leq j_i < p_i \}.$$

We also assume a rectangular tile space because the PE of a TCPA are allocated in rectangular regions. Assuming the rectangular hull of the untiled iteration space $\overline{I}$ is given by the Cartesian product $[l_0, u_0] \times \cdots \times [l_{n-1}, u_{n-1}]$, the tile space is

$$K = \left\{ k = (k_0, k_1, \ldots, k_{n-1})^T \mid \left| \frac{l_1}{p_1} \right| \leq k_i < \left| \frac{u_1}{p_1} \right| \right\},$$

where $\left| \frac{u_1}{p_1} \right| - \left| \frac{l_1}{p_1} \right| = t_i$ denotes the number of tiles in dimension $i$.

While obtaining the original iteration $i$ from a tiled iteration $i^*$ is an affine transformation (as given by Equation (3.3)), obtaining the tiled iteration from the untiled iteration is not:

$$\forall 0 \leq \ell < n : i_\ell = p_\ell \cdot k_\ell + j_\ell \implies k_\ell = \left| \frac{i_\ell}{p_\ell} \right|, j_\ell = i_\ell \mod p_\ell$$

This discrepancy causes tiling to be not closed in the polyhedral model, undermining its advantages in some situations.\footnote{Some approaches, such as mono-parametric tiling [IRA’15], aim to make tiling closed in the polyhedral model, but put restrictions on it that are unreasonable for our purposes. Mono-parametric tiling in particular requires each tile size parameter $p_\ell$ to be a multiple of a single parameter $b$ (hence the name mono-parametric).} We will mention cases where this non-affineness is problematic and address possible solutions (mostly assuming tiling only implicitly and work on the untiled iteration space).

**Example** (FIR filter, continuing from p. 45). The iteration space of the FIR filter is

$$I = \{(i, j)^T \mid 0 \leq i < T \land 0 \leq j < N \}.$$

Tiling it symbolically yields the intra-tile space

$$J = \{ j = (j_0, j_1)^T \mid 0 \leq j_0 < p_0 \land 0 \leq j_1 < p_1 \}.$$
Figure 3.6: Tiled iteration space $I^*$ of the FIR filter example. Iterations marked in gray show the embedded domain of $v_2$. Bold arrows mark representatives of the tiled dependences of $(1, 1)^T$.

and the tile space

$$
K = \{ k = (k_0, k_1)^T \mid 0 \leq k_0 < t_0 = \left\lceil \frac{T}{p_0} \right\rceil \land 0 \leq k_1 < t_1 = \left\lceil \frac{N}{p_1} \right\rceil \}.
$$

An iteration vector of the tiled iteration space $I^*$ is 4-dimensional: $i^* = (j_0, j_1, k_0, k_1)^T$. The tiled iteration space is visualized in Figure 3.6.

Strictly speaking, tiling only transforms a UDA's iteration space. However, "tiling a UDA" implies for all other parts of the UDA—that is, indexed variables, condition spaces, and indexing functions—to be embedded accordingly to preserve syntactic correctness and semantic equivalence to the untiled UDA.
First, the condition space $I_i$ of each equation $S_i$ is embedded into the $2n$-dimensional tiled iteration space:

$$I^*_i = \{ i^* = (j^T, k^T)^T \mid Pk + j \in I_i \}$$

**Example** (FIR filter, continuing from p. 51). Given the condition space of $S_5$ (corresponding to the domain of $v_2$ in Figure 3.3),

$$I_5 = \{ i = (i, j)^T \mid i > 0 \land j > 0 \}$$

embedding it into the tiled iteration space $I^*_5$ yields

$$I^*_5 \leftarrow \{ (j_0, j_1, k_0, k_1)^T \mid p_0 \cdot k_0 + j_0 > 0 \land p_1 \cdot k_1 + j_1 > 0 \}. $$

The embedded condition space is visualized using gray iteration nodes in Figure 3.6.

Then, the indexing functions of all input variables and output variables of the UDA are embedded as follows:

$$i := Pk + j \quad \Rightarrow \quad \phi^*_{i,j}(i^*) := Q_{i,j}(Pk + j) - d_{i,j} \quad \text{(input variables)}$$

$$\quad \Rightarrow \quad \psi^*_i(i^*) := Q^*_i(Pk + j) - d'_i \quad \text{(output variables)}$$

Next, all internal variables are lifted to be $2n$-dimensional, requiring their indexing also to become $2n$-dimensional. This entails embedding the uniform dependence vectors $d \in D$ into the tiled iteration space. Each original dependence vector $d$ results in a set $D^*(d)$ of tiled dependence vectors, which, for orthogonal tiling, is given by

$$D^*(d) = (d_0, \ldots, d_{n-1})^T = \left\{ d^* = (d^i, d^k)^T = \left( d^i_0, \ldots, d^i_{n-1}, d^k_0, \ldots, d^k_{n-1} \right) \right\},$$

where

$$(d^i_\ell, d^k_\ell) \in \left\{ \left[ i_\ell - p_\ell, \frac{i_\ell + d_\ell}{p_\ell} \right], \left[ i_\ell + d_\ell, \frac{i_\ell + d_\ell}{p_\ell} \right] \mid 0 \leq i_\ell < p_\ell \right\}. $$

For short dependences, this further simplifies to

$$\left( d^i_\ell, d^k_\ell \right) \in \{ (d_\ell, 0), (d_\ell - \text{sgn}(d_\ell) \cdot p_\ell, \text{sgn}(d_\ell)) \}. \quad (3.4)$$

We call $d^i$ the *intra-tile dependence vector* and $d^k$ the *inter-tile dependence vector*. The cardinality of $D^*(d)$ depends on the number $z$ of non-zero elements in the dependence vector $d$: For a zero element in dimension $i$, the vector never crosses into another tile in that dimension. For a non-zero element in dimension $i$, the vector either does or does not cross into another tile depending on $j$. If it does, then $k_j \neq 0$, if it does not, then $k_i = 0$. Consequently, for each non-zero dimension $i$, there are two possibilities for $k_i$, resulting in $|D^*(d)| = 2^z$. (The set in Equation (3.4) accordingly has a cardinality of 1 if $d_\ell = 0$ and 2 otherwise.)
Example (Fibonacci, continuing from p. 44). Embedding the dependence vector \( \mathbf{d} = (2) \) in the Fibonacci UDA into its tiled iteration space \( \mathcal{I}^* \) yields
\[
D^*((2)) = \left\{ \begin{pmatrix} 2 \\ 0 \\ 2 - p_0 \\ 1 \end{pmatrix} \right\}.
\]

Example (FIR filter, continuing from p. 53). The given UDA \( \mathbf{U}_{FIR} \) has the following set of dependence vectors:
\[
D_{FIR} = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \right\}
\]

Tiling each of these vectors yields
\[
D^*((0)) = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \right\},
\]
\[
D^*((1)) = \left\{ \begin{pmatrix} 1 \\ 0 \\ 1 - p_0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \right\}, \quad D^*((0)) = \left\{ \begin{pmatrix} 0 \\ 1 \\ 1 - p_0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \right\},
\]
\[
D^*((1)) = \left\{ \begin{pmatrix} 1 \\ 1 \\ 1 - p_0 \\ 1 \\ 1 \\ 1 \end{pmatrix} \right\}.
\]

Figure 3.6 illustrates a representative for each vector in \( D^*((1, 1)^T) \) by a bold arrow. \( \triangle \)

Remark. Because of symbolic tiling, the tile parameters \( p_i \) typically remain unknown during compilation, meaning the absolute number of time steps between production and consumption of an intermediate result that are available to not violate tiled data dependences is unknown. We address this problem in Chapter 4.

As previously implied, which \( \mathbf{d}^* \in D^*(\mathbf{d}) \) represents \( \mathbf{d} \) in the tiled iteration space depends on the intra-tile iteration \( j \). The corresponding original equation of the UDA to be tiled must therefore be split into \( |D^*(\mathbf{d})| \) new equations whose condition spaces form a partition of the original condition space, as demonstrated by the following example.\(^{12}\)

\(^{12}\)The algorithm of how to split equations according to tiled dependences is not essential for this thesis and was therefore omitted.
Example (FIR filter, continuing from p. 54). The following UDA represents a tiled version of the UDA given for the FIR filter.

\[
S_1: \quad a[j_0, j_1, k_0, k_1] = A[p_1 \cdot k_1 + j_1] \quad \text{if} \quad k_0 = 0 \land j_0 = 0 \\
S_{2a}: \quad a[j_0, j_1, k_0, k_1] = a[j_0 - 1, j_1, k_0, k_1] \quad \text{if} \quad j_0 > 0 \\
S_{2b}: \quad a[j_0, j_1, k_0, k_1] = a[j_0 + 1 - p_0, j_1, k_0 - 1, k_1] \quad \text{if} \quad k_0 > 0 \land j_0 = 0 \\
S_3: \quad u[j_0, j_1, k_0, k_1] = U[p_0 \cdot k_0 + j_0] \quad \text{if} \quad k_1 = 0 \land j_1 = 0 \\
S_4: \quad u[j_0, j_1, k_0, k_1] = 0 \quad \text{if} \quad ((k_1 = 0 \land j_1 > 0) \lor (k_1 > 0)) \land k_0 = 0 \land j_0 = 0 \\
S_{5a}: \quad u[j_0, j_1, k_0, k_1] = u[j_0 - 1, j_1 - 1, k_0, k_1] \quad \text{if} \quad j_0 > 0 \land j_1 > 0 \\
S_{5b}: \quad u[j_0, j_1, k_0, k_1] = u[j_0 + 1 - p_0, j_1 - 1, k_0 - 1, k_2] \quad \text{if} \quad j_0 = 0 \land k_0 > 0 \land j_1 > 0 \\
S_{5c}: \quad u[j_0, j_1, k_0, k_1] = u[j_0 - 1, j_1 + 1 - p_1, k_0, k_2 - 1] \quad \text{if} \quad j_0 > 0 \land j_1 = 0 \land k_1 > 0 \\
S_{5d}: \quad u[j_0, j_1, k_0, k_1] = u[j_0 + 1 - p_0, j_1 + 1 - p_1, k_0 - 1, k_2 - 1] \quad \text{if} \quad j_0 = 0 \land k_0 > 0 \land j_1 = 0 \land k_1 > 0 \\
S_6: \quad x[j_0, j_1, k_0, k_1] = a[j_0, j_1, k_0, k_1] \cdot u[j_0, j_1, k_0, k_1] \\
S_7: \quad y[j_0, j_1, k_0, k_1] = x[j_0, j_1, k_0, k_1] \quad \text{if} \quad j = 0 \\
S_{8a}: \quad y[j_0, j_1, k_0, k_1] = y[j_0, j_1 - 1, k_0, k_1] + x[j_0, j_1, k_0, k_1] \quad \text{if} \quad j_1 > 0 \\
S_{8b}: \quad y[j_0, j_1, k_0, k_1] = y[j_0, j_1 + 1 - p_1, k_0, k_1 - 1] + x[j_0, j_1, k_0, k_1] \quad \text{if} \quad j_1 = 0 \land k_1 > 0 \\
S_9: \quad Y[p_0 \cdot k_0 + j_0] = y[j_0, j_1, k_0, k_1] \quad \text{if} \quad p_1 \cdot k_1 + j_1 = N - 1
\]

The iteration space, all condition spaces, and all internal variables are now \((2n = 4)\)-dimensional. Equations that have been split according to tiled dependences have been labeled with letters (for example, \(S_3\) has been split into \(S_{5a}\) to \(S_{5d}\)).

\[\triangle\]

Clearly, the RDG of a tiled UDA has twice the dimensions, possibly exponentially more nodes, and possibly exponentially more edges than the untiled version despite their semantic equivalency. This bloating obfuscates the original UDA and introduces unnecessary additional workload during compilation. Furthermore, the non-affineness of tiling undermines the advantages of the polyhedral model. Instead, the following sections and chapters assume tiling only implicitly and use the explicitly tiled and embedded entities only where necessary. In the latter case, we abbreviate "RDG of a tiled UDA" to simply "tiled RDG". Sometimes, we may also distinguish between "tiled" and "original" entities.

Remark. Besides one-level tiling as described in this section, there are also multi-level approaches that aim to balance I/O bandwidth and local memory requirements. We developed one such approach, called symbolic hierarchical tiling [TWT+15].
3.4.3 Processing element allocation

During global allocation, a so-called **space mapping** assigns each tile \( k \) in the tile space \( \mathcal{K} \) a processing element \( p = (r, c) \) in the processor space \( \mathcal{P} \) to execute exactly the operations inside that tile.

**Definition 3.11.** A two-dimensional **space mapping** is an affine function

\[
\phi(k) : \mathcal{K} \mapsto \mathcal{P} := p = \Phi \cdot k - \tilde{p}, \quad \Phi \in \mathbb{Z}^{2 \times n}, \tilde{p} \in \mathbb{Z}^2
\]  

(3.5)

where \( \Phi \) is called the allocation matrix and \( \tilde{p} \) the allocation base.

Given the tile space \( \mathcal{K} \) of a tiled UDA and a space mapping \( \phi \), its image

\[
\phi(\mathcal{K}) := \{ \phi(k) | k \in \mathcal{K} \} \subseteq \mathcal{P}
\]

represents the set of allocated PEs. However, during the instantiation (see Chapter 6) of a symbolically mapped UDA, we are interested in the inverse problem: Given the space mapping \( \phi \) with which the algorithm was mapped and the desired rectangular region \( \widehat{\mathcal{P}} \) of PEs to execute on, find concrete tiling parameters that concretize \( \mathcal{K} \) such that \( \phi(\mathcal{K}) \cong \widehat{\mathcal{P}} \). The size \( \widehat{R} \times \widehat{C} \) of \( \widehat{\mathcal{P}} \) is chosen, for example, according to resource constraints—other applications might be occupying parts of the TCPA already—or according to non-functional properties (see Chapter 7)—for instance, fewer or more PEs to achieve a certain latency requirement while minimizing energy consumption.

Because the allocation matrix \( \Phi \) is not invertible in general, the space mapping has no unique solution when solving for \( k \) given \( p \). However, we are interested in a one-to-one mapping of tiles to PEs, which has the advantage that tiling does not need to be considered within a PE, simplifying code generation (see Chapter 6). We therefore reasonably restrict \( \Phi \) as follows: Each row vector of \( \Phi = (\phi_0, \phi_1)^T \) is equal to a standard basis vector \( e_i = (0, \ldots, 1, \ldots, 0) \in \mathbb{Z}^n \) whose elements are all zero, except for the \( i \)-th element, which is 1. In addition, \( \phi_0 \neq \phi_1 \). Equation (3.5) then simplifies to

\[
\begin{align*}
    r &= \phi_0 \cdot k - \tilde{p}_0 = e_i \cdot k - \tilde{p}_0 \implies r = k_i - \tilde{p}_0, \\
    c &= \phi_1 \cdot k - \tilde{p}_1 = e_i \cdot k - \tilde{p}_1 \implies c = k_i - \tilde{p}_1.
\end{align*}
\]

These restrictions effectively make \( \Phi \) an orthogonal projection.\(^{13}\)

**Example.** Assume \( \mathcal{K} \) is a 3-dimensional tile space. Let

\[
\Phi_1 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \Phi_2 = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}, \quad \tilde{p} = 0,
\]

then \( \phi(k) = (k_1, k_2) \) for \( \Phi_1 \) and \( \phi(k) = (k_2, k_0) \) for \( \Phi_2 \).

\(^{13}\)Note that this is not related to global allocation by projection. Furthermore, if \( n > 2 \), then \( \Phi \) is not a projection matrix in the strict sense for which \( \Phi^2 = \Phi \) would need to hold. However, this is easily remedied by extending \( \Phi \) by \( n - 2 \) rows of zeroes. This would mean that \( n - 2 \) dimensions of the result are always zero, which is why we omit them for clarity.
From \( r = k_1 \) follows that the tile count \( t_i = \frac{1}{k_1} R \), from \( c = k_1 \) that the tile count \( t_i = \frac{1}{k_1} C \). Setting all other tile counts to 1 results in the desired one-to-one mapping of tiles to PEs. To have \( t_i = \frac{1}{k_1} R \), the tile size \( p_i \) must be chosen appropriately. Let the Cartesian product \([l_0, u_0] \times \cdots \times [l_{n-1}, u_{n-1}]\) describe the rectangular hull of the untiled iteration space. From \( p_i = \lceil (u_i - l_i)/t_i \rceil \), we conclude

\[
\begin{cases}
\frac{1}{k_1} R & \text{if } \phi_0 = e_i \\
\frac{1}{k_1} C & \text{if } \phi_1 = e_i \\
1 & \text{else}
\end{cases}
\]

We also set \( \tilde{p} = (-\lfloor l_r/p_r \rfloor, -\lfloor l_c/p_c \rfloor) \) to make \( p \) zero-based.

**Remark.** Allocating a 1-dimensional region of PEs is equivalent to setting either \( \tilde{R} \) or \( \tilde{C} \) to 1.

**Example.** Assume \( \tilde{R} = 2 \) and \( \tilde{C} = 4 \). Given \( \Phi_1 \) and \( \Phi_2 \) from the previous example and a rectangular 3-dimensional iteration space

\[
I = \{(i_0, i_1, i_2) \mid 0 \leq i_0 < 10 \wedge 0 \leq i_1 < 16 \wedge 0 \leq i_2 < 24\},
\]

then

\[
\Phi_1: P_1 = \text{diag}(10 - 0, \frac{16 - 0}{2}, \frac{24 - 0}{4}) = \text{diag}(10, 8, 6),
\]

\[
\Phi_2: P_2 = \text{diag}(\frac{10 - 0}{4}, 16 - 0, \frac{24 - 0}{2}) = \text{diag}(3, 16, 12).
\]

Both tiling matrices result in the requested \( 2 \times 4 \) PEs.

**Processor displacements**

The distribution of iterations across multiple PEs may entail inter-processor communication for loop-carried dependences \( d \neq 0 \), for example \( x[i] = y[i - 1] \). We assume that communication always takes place between neighboring PEs, an assumption that is ensured by the short dependence constraint given in Equation (3.2). Which PEs do communicate for such a dependence \( d \neq 0 \)? Suppose tiling maps an iteration \( i \) to tile \( k \). Then the source iteration \( i - d \) is on tiles \( \{k - \theta \mid \theta \in \Theta(d)\} \) where \( \Theta(d) \) is the set of tile displacements

\[
\Theta(d) := \left\{ \theta = d^k \mid d^* = \left((d^i)^T, (d^k)^T\right)^T \in D^*(d) \right\}.
\]
Note that this includes the 0-displacement $\theta = d^k = 0$, which indicates an intra-tile dependence.

The set of processor displacements—which processors $\{\phi(k) - \delta\}$ need to output the value—is

$$\Delta(d) := \{\delta \mid \delta = \Phi\theta \land \theta \in \Theta(d)\}.$$  

Again, the special case $\delta = 0$ indicates that only intra-processor communication is necessary. For $\delta \neq 0$, inter-processor communication is necessary.

**Example.** Given a 3-dimensional dependence vector $d = (1, 1, 0)^T$, the set of tile displacements is

$$\Theta(d) = \left\{ \begin{array}{c} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \end{array} \right\}.$$  

Given the allocation matrices $\Phi_1$ and $\Phi_2$ from the previous examples, the corresponding sets of processor displacements are

$$\Phi_1: \Delta_1(d) = \left\{ \begin{array}{c} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \end{array} \right\},$$

$$\Phi_2: \Delta_2(d) = \left\{ \begin{array}{c} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \end{array} \right\}.$$  

Both cases induce inter-processor communication: when using $\Phi_1$, processor $(k_0, k_1 - 1)^T$ has to communicate to processor $(k_0, k_1)^T$; when using $\Phi_2$, processor $(k_0 - 1, k_1)^T$ has to communicate to processor $(k_0, k_1)^T$. Clearly, the space mapping influences the necessary communication channels.

It depends on the current iteration $i^*$ whether the value along the original dependence $d$ is stored internally or communicated to one of the processors within the set of processor displacements, and whether the value is read from internal storage (that is, registers) or from another processor.

**Definition 3.12.** Assume $d \in D$ with the set of processor displacements $\Delta(d)$. Each $\delta \in \Delta(d)$ is associated with an input domain $D_{d,\delta}^{\text{in}}$ that contains the tiled iterations in which the the corresponding data value is received from processor $k - \delta$. Analogously, each $\delta \in \Delta(d)$ is associated with an output domain $D_{d,\delta}^{\text{out}}$ that contains the tiled iterations in which the corresponding data value is sent to processor $k + \delta$.

Let $\phi$ be the space mapping with which $\Delta$ was determined. Then, the input domain of processor displacement $\delta$ of dependence vector $d$ is

$$D_{d,\delta}^{\text{in}} := \{ (j, k)^T \mid \phi(k - k') = \delta \land i \in I \land i^* := (j, k)^T \land (i - d)^* := (j', k')^T \}.$$  

Analogously, the output domain is

$$D_{d,\delta}^{\text{out}} := \{ (j, k)^T \mid \phi(k - k') = \delta \land i \in I \land i^* := (j, k)^T \land (i + d)^* := (j', k')^T \}.$$
3.4 Space-time mapping

For a dependence $d$, we denote the set of all input domains $D_{d}^{\text{in}}$, the set of all output domains $D_{d}^{\text{out}}$.

**Example.** Given a dependence vector $d = (1, 1)^{T}$, assume the set of processor displacements is

$$\Delta(d) = \left\{ \delta_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \delta_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \delta_3 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \delta_4 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}$$

The corresponding input domains, given the identity matrix as allocation matrix, are visualized in Figure 3.7.

### 3.4.4 Modulo scheduling

After global allocation, the local allocation of functional units, registers, and interconnect routes, as well as scheduling of operations remain to complete the space-time mapping. Scheduling an RDG is the problem of assigning each operation node instance $v[i]$ a start time $t(v, i)$ such that no data dependences are violated. This assignment is called a schedule. The typical goal of scheduling is to minimize the latency

$$L = \max_{v \in V, i \in I} (t(v, i) + l[v]) - \min_{v \in V, i \in I} t(v, i),$$

(3.6)

where $l[v]$ is the latency of the node’s operation.

The theoretically latency-minimal schedule is called free schedule and assigns each operation node instance the earliest feasible time [DKR92]:

$$\forall i \in I : t(v, i) = \begin{cases} 0 & \text{if } \exists i' \in I : i > i' \\ \max_{i' \in I, i > i'} t(v, i') + 1 & \text{else} \end{cases},$$

where $i > i'$ denotes that iteration $i$ depends on iteration $i'$—in other words, some operation instance in iteration $i$ directly or indirectly uses at least one result computed in iteration $i'$. While optimal and useful for theoretical studies, free schedules are ill-suited for practical use in loop accelerators because they are in general unstructured and thus difficult to analytically derive and represent. Instead, we use affine schedules [Fen92].

**Definition 3.13.** Given an $n$-dimensional RDG, an affine schedule is defined by an $n$-dimensional schedule vector $\lambda \in \mathbb{Z}^{n}$ and a start time offset $\tau_v \in \mathbb{N}_0$ for each $v \in V$:

$$\forall v \in V, i \in I : t(v, i) = \lambda i + \tau_v$$

For notational convenience, we denote an affine schedule as a tuple $(\lambda, \tau)$, where $\tau$ is a vector containing all start time offsets $\tau_v$.

\[\text{Note that there is work regarding determination of free schedules, such as by Bielecki [BPK12], who also gives an overview of other approaches.}\]
Figure 3.7: For the dependence vector \( d = (1, 1)^T \) in a two-dimensional UDA using the identity matrix as allocation matrix, this figure shows the input domains of the four processor displacements \( \delta_1 = (0, 0)^T, \delta_2 = (1, 0)^T, \delta_3 = (0, 1)^T, \) and \( \delta_4 = (1, 1)^T \) by the numbers in the iteration nodes. Additionally, the intra-tile iterations are grouped according to the input domains using shaded rectangles.
3.4 Space-time mapping

Affine schedules are close to optimality for UDAs [DKR92], are very tractable by virtue of their roots lying in linear algebra, and are cheap to implement in hardware.

When scheduling the RDG of a tiled UDA, the affine schedule applies to the tiled iteration space:

$$t(v, i^*) = \lambda^* i^* + \tau_v$$

For LSGP in particular, the schedule vector $\lambda^* = (\lambda^i, \lambda^k)$ consists of an inter-tile schedule vector $\lambda^k$ that describes a parallelized execution order of the tiles $k \in K$ and an intra-tile schedule vector $\lambda^i$ that describes a sequential execution order of the iterations $j \in J$ within every tile; see Figure 3.8 for a concrete example.

Although the intra-tile schedule imposes a sequential execution order of the iterations, the operations within the iterations still ought to be parallelized. For loops, an effective technique is software pipelining [Cha81], which overlaps the execution of operations from consecutive iterations. This not only increases the degree of available instruction-level parallelism, but consequently decreases the time between starting consecutive iterations, significantly improving the overall latency. The state-of-the-art
technique for software-pipelining loops is *modulo scheduling* [RG81]. Modulo scheduling produces a cyclic schedule with a period of $\pi$, called the *initiation interval*, that results in software pipelining as depicted in Figure 3.9.

By incorporating *resource constraints* for local allocation, modulo scheduling becomes NP-hard. While the literature presents many solutions to the one-dimensional so-called *resource-constrained modulo scheduling* problem (see Section 4.11), none considers directly modulo-scheduling parametric multi-dimensional iteration spaces despite most loops being nested with parametric bounds. Furthermore, none consider loop parallelization during modulo scheduling. The next chapter presents *symbolic modulo scheduling*, as the first solution to consider these important aspects.
Symbolic Modulo Scheduling

Scheduling the operations in a RDG (reduced dependence graph) for a TCPA such that PE programs can be generated (see Chapter 5) requires solving the so-called RCMSP (resource-constrained modulo scheduling problem). The RCMSP involves minimizing the latency under two kinds of constraints: feasibility constraints, derived from the RDG’s dependence edges (see Section 3.1), and resource constraints, derived from the architectural description (see Section 2.1). Solving the RCMSP symbolically as-is is difficult because both the feasibility constraints and the latency function contain products of the unknown tile size parameters. This renders both existing general approaches, such as integer linear programming, but also existing heuristic approaches, such as iterative modulo scheduling [Rau94], inapplicable. Still, we would like to enable these proven solving techniques. To this effect, we propose a reduction theorem in this chapter, which states that a feasible solution to the RCMSP with unknown loop bounds and number of available PEs can be found as follows: First, solve the RCMSP only for feasibility constraints that do not contain any tile size parameters. Then, from the solution, compute minimal values of the tile size parameters that satisfy all remaining feasibility constraints. These constitute per-dimension minimal tile sizes: As long as the concrete tile sizes at run time are at least the minimal tile sizes in each dimension, the found optimal solution of the reduced RCMSP is a solution to the original RCMSP. In addition, the found solution is latency-minimal if the number of PEs is known beforehand and almost latency-minimal if it is unknown.

Roadmap. Section 4.1 details the problems associated with symbolic modulo scheduling and states the reduction theorem. The subsequent four sections lay the theoretic foundation: Sections 4.2 and 4.3 describe the classification of dependences into simple (no parameters) and complex (with parameters), while Section 4.4 explains how to eliminate the tile size parameters from the latency function. Section 4.5 elaborates on solving the complex feasibility constraints to obtain minimum tile sizes. Using this theoretic foundation, Section 4.6 proves the reduction theorem, including optimality characteristics. Section 4.7 summarizes symbolic modulo scheduling, that is, how to use the reduction theorem to solve the RCMSP to obtain a set symbolic mappings. Since symbolic modulo scheduling is also responsible for the local allocation of functional units and registers, Section 4.8 gives an overview of modeling the resource constraints with the goal of obtaining a so-called resource binding, and Section 4.9 explains how to
include register allocation in symbolic modulo scheduling. Finally, Section 4.10 discusses
the wide applicability and of symbolic modulo scheduling and experimentally shows
the resulting schedules are latency-optimal (when the loop bounds are unknown, but
the number of available PEs is known beforehand) or near latency-optimal (when both
the loop bounds and number of available PEs are unknown).

4.1 Reduction theorem

We consider the RCMSP for a tiled n-dimensional UDA, given its RDG, equivalent to
the following integer linear program [WTH+16]:

\[
\begin{align*}
\text{minimize} & \quad L \\
\text{subject to} & \quad \text{Res, } C^*, \\
& \quad \lambda^* \in \mathbb{Z}^{2n}, \quad \tau \in \mathbb{N}_{\forall o}
\end{align*}
\]

Here, the objective function is the latency \( L \) of the schedule, the set \( C^* \) contains feasibility
constraints derived from the tiled RDG, and \( \text{Res} \) represents the resource constraints.
Intuitively, this integer linear program finds a schedule \((\lambda^*, \tau)\) with minimal latency \( L \)
while satisfying all feasibility and resource constraints.

In our case, the resource constraints \( \text{Res} \) are derived from the architecture description
(see Section 2.1) and fulfill two goals: First, they ensure that the schedule is free of
resource conflicts and does not exceed given resource limits. Second, they model the
local allocation and yield a resource binding of operation nodes to functional units, as
well as of operation to instruction. The concrete nature of the resource constraints and
finding a binding as a byproduct is not relevant to symbolic modulo scheduling and
the following reduction theorem. We therefore postpone a detailed discussion until
Section 4.8, which goes into more detail for TCPAs, including instruction selection and
register allocation.

The feasibility constraints \( C^* \) ensure that the schedule is \emph{feasible}, that is, it does not
violate data dependences. How does a feasibility constraint from a tiled RDG look like?
First, suppose an untiled RDG shall be scheduled. For each dependence edge \( e = (v, w) \)
in the RDG, the corresponding \emph{feasibility constraint} \( c_e \) must be satisfied for the edge’s
annotated dependence vector \( d[e] \):

\[
c_e := \lambda d[e] \geq \tau_v - \tau_w + l_v.
\]

Here, \( l_v \) is the latency of the operation, which equals \( l_{\text{instr}} \) of the selected instruction.
(Note that we write \( \tau_v \) instead of \( \tau [v] \) in the following for notational convenience.) This
constraint is easily derived: If an operation instance \( w[i] \) depends on another operation
instance \( v[i - d] \), as is the case for a dependence edge, the former may only start after
the latter has finished:

\[
t(w, i) \geq t(v, i - d[e]) + l_v
\]
Substituting Definition 3.13 of an affine schedule yields

\[ \lambda i + \tau_w \geq \lambda (i - d[e]) + \tau_v + l_v \iff \lambda i + \tau_w \geq \lambda i - \lambda d[e] + \tau_v + l_v \iff \lambda d[e] \geq \tau_v - \tau_w + l_v. \]

The set of all feasibility constraints of an RDG \((V, E)\) then is

\[ C = \{ c_e := \lambda d[e] \geq \tau_v - \tau_w + l_v \mid \forall e = (v, w) \in E^{dep} \}. \]

Next, suppose a tiled UDA shall be scheduled. As explained in the previous chapter, we assume tiling only implicitly in the corresponding RDG. Thus, each dependence edge \(e\) in the RDG actually represents \(|D^*(d)|\) tiled dependence vectors (see Section 3.4.2), and hence as many tiled feasibility constraints:

\[ C^* = \{ c^* := \lambda^* d^* \geq \tau_v - \tau_w + l_w \mid \forall d^* \in D^*(d[e]), e = (v, w) \in E^{dep} \}. \]

We call a tiled feasibility constraint \(c^*\) an **intra-tile feasibility constraint** if it is induced by an intra-tile dependence \((d^k = 0)\) and an **inter-tile feasibility constraint** otherwise. The set of tiled feasibility constraints \(C^*\) is then the union of the intra-tile feasibility constraints \(C^I\) and inter-tile feasibility constraints \(C^k\). Note that many, but not all of both intra- and inter-tile feasibility constraints contain at least one tile size parameter \(p_i\) because of the symbolic schedule vector \(\lambda^*\) (see Sections 4.2 and 4.3).

Note that the following discourse assumes locally sequential, globally parallel scheduling (LSGP), that is, the intra-tile schedule vector \(\lambda^I\) is derived from a sequential scanning \(s\) (see Section 4.2). As explained later, there may exist multiple scannings \(s\) that satisfy the feasibility constraints of a given UDA. Because there is no feasible scanning that is latency-minimal for all loop bounds and number of PEs [TTH13], we solve the RCMSP once for each feasible \(s\):

\[
\begin{align*}
\text{minimize} & \quad L \\
\text{subject to} & \quad \text{Res}, C^* \\
& \quad \lambda^* \in \Lambda_s, \tau \in \mathbb{N}^{V_{op}}_{\lambda^I}
\end{align*}
\]

Here, \(\Lambda_s\) is the set of all LSGP schedules derived from a scanning \(s\). The resulting (symbolic) schedules from all feasible scannings are collected in a set \(M^{sym}\). At run time, the latency-minimal schedule is then selected among this set (refer to Section 4.7).

It is known that the RCMSP is NP-hard, but as long as the tile sizes and loop bounds are known a priori, solving the corresponding integer linear program is straight-forward using existing exact or heuristic solvers. If they are however not known a priori—as in the case of symbolic modulo scheduling—existing solvers fail to apply because products of the parameters appear in both the objective function and feasibility constraints. To

\[^1\text{However, deriving similar results for locally parallel, globally sequential (LGPS) scheduling is analogous.} \]
mitigate this issue, we propose to instead solve a reduced RCMSP that contains only the subset of $C^*$ not containing parameters, which we call simple feasibility constraints and denote simple($C^*$).

**Definition 4.1.** The reduced RCMSP for a UDA, given its RDG, is:

$$\begin{align*}
\text{minimize} & \quad L \text{ with } p = 1 \\
\text{subject to} & \quad \text{Res}, \text{ simple}(C^*) \\
& \quad \lambda^* \in \Lambda_0, \tau \in \mathbb{N}^{[\nu^0]}.
\end{align*}$$

The subset of constraints containing parameters, which we call complex feasibility constraints and denote complex($C^*$), is satisfied a posteriori by determining minimal tile sizes. The feasibility and optimality of such a solution is given by the reduction theorem:

**Theorem 4.1** (Reduction theorem [WTH’16]). Given an $n$-dimensional UDA and a feasible scanning $s$, a solution $(\hat{\lambda}^*, \hat{\tau})$ to an instance with tile counts $t = i, i \in \mathbb{N}^n$ of the reduced RCMSP is feasible for given tile sizes $p'$ if $p'_i \geq p^\min_i, i = 1, \ldots, n$. Here, $p^\min_i$ are minimal tile sizes that solve the system of inequalities given by the complex feasibility constraints complex($C^*$) with $\lambda^* \mapsto \hat{\lambda}^*$ and $\tau \mapsto \hat{\tau}$ substituted. The solution is latency-minimal for given tile counts $t'$ if it is feasible and $\hat{t} = t'$.

Before we prove the reduction theorem in Section 4.6, the next sections first lay the necessary theoretic foundations and characteristics regarding intra-tile and inter-tile schedule and how to classify the tiled feasibility constraints into simple and complex.

**Remark.** Intra-iteration dependences ($d = 0$ or $d^* = 0$) never yield symbolic feasibility constraints. In the following, we therefore presume that they are part of simple($C^*$).

### 4.2 Classifying the intra-tile feasibility constraints

To classify the intra-tile feasibility constraints $C^j$ into simple and complex, we first need to understand the properties of the intra-schedule vector $\lambda^j$ and how they relate to introducing parameters into the feasibility constraints. With LSGP (Section 3.4), the intra-tile schedule vector $\lambda^j$ represents a linear and tight sequential scanning of the intra-tile space $J$:

- **Sequential scanning** means that $\lambda^j$ is an injective function: Each iteration $j \in J$ is assigned a unique start time $t$. In other words, all iterations in $J$ are assigned a start time, but in no time step more than one iteration is started.

- **Tight** in this context means that the schedule has a latency of exactly $\pi \cdot |J|$ time steps and that $\lambda^j \mod \pi = 0$. In other words, a new iteration is started every $\pi$ cycles. The latter condition stems from modulo scheduling, where a constant number of $\pi$ slots is used to schedule operations in order to avoid resource conflicts.
• **Linear** means that $\lambda^j$ is a linear function (which is trivial, since $\lambda^j$ is a vector in our case).

In the sequel, we refer to linear and tight sequential scannings as just a scanning. For hyperrectangles such as the intra-tile space $\mathcal{J}$, a scanning $s$ can be described by a tuple $(\sigma, \omega)$, where $\sigma$ is a permutation of the first $n$ natural numbers and $\omega \in \{1, -1\}^n$ is an $n$-dimensional vector. The permutation $\sigma$ defines the order, $\omega$ the orientation (in each dimension positive or negative) of the scanning. Consequently, an $n$-dimensional hyperrectangle has $n! \cdot 2^n$ possible scannings. However, many of the $n! \cdot 2^n$ scannings may be pruned in advance because they violate the dependences, that is, they cannot satisfy the feasibility constraints and thus never result in a feasible schedule. For example, if an RDG contains a dependence vector $\mathbf{d} = (1, 0)^T$, any scanning that scans in negative $j_1$ direction (that is, $\omega_1 = -1$) cannot be feasible. Efficient pruning of infeasible scannings is performed in a tree-like fashion; for details, see [TTH14]. After pruning, each of the remaining scannings results in a feasible schedule vector $\lambda^j$ (and by extension $\lambda^*$) that could potentially be latency-minimal at runtime (see Section 4.7).

**Example.** Figure 4.1 depicts the $2! \cdot 2^2 = 8$ scannings for a rectangle. The scanning marked in white, for example, first scans in negative direction $j_2$ and then in positive direction $j_1$:

$$\sigma = (\sigma(1), \sigma(2)) = (2, 1), \quad \omega = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

We specify permutations as tuples: Here, $\sigma(1)$ is 2 (first scanning direction is $j_2$) and $\sigma(2)$ is 1 (second scanning direction is $j_1$). In general, $\sigma(i)$ yields the $i$-th scanning direction and $\omega_i$ the $i$-th scanning orientation.

**Example** (Edge detection). Figure 4.2 shows the RDG of a UDA describing two-dimensional Sobel edge detection. Its set of dependences is

$$D_{edge} = \left\{ \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ 2 \\ 2 \\ 2 \end{pmatrix}, \begin{pmatrix} 3 \\ 3 \\ 3 \\ 3 \end{pmatrix} \right\}.$$

Hence, only the two scannings that scan both dimensions in the positive direction result in feasible schedules:

$$s_1 = (\sigma = (1, 2), \omega = (1, 1)^T) \quad \text{and} \quad s_2 = (\sigma = (2, 1), \omega = (1, 1)^T) \quad \triangle$$

**Example** (Matrix multiplication, continuing from p. 40). Just as the edge detection in the previous example, the presented implementation of a matrix-matrix multiplication has only element-wise positive dependence vectors, resulting in the following six feasible scannings:

$$s_1 = (\sigma = (1, 2, 3), \omega = 1), \quad s_2 = (\sigma = (1, 3, 2), \omega = 1), \quad s_3 = (\sigma = (2, 1, 3), \omega = 1),$$

$$s_4 = (\sigma = (2, 3, 1), \omega = 1), \quad s_5 = (\sigma = (3, 1, 2), \omega = 1), \quad s_6 = (\sigma = (3, 2, 1), \omega = 1)$$
4 Symbolic Modulo Scheduling

Figure 4.1: (Adapted from [WTH+16]) The eight possible scannings for two-dimensional intra-iteration spaces (the beginning of the scanning is marked in bold).

Overall, there are 48 possible scannings for a three-dimensional hyperrectangle, meaning $32/48 = 2/3$ have been pruned.

Example. Assume a given two-dimensional UDA has the set of dependences

$$D = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix} \right\}.$$ 

Because there is no dependence in the second dimension, both $\omega_2 = 1$ and $\omega_2 = -1$ result in feasible schedules. Hence, there are four feasible scannings left:

$$s_1 = (\sigma = (1, 2), \omega = (-1, 1)^T), s_2 = (\sigma = (2, 1), \omega = (-1, 1)^T),$$

$$s_3 = (\sigma = (1, 2), \omega = (-1, -1)^T), s_4 = (\sigma = (2, 1), \omega = (-1, -1)^T)$$

In this case, only $4/8 = 1/2$ of all scannings have been pruned.
4.2 Classifying the intra-tile feasibility constraints

Figure 4.2: RDG of a UDA that describes a Sobel edge detection. ($E$ denotes the identity matrix.)
4 Symbolic Modulo Scheduling

The main scanning direction (or dimension) \( \sigma(1) \) of a scanning \( s \), which basically corresponds to the inner-most loop, is especially important because it does not induce any parameters in the corresponding schedule vector \( \lambda^j \).

**Lemma 4.1** ([WTH\textsuperscript{+}16]). Given an \( n \)-dimensional hyperrectangle of size \( p_1 \times p_2 \times \cdots \times p_n \) and an \( n \)-dimensional scanning \( s = (\sigma, \omega) \), the corresponding intra-tile schedule vector \( \lambda^j \) is given by

\[
\lambda^j_{\sigma(1)} = \omega_1 \pi \quad (4.1)
\]
\[
\lambda^j_{\sigma(k)} = \lambda^j_{\sigma(k-1)} \cdot \omega_k p_{\sigma(k-1)}. \quad (4.2)
\]

**Proof.** An equivalent formulation of this lemma was proved by Darte et al. [DSR\textsuperscript{+}02].

Due to the recursive definition, all elements of \( \lambda^j \) are products of the initiation interval \( \pi \) and tile size parameters \( p_i \), except the element corresponding to the main scanning direction, \( \lambda^j_{\sigma(1)} \), which never contains any parameters.

**Example** (Edge detection, continuing from p. 67). The two feasible intra-tile schedules are:

\[
s_1 = (\sigma = (1, 2), \omega = 1) \implies \lambda^j_1 = \left( \omega_1 \pi \quad \omega_2 p_1 \lambda^j_{1_1} \right) = \left( \pi \quad \pi p_1 \right)
\]
\[
s_2 = (\sigma = (2, 1), \omega = 1) \implies \lambda^j_2 = \left( \omega_1 p_2 \lambda^j_{2_2} \quad \omega_2 \pi \right) = \left( \pi p_2 \quad \pi \right)
\]

In both cases, the main scanning dimension does not contain any tile size parameters (only the initiation interval \( \pi \)). \( \triangle \)

**Example** (Matrix multiplication, continuing from p. 67). The six feasible intra-tile schedules are:

\[
s_1 = (\sigma = (1, 2, 3), \omega = 1) \implies \lambda^j_3 = \left( \pi \quad \pi p_1 \quad \pi p_1 p_2 \right)
\]
\[
s_2 = (\sigma = (1, 3, 2), \omega = 1) \implies \lambda^j_3 = \left( \pi \quad \pi p_1 p_3 \quad \pi p_1 \right)
\]
\[
s_3 = (\sigma = (2, 1, 3), \omega = 1) \implies \lambda^j_3 = \left( \pi p_2 \quad \pi \quad \pi p_1 p_2 \right)
\]
\[
s_4 = (\sigma = (2, 3, 1), \omega = 1) \implies \lambda^j_3 = \left( \pi p_2 p_3 \quad \pi \quad \pi p_2 \right)
\]
\[
s_5 = (\sigma = (3, 1, 2), \omega = 1) \implies \lambda^j_3 = \left( \pi p_3 \quad \pi p_1 p_3 \quad \pi \right)
\]
\[
s_6 = (\sigma = (3, 2, 1), \omega = 1) \implies \lambda^j_3 = \left( \pi p_2 p_3 \quad \pi p_3 \quad \pi \right) \quad \triangle
\]

What does this imply for the intra-tile feasibility constraints induced by intra-tile dependence vectors \( d^* = (d^j, 0)^T \)? First note that for each dependence edge \( e = (v, \omega) \) in the untiled RDG, there is exactly one intra-tile dependence vector \( d^* \) with \( d^j = d[e] \).
Because \( d^k = 0 \) and \( d^j = d[e] = (d_1, \ldots, d_n)^T \), each intra-tile feasibility constraint \( c^j \) is given by

\[
\forall c^j \in C^j: \quad \lambda^k d^k + \lambda^j d^j \geq \tau_v - \tau_w + l_v \iff \lambda^j d[e] \geq \tau_v - \tau_w + l_v.
\]

When we expand the scalar product \( \lambda^j d[e] \) using the definition of an intra-tile schedule from Lemma 4.1, we obtain

\[
\pi \omega_{\sigma(1)} d_{\sigma(1)} + p_1 \pi \omega_{\sigma(2)} d_{\sigma(2)} + \cdots + p_1 p_2 \cdots p_n \pi \omega_{\sigma(n)} d_{\sigma(n)} \geq \tau_v - \tau_w + l_v.
\]

Therefore, if \( d[e] \) is zero for all elements except \( d_{\sigma(1)} \) for a given \( \sigma \), the resulting feasibility constraint does not contain any of the tile parameters \( p_i \) and hence is simple.

**Corollary 4.1 ([WTH+16]).** Given a scanning \( s = (\sigma, \omega) \), an original dependence vector \( d \) induces a simple intra-tile feasibility constraint if

\[
d_i = \begin{cases} 
\alpha \in \mathbb{Z} \setminus \{0\} & \text{if } i = \sigma(1) \\
0 & \text{else}
\end{cases}.
\]

**Example** (Edge detection, continuing from p. 70). Let us look at scanning \( s_2 = (2, 1), \omega = 1 \). The main scanning dimension is \( \sigma(1) = 2 \), therefore only dependence vectors that are zero in all elements except \( d_{\sigma(1)} \) induce simple constraints. Within \( D \), only \( d = (0, 1)^T \) satisfies this, annotated to dependence edges \((v_2, v_4)\) and \((v_5, v_8)\). For example, \((v_2, v_4)\) induces the simple feasibility constraint

\[
\lambda^j d[(v_2, v_4)] \geq \tau_{v_2} - \tau_{v_4} + l_{v_2} \iff \pi \geq \tau_{v_2} - \tau_{v_4} + l_{v_2}.
\]

Overall, this results in the set of simple intra-tile feasibility constraints

\[
\text{simple}(C^j) = \{ \pi \geq \tau_{v_2} - \tau_{v_4} + l_{v_2}, \pi \geq \tau_{v_5} - \tau_{v_8} + l_{v_5} \}.
\]

The remaining dependence vectors

\[
\left\{ \left( \begin{array}{c} 1 \\ 0 \end{array} \right), \left( \begin{array}{c} 1 \\ 1 \end{array} \right), \left( \begin{array}{c} 2 \\ 1 \end{array} \right), \left( \begin{array}{c} 1 \\ 2 \end{array} \right), \left( \begin{array}{c} 2 \\ 2 \end{array} \right) \right\}
\]

result in complex feasibility constraints

\[
\text{complex}(C^j) = \{ \pi p_2 \geq \tau_{v_2} - \tau_{v_3} + l_{v_2}, \ldots \}.
\]
These findings support the reduction theorem: Dependence vectors \( d = i' - i \) that induce simple feasibility constraints influence the schedule the most because they are parallel to the main scanning direction and therefore the time \( \lambda^j \cdot d \) between the two involved iterations is small. The smaller \( \lambda^j \cdot d \) is, the more likely it is to be not long enough for the depended-on operation \( \text{op}[v] \) to finish in time. Including the simple feasibility constraints in the modulo scheduling problem is thus paramount to result in an optimal solution. On the other hand, dependence vectors that yield complex constraints do in general not influence the solution as much since the tile sizes \( p_i \) are generally large, resulting in \( \lambda \cdot d \) to be larger. The larger, the more likely it is that a given schedule already satisfies the given dependence.

4.3 Classifying the inter-tile feasibility constraints

In contrast to the intra-tile schedule vector \( \lambda^j \), time steps assigned to the start of a tile by the inter-tile schedule vector \( \lambda^k \) need not be divisible by \( \pi \). This is because there is no need to adhere to resource constraints (which are upheld by the initiation interval) when a PE is started, since none of its resources are yet occupied. Therefore, the earliest time step a PE may start in is when all operations have finished upon whose results the temporally first iteration of the tile depends. This time step depends on the time offsets \( \tau \). However, during the classification of the inter-tile feasibility constraints, we do not yet know \( \tau \) (recall that classification is performed before solving the reduced RCMSP), and can therefore not determine \( \lambda^k \) accurately. However, we can determine a lower bound on how early PEs may start imposed only by the dependence vectors, but independent of \( \tau \). Figure 4.3 illustrates the relation: In no case may the first iteration \( j_1 \) of the temporally next tile in direction \( k_1 \) start before its dependence on iteration \( j_0 \) is satisfied.

In the following, we call \( b_i = \pm e_i \) (where \( e_i \) denotes the \( i \)-th standard basis vector) the \( i \)-th basis vector or simply basis vector in direction \( i \).

**Lemma 4.2 ([WTH+16]).** Assume the set of dependence vectors \( D \) of an RDG contains either \( b_i = e_i \) or \( b_i = -e_i \) for \( 1 \leq i \leq n \). Then, given an intra-tile schedule vector \( \lambda^j \) that is a scanning, an element-wise lower bound on a corresponding inter-tile schedule vector \( \lambda^k \) is given by

\[
|\lambda^k_i| > |\lambda^j_i(p_i - 1)|, \quad i = 1, \ldots, n.
\] (4.3)

**Remark.** According to these assumptions, while \( D \) can never contain both \( e_i \) and \( -e_i \) because no scanning would be feasible (see Section 4.2), \( D \) has to contain one of them. Therefore, even if \( D \) contains neither, we assume it does; although this has no implications on feasibility, it might make the schedule slightly worse than a latency-minimal one because \( \pm e_i \) are the shortest dependence vectors and hence apply the most pressure to the schedule. Note that the sign of \( \pm e_i \) determines the direction of communication between processing elements (for example, whether the data streams
from left to right or vice versa). Always assuming the inclusion of either \(e_i\) or \(-e_i\) therefore has the same effect as forward communication only, a constraint that restricts communication between processing elements in each dimension in order to avoid communication deadlocks [GFG02; GFL04].

**Proof.** An inter-tile feasibility constraint induced by \(d^* = (d^i, d^k)^T \in D^*(d[e])\) of an edge \(e = (v, w)\) is given by

\[
\lambda^k d^k + \lambda^i d^i \geq \tau_v - \tau_w + l_v.
\]

However, we do not yet know \(\tau_v, l_v,\) and \(\tau_w\). To find a lower bound for each element of \(\lambda^k\), we instead look at the strictest possible case, where there is only one time step between the production of the corresponding value by the operation of \(v\) and consumption by \(w\):

\[
\lambda^k d^k + \lambda^i d^i \geq 1
\]

Assuming \(b_i = \pm e_i \in D\), only inter-tile dependences \(D^*(b_i)\) induced by \(b_i\) are relevant (see [TTH13]). (This is because \(b_i\) are the shortest and thus strictest loop-carried dependence vectors.) From symbolic tiling (see Section 3.4.2), we know that \(|D^*(b_i)| = 2\), of which one is an intra-tile dependence (already classified in the previous section), leaving only a single inter-tile dependence vector. Since that inter-tile dependence vector has only \(d^i\) and \(d^k\) not equal to zero, it induces the following inter-tile feasibility constraint:

\[
\lambda^i d^i \cdot \sgn(b_i) + \lambda^j d^j \cdot \sgn(b_i)(1 - p_i) \geq 1,
\]

where \(b_i = (\lambda^i d^i, \lambda^j d^j)\) \(i = 1, \ldots, n\) (4.3)

which can be transformed into Eq. (4.4) by dividing by \(\sgn(b_i)\):

\[
\lambda^i d^i > \lambda^j d^j \cdot (p_i - 1) \text{ if } \sgn(b_i) = 1
\]

\[
\lambda^i d^i < \lambda^j d^j \cdot (p_i - 1) \text{ if } \sgn(b_i) = -1
\]

\[\iff |\lambda^i| > |\lambda^j(1 - p_i)| \]

When taking the time offsets \(\tau\) into account, \(|\lambda^i|\) is generally greater than the lower bound from Lemma 4.2 by a natural number \(\epsilon_i\) that only depends on \(\tau\) and the instruction latencies \(l_v\). Consequently, \(\epsilon_i\) does not contain any parameters. If we define \(\lambda^k\) in terms of the lower bound and \(\epsilon_i\), we can thus classify the inter-tile feasibility constraints into simple and complex without known \(\tau\). Minimizing \(\epsilon_i\) is then left to the solver when solving the reduced RCMSP.

**Lemma 4.3 ([WTH+16]).** Given an intra-tile schedule vector \(\lambda^i\) that is a scanning, a corresponding inter-tile schedule vector \(\lambda^k\) is defined by

\[
\lambda^k_i = \lambda^i(p_i - 1) + \sgn(b_i)\epsilon_i, \quad i = 1, \ldots, n
\]

where we call \(\epsilon = (\epsilon_1, \ldots, \epsilon_n) \in \mathbb{N}^n\) inter-tile offsets. The inter-tile schedule is feasible if it satisfies the reduced set of inter-tile feasibility constraints

\[
C^{\epsilon_k} := \{\lambda^i(d - b_i) + \epsilon_i \geq \tau_v - \tau_w + l_v \mid \forall d \in D: d_i \neq 0, i = 1, \ldots, n\}.
\]

73
Figure 4.3: (From [WTH+16]) Inter-tile dependences that influence $\lambda^k_j$. Iteration $j_1$ in the temporally next tile may only start once it is guaranteed that the dependence on $j_0$ is satisfied not only for $j_1$, but also $j_2$ to $j_4$.

**Proof.** Since the assumptions are the same, analogously to the previous proof, it suffices to satisfy only the inter-tile feasibility constraints $d^* \in D^*(b_i)$ induced by $b_i$, that is only feasibility constraints of the form

$$\lambda^k_i \text{sgn}(b_i) + \chi^j_i d^j \geq \tau_v - \tau_w + l_v,$$

because $d^k = b_i$. Substituting Equation (4.4) for $\lambda^k_i$ yields

$$\left(\lambda^j_i(p_i - 1) + \text{sgn}(b_i)\epsilon_i\right) \cdot \text{sgn}(b_i) + \chi^j_i d^j \geq \tau_v - \tau_w + l_v \iff \lambda^j_i(p_i - 1) + \epsilon_i + \chi^j_i d^j \geq \tau_v - \tau_w + l_v.$$ 

According to the foundations of symbolic tiling (see Section 3.4.2),

$$d^k = b_i \implies d^j_\ell = \begin{cases} d_i - \text{sgn}(b_i)p_i & \text{for } i = \ell, \\ d_\ell & \text{else.} \end{cases}$$

Substituting these values for $d^j_\ell$ results in:

$$\text{sgn}(b_i)\lambda^j_i(p_i - 1) + \epsilon_i + \lambda^j_i(d_i - \text{sgn}(b_i)p_i) + \sum_{1 \leq \ell \leq n, \ell \neq i} \lambda^j_\ell d^j_\ell \geq \tau_v - \tau_w + l_v \iff \lambda^j_i(\text{sgn}(b_i)p_i - \text{sgn}(b_i)) + \epsilon_i + \lambda^j_i(d_i - \text{sgn}(b_i)p_i) + \sum_{1 \leq \ell \leq n, \ell \neq i} \lambda^j_\ell d^j_\ell \geq \tau_v - \tau_w + l_v$$

Summing up the terms containing $\lambda^j_i$ yields

$$\lambda^j_i(d_i - \text{sgn}(b_i)) + \sum_{1 \leq \ell \leq n, \ell \neq i} \lambda^j_\ell d^j_\ell + \epsilon_i \geq \tau_v - \tau_w + l_v,$$

which is equivalent to the feasibility constraints given in Equation (4.5) when $1 \leq i \leq n$, which was to be proven. \qed
The subtraction of $b_i$ may be interpreted intuitively as follows: Assume a tile $k$ (PE) is currently being executed. According to the definition of $\lambda^i_k$ in Lemma 4.3, the next tile in dimension $i$ (that is, $k' = k \pm e_i$) starts with a time offset of

$$\lambda^i_j(p_i - 1) + \text{sgn}(b_i)e_i.$$  

That is, it is based on the first iteration $j_{\text{base}}$ within the last “row” of the current tile $k$, only offset by an as-of-yet unknown $\epsilon_i$. Therefore, the subtraction of $b_i$ in the inter-tile feasibility constraints rebases the start of the next tile $k'$ in dimension $i$ to iteration $j_{\text{base}}$ of the current tile $k$. As an example, in Figure 4.3, $j_1$ is rebased to $j_0$. In other words, disregarding any dependences, the earliest time $j_1$ may start is simultaneously with $j_0$. However, due to the way $\epsilon_1$ is determined, $j_1$ actually starts as soon as all dependences from $j_0$ to $j_1$ are satisfied. This serves two purposes: First, $k'$ may start before $k$ has finished. Second, the initiation interval is ignored, which makes sense because $k'$ starts executing on a PE with free resources.

**Corollary 4.2 ([WTH+16]).** Given a scanning $s = (\sigma, \omega)$, an original dependence vector $d$ induces a simple inter-tile feasibility constraint in dimension $i = 1, \ldots, n$ if

$$(d - b_i)_{j\neq \sigma(1)} = 0.$$  

This corollary follows directly from Lemmas 4.1 and 4.3: All elements of the intra-tile schedule vector $\lambda^j_i$ contain parameters except the element corresponding to the main scanning dimension, $\lambda^j_{\sigma(1)}$. Since $\epsilon$ also never contains parameters, the result of the scalar product $\lambda^j_i(d - b_i)$ in Equation (4.5) does not contain parameters if all elements of $d - b_i$ are zero with the possible exception of element $\sigma(1)$.

**Example** (Edge detection, continuing from p. 70). The inter-tile feasibility constraints induced by the original dependence vectors need to be classified individually for both dimensions $i \in \{1, 2\}$. Assume $s_2 = (\sigma = (2, 1), \omega = 1)$ and therefore $\sigma(1) = 2$, that is, the main scanning dimension is in positive $j_2$. For example, for dimension $i = 1$, the following dependence vector induces a simple inter-tile feasibility constraint:

$$d[(v_2, v_6)]: \begin{pmatrix} 1 \\ 1 \end{pmatrix} - b_1 = \begin{pmatrix} d_{\sigma(2)} = 0 \\ d_{\sigma(1)} = 1 \end{pmatrix} \implies c := \pi + \epsilon_1 \geq \tau_{v_2} - \tau_{v_6} + l_{v_2},$$

because after subtracting $b_1$, all elements are zero except for the main scanning dimension $d_{\sigma(1)} = 1$. In contrast, the following dependence vector induces a complex inter-tile feasibility constraint for $i = 1$:

$$d[(v_5, v_8)]: \begin{pmatrix} 2 \\ 1 \end{pmatrix} - b_1 = \begin{pmatrix} d_{\sigma(2)} = 1 \\ d_{\sigma(1)} = 1 \end{pmatrix} \implies c := p_1\pi + \pi + \epsilon_1 \geq \tau_{v_5} - \tau_{v_8} + l_{v_5},$$

because $d_{\sigma(2)} \neq 0$; note the tile size parameter $p_1$, which makes the constraint complex.
Considering all dependence vectors $d \in D$ and $i \in \{1, 2\}$, the dependence vectors
\[
\begin{cases}
(1, 0), (1, 1), (1, 2) & \text{for } i = 1, \\
(0, 1) & \text{for } i = 2
\end{cases}
\]
induce simple feasibility constraints. On the other hand, the dependence vectors
\[
\begin{cases}
(2, 1) & \text{for } i = 1, \\
(1, 2), (2, 1) & \text{for } i = 2
\end{cases}
\]
result in complex feasibility constraints. The remaining dependences with $d_i = 0$ are irrelevant because they are already satisfied by the intra-tile schedule.

**Example** (Matrix multiplication, continuing from p. 70). Here, the intra-tile feasibility constraints induced by all original dependence vectors must be classified for all three dimensions $i \in \{1, 2, 3\}$, but is otherwise analogous to the example above.

**Remark.** Using Corollaries 4.1 and 4.2, it is possible to classify both intra- and inter-tile feasibility constraints using only the original dependence vectors $d \in D$. This eschews the exponential growth of the number of tiled dependence vectors, strengthening our decision to work on untiled RDGs.

### 4.4 Eliminating tile size parameters from the latency function

Recall the reduced RCMSP:

\[
\begin{align*}
\text{minimize} & \quad L \text{ with } p = 1 \\
\text{subject to} & \quad \text{Res, simple}(C^*) \\
& \quad \lambda^* \in \Lambda_0, \tau \in \mathbb{N}^{\|V^p\|}
\end{align*}
\]

So far, we showed how to determine the simple feasibility constraints $\text{simple}(C^*)$. Next, we prove that the objective function, the latency $L$, orders two schedules $(\lambda^*_1, \tau_1)$ and $(\lambda^*_2, \tau_2)$ the same according to $L(\lambda^*_1, \tau_1) \leq L(\lambda^*_2, \tau_2)$, independent of $p$. This means $L$ evaluated at $p = 1$ is an equivalent objective function for the reduced RCMSP.

First, let us recapitulate the latency function $L$. The latency $L$ of a schedule for a tiled iteration space $I^*$ is given by the time span of operations in execution as follows:

\[
L = \max_{v \in V^p, i^* \in I^*} (t(v, i^*) + l[v]) - \min_{v \in V^p, i^* \in I^*} t(v, i^*)
\]

For an LSGP schedule, we have
\[
t(v, i^*) = \lambda^* i^* + \tau_v, \quad \lambda^* = (\lambda^j, \lambda^k).
\]
Therefore,

\[ L = \max_{\nu \in V^{op}, i^* \in I^*} (\lambda^* i^* + \tau_{\nu} + l[\nu]) - \min_{\nu \in V^{op}, i^* \in I^*} (\lambda^* i^* + \tau_{\nu}) \]

\[ = \max_{i^* \in I^*} \lambda^* i^* + \max_{\nu \in V^{op}} (\tau_{\nu} + l[\nu]) - \min_{i^* \in I^*} \lambda^* i^* - \min_{\nu \in V^{op}} \tau_{\nu} . \]

Next, let \( i^*_{\text{first}} \) be the temporally first iteration and \( i^*_{\text{last}} \) the temporally last iteration:

\[ i^*_{\text{first}} = \arg \min_{i^*} \lambda^* i^*, \quad i^*_{\text{last}} = \arg \max_{i^*} \lambda^* i^* \]

Then,

\[ L = \lambda^* i^*_{\text{last}} - \lambda^* i^*_{\text{first}} + \max_{\nu \in V^{op}} (\tau_{\nu} + l_{\nu}) . \]

Further transforming \( L \) yields:

\[ L = \lambda^k k_{\text{last}} + \lambda^j j_{\text{last}} - \lambda^k k_{\text{first}} - \lambda^j j_{\text{first}} + \max_{\nu \in V^{op}} (\tau_{\nu} + l_{\nu}) \]

\[ = \lambda^k (k_{\text{last}} - k_{\text{first}}) + \lambda^j (j_{\text{last}} - j_{\text{first}}) + \max_{\nu \in V^{op}} (\tau_{\nu} + l_{\nu}) \]

Note that \( k_{\text{last}} - k_{\text{first}} + 1 \) is equal to the vector \( t = (t_1, \ldots, t_0)^T \), where \( t_i \) denotes the number of tiles in dimension \( i \) as given by symbolic tiling (Section 3.4.2). For brevity, we call this vector simply the number of tiles in the sequel. Also, since \( \lambda^j \) is a tight sequential scanning that schedules iterations \( \pi \) time steps apart, it holds by definition that

\[ \lambda^j (j_{\text{last}} - j_{\text{first}}) = \pi \left( \prod_{i=1}^{n} p_i \right) - 1 . \]

Incorporating these two observations, we arrive at

\[ L = \lambda^k (t - 1) + \pi \left( \prod_{i=1}^{n} p_i - 1 \right) + \max_{\nu \in V^{op}} (\tau_{\nu} + l_{\nu}) . \]

Here, the global latency \( L_{\text{global}} \) describes the number of time steps from the start of the temporally first iteration to the start of the last. The local latency \( L_{\text{local}} \) describes the latency of a single iteration. To make it clear which variables \( L \) depends on, we use function notation in the following and write \( L(\lambda^*, \tau, t, p) \).

Note that we assume both the initiation interval \( \pi \) and scanning \( s \) fixed during solving the RCMSP. We start by making \( \pi \) a constant instead of a variable to be solved; this
might seem detrimental, but is the usual practice [RG81]: \( \pi \) is initialized with a minimum value (derived from the feasibility and resource constraints) and repeatedly solved until a solution is found, incrementing \( \pi \) after each failed solving attempt. This works for both the integer linear program formulation (see, for example, [ED97]) and for heuristic algorithms (such as iterative modulo scheduling [Rau94]). We account for this by restricting the schedules to the set of all LSGP schedules \( \Lambda_{s,\pi} \) arising from \( s \) and \( \pi \) derived from \( s \) and \( \pi \).

**Lemma 4.4** ([WH+16]). Given tile counts \( i \in \mathbb{N}^n \), let \( (\lambda_1^*, \tau_1) \) and \( (\lambda_2^*, \tau_2) \) be two schedules with \( \lambda_1^* \in \Lambda_{s,\pi} \) and \( \lambda_2^* \in \Lambda_{s,\pi} \). Then:

\[
L(\lambda_1^*, \tau_1, i, p_1) \leq L(\lambda_2^*, \tau_1, i, p_1) \quad \Leftrightarrow \quad L(\lambda_1^*, \tau_2, i, p_2) \leq L(\lambda_2^*, \tau_2, i, p_2), \quad \forall p_1 \in \mathbb{N}^n, p_2 \in \mathbb{N}^n
\]

**Proof.** First, we collect all terms containing \( p \) in the latency function \( L \) into a function \( X_{s,\pi}(t, p) \):

\[
L(\lambda^*, \tau, t, p) = \lambda^*(t - 1) + \pi (\prod_{i=1}^n p_i - 1) + L_{local}
\]

\[
\begin{align*}
\equiv (4.4) & \left( \sum_{i=1}^n \lambda_i^*(p_i - 1) + \varepsilon_i \cdot \text{sgn}(b_i) \right) (t - 1) + \pi (\prod_{i=1}^n p_i - 1) + L_{local} \\
= \text{sgn}(b_i)(t - 1) \sum_{i=1}^n \lambda_i^*(p_i - 1) + \pi (\prod_{i=1}^n p_i - 1) + (t - 1)^T \varepsilon + L_{local}
\end{align*}
\]

\[
X_{s,\pi}(t, p)
\]

As per Lemma 4.1, the intra-tile schedule elements \( \lambda_i^* \) only depend on the tile sizes \( p \), the scanning \( s \), and the initiation interval \( \pi \), the latter two of which we assume fixed.

Now, given two schedules \( (\lambda_1^*, \tau_1) \) with \( \lambda_1^* \in \Lambda_{s,\pi} \) and \( (\lambda_2^*, \tau_2) \) with \( \lambda_2^* \in \Lambda_{s,\pi} \):

\[
L(\lambda_1^*, \tau_1, i, p_1) \leq L(\lambda_2^*, \tau_2, i, p_1) \quad \Leftrightarrow \quad X_{s,\pi}(i, p_1) + L'(\varepsilon_1, \tau_1, i) \leq X_{s,\pi}(i, p_1) + L'(\varepsilon_2, \tau_2, i) \\
\quad \Leftrightarrow \quad L'(\varepsilon_1, \tau_1, i) \leq L'(\varepsilon_2, \tau_2, i) \\
\quad \Leftrightarrow \quad X_{s,\pi}(i, p_2) + L'(\varepsilon_1, \tau_1, i) \leq X_{s,\pi}(i, p_2) + L'(\varepsilon_2, \tau_2, i) \\
\quad \Leftrightarrow \quad L(\lambda_1^*, \tau_1, i, p_2) \leq L(\lambda_2^*, \tau_2, i, p_2)
\]

\[\Box\]

In Definition 4.1 of a reduced RCMSP, \( p = 1 \) is a good choice because \( X_{s,\pi}(t, 1) = 0 \), thus eliminating all terms from \( L \) except \( L' \). However, \( L' \) still depends on \( i \), which is also an unknown parameter. The reduction theorem states to solve the reduced RCMSP for a specific \( i \in \mathbb{N}^n \); the determined schedule is then latency-minimal by construction.
4.5 Calculating the minimal tile sizes

for \( t \) if \( t = t' \) for a given number of tiles \( t' \)—that is, for a given number of available PEs. If, however, \( t' \neq t \), then the determined schedule is almost optimal within the bounds given by the following corollary.

**Corollary 4.3.** Let a schedule \( (\lambda_i^*, \tau_i) \) with \( \lambda^* \in \Lambda_{x, \pi} \) be given that minimizes \( L \) for tile counts \( t \in \mathbb{N}^n \). Given a schedule \( (\lambda^*, \tau_i^*) \) with \( \lambda^* \in \Lambda_{x, \pi} \) that minimizes \( L \) for another tile counts \( t' \in \mathbb{N}^n \):

\[
L(\lambda_i^*, \tau_i^*, t') - L(\lambda_i^*, \tau_i, t') \leq D := \sum_{i=1}^{n} \pi(t_i^* - 1) + |L_{local,t} - L_{local,t'}|
\]

**Proof.** First,

\[
L(\lambda_i^*, \tau_i^*, t') - L(\lambda_i^*, \tau_i, t') = L'(\epsilon_i t', \tau_i^*, t') - L'(\epsilon_i, \tau_i, t')
\]

\[
= (t' - 1)^T (\epsilon_i t' - \epsilon_i) + L_{local,t'} - L_{local,i}
\]

\[
\leq |(t' - 1)^T (\epsilon_i t' - \epsilon_i)| + |L_{local,t'} - L_{local,i}|
\]

By construction \( 0 \leq \epsilon_i \leq \pi \) and therefore \( |\epsilon_{i, \min} - \epsilon_i| \leq \pi \):

\[
D_1 \leq \sum_{i=1}^{n} \pi(t_i^* - 1)
\]

In most cases, \( D \) is negligibly small compared to the overall latency \( L \) because \( \pi, t' \), and the local latency \( L_{local,t'} \) of a schedule are usually small. The initiation interval is very often below 10, the number of PEs below 100, and the local latency below the sum of operation latencies. In cases where \( D \) is not negligibly small, there is most likely an imbalance between the size of iteration space and the number of PEs used to accelerate the loop.

One question remains: For which \( p \) is a solution \( (\lambda^*, \tau) \) to the reduced RCMSP feasible? Obviously, neither the objective function nor the simple feasibility constraints contain the tile size parameters \( p \) anymore, meaning any feasibility constraints containing \( p \) must be satisfied elsewhere.

4.5 Calculating the minimal tile sizes

We know that a solution \( (\lambda^*, \tau) \) to the reduced RCMSP according to Definition 4.1 satisfies both the resource and simple feasibility constraints. It remains to be shown how to finally also satisfy the complex feasibility constraints. There are two possibilities:
1. Check at run time whether the concrete tile sizes $p_i'$ satisfy the complex feasibility constraints. While this never results in false negatives and each individual check is relatively cheap (there is no solving involved, only checking whether an inequality is satisfied), it may still be computationally expensive because of the potentially large number of complex feasibility constraints.

2. Precompute minimal tile sizes $p_i^{\text{min}}$ at compile time and compare the concrete tile sizes $p_i'$ individually against them at run time:

$$\lambda^* \text{ is feasible } \iff p_i' \geq p_i^{\text{min}}, \ 1 \leq i \leq n$$

While this is computationally cheap, there may be false negatives. For example, assume there are two complex feasibility constraints $p_1 \geq 1$ and $p_2 + p_1 \geq 4$. Let $p_1^{\text{min}} = 1$ and $p_2^{\text{min}} = 3$. These values ensure that the two constraints are always satisfied. However, $p_1' = 2$ and $p_2' = 2 < 3$ would be considered infeasible although they satisfy the two given constraints.

Usually, the concrete tile sizes $p_i'$ are quite large, and the minimal tile sizes are usually quite small (see Section 4.10). Hence, false negatives are very rare, and even if one occurs, using a fallback schedule (see below) would not drastically increase the latency because of the small tile size. Therefore, we propose to precompute minimal tile sizes according to the simple scheme explained in the following.

Due to the regular structure of the intra-tile schedule vector $\lambda^i$, the complex feasibility constraints complex($C^*$) can be classified into equivalence classes as follows:

$$[c]_i = \{c \in \text{complex} (C^*) \mid c \text{ contains } p_{\sigma(i)} \text{ but not } p_{\sigma(i+1)}, \ldots, p_{\sigma(n)}\}.$$ 

This results in a triangular system of inequalities: $[c]_1$ contains only $p_{\sigma(1)}$, $[c]_2$ contains $p_{\sigma(2)}$ and possibly $p_{\sigma(1)}$, and so on. We determine the minimal tile sizes $p_i^{\text{min}}$ by first solving $[c]_1$ for $p_i^{\text{min}}$ for $p_{\sigma(1)}$, then use this to solve $[c]_2$ for $p_i^{\text{min}}$, and so on.

In case the concrete tile sizes $p_i'$ do not meet the minimal tile size requirements at run time, we propose to use a fallback schedule ($\lambda^*_f, \pi_f$) that is always feasible, independent of the tile size parameters. A fallback schedule can be found by incrementing $\pi$ and solving the reduced RCMS until $p_{i}^{\text{min}} = 1$ for all $1 \leq i \leq n$. This is because increasing $\pi$ relaxes the feasibility constraints: $\lambda^* d^* \geq L_{\text{local}}$. In this scalar product, each term is multiplied by $\pi$; therefore, increasing it increases the value on the left-hand side of the constraint, hence relaxing it. Note that if already $p_{i}^{\text{min}} = 1$ for $1 \leq i \leq n$ for at least one schedule, then no fallback schedule is required.

4.6 Proving the reduction theorem

**Proof.** The solutions to the RCMS lie on a polyhedron induced by the resource constraints $Res$ and feasibility constraints $C^*$. The reduction theorem makes two claims we
need to prove: (1) a solution $S = (\lambda^*, \tau)$ to the RCMSP with $t = \hat{t}$ is feasible if $p'_i \geq p^\min_i$ for $i = 1, \ldots, n$, and (2) the solution is latency-minimal if $i = t'$.

First, we prove feasibility: Because $S$ is a solution to the reduced RCMSP, all resource and simple feasibility constraints are already satisfied by construction. The minimal tile sizes $p^\min$ are constructed by solving the system of inequalities given by the complex feasibility constraints, each of the form

$$\lambda^* d^* \geq \tau_v + \tau_w + \ell_v.$$ 

As implied by Lemmas 4.1 and 4.3, the left-hand side of this inequality increases monotonically with $p_i$. Therefore, the inequality is still satisfied for any $p'_i \geq p^\min_i$, proving claim (1). Note that feasibility is independent of $t'$.

Second, we prove optimality, which consists of two steps: (a) show that omitting the complex feasibility constraints does not change where $L$ is minimal given $s$, $\pi$, and $t$, and (b) that omitting the complex feasibility constraints does not influence the minimal $\pi$.

Lemma 4.4 already showed that for a given initiation interval $\pi$, scanning order $s$, and tile counts $\hat{t}$, the order defined by the objective function $L$ is independent of $p$. It remains to be shown that omitting the complex feasibility constraints does not change the minimum. The minima of $L$ lie on a polyhedron $F$ spanned by the feasibility and resource constraints of the given UDA. In particular, $F$ is the intersection of the half-spaces spanned by the constraints:

$$F = \bigcap_{c \in C^*} H_c$$

Intersecting a polyhedron with a half-space $H$ may only shrink the polyhedron, never grow it, that is

$$F \cap H \subseteq F.$$ 

It follows that omitting the complex feasibility constraints may only grow $F$:

$$\bigcap_{c \in C^*} H_c \subseteq \bigcap_{c \in \text{simple}(C^*)} H_c$$

Furthermore, the objective function $L$ is monotonically increasing; making $F$ (that is, the domain of $L$) larger may only decrease the minimum of $L$, while shrinking $F$ may only increase the minimum of $L$:

$$\min L|_F \leq \min L|_{F \cap H}$$

Omitting the feasibility constraints from the RCMSP thus never shrinks $F$ and the minimum of $L$ is never increased. Finally, reintroducing the complex feasibility constraints with $p \geq p^\min$ may only shrink $F$, therefore it may only ever increase the minimum
4 Symbolic Modulo Scheduling

$L$ takes. However, by construction of $p^\text{min}$, the solution $S$ still lies on $F$, meaning the minimum of $L$ remains unchanged.

The same reasoning applies to $\pi$: Omitting the complex feasibility constraints makes $F$ larger, therefore, if $\pi$ is feasible in the original RCMSP, it is also feasible in the reduced RCMSP. Thus, (2) is proved.

Note that there are edge cases where a higher $\pi$ results in a lower latency because the latency $L$ is dominated by the local latency because of an exceedingly small iteration space; however, this issue is not unique to symbolic modulo scheduling, but all techniques that iteratively solve for $\pi$. In the case of symbolic modulo scheduling, it only makes sense to choose the smallest $\pi$ regardless of these edge cases because in the usual case this leads to the minimum latency.

4.7 Summary of symbolic modulo scheduling

Let us concretize how to use the reduction theorem to yield a set of potentially latency-minimal symbolic mappings $M^{\text{sym}}$. We call each $\mu \in M^{\text{sym}}$ a mapping candidate to indicate that it depends on the concrete tile sizes $p$ which candidate is latency-minimal. For a UDA, given its RDG and a space mapping $\phi$:

1. Find a feasible scanning order $s = (\sigma, \omega)$ (see [TTH14]).

2. From the scanning order $s$, determine the simple feasibility constraints $\text{simple}(C^*)$.

3. Solve the reduced RCMSP according to Definition 4.1 for $(\lambda^*, \tau)$ iteratively from $\pi = 1$ using $t' = t'$ if the number of PEs $t'$ to map to is known; otherwise adequately estimate $t$.

4. Annotate the start offsets $\tau^{opt}$ and results of local allocation (instruction latency, registers, ...) back to the RDG.

5. Compute the minimal tile sizes $p^\text{min}_i$ from the complex feasibility constraints $\text{complex}(C^*)$.

6. Insert $m = (\phi, \lambda^*, \tau)$ into $M^{\text{sym}}$.

7. Repeat from 1) until no feasible scannings remain.

Depending on the concrete tile sizes $p'$, which are only known at run time, any of the mapping candidates $\mu \in M^{\text{sym}}$ could be the global optimum. Therefore, at run time, the latency-minimal mapping $m$ according to $p'$ is selected (see [TTH13]). Moreover, if none of mappings is feasible because $p'$ is element-wise smaller than the minimal tile sizes $p^\text{min}$ associated with the schedule of that mapping, then the fallback schedule $\lambda^*_i$ (Section 4.5) is selected instead, which is always feasible.
Remark. We investigate the inverse problem—finding a number of PEs that satisfies given latency bounds—in Chapter 7.

4.8 Resource binding and instruction selection

Besides finding a latency-minimal schedule, the secondary goal of solving the RCMSP is to perform a local allocation of functional units, registers, and routes on the interconnect. In particular, each operation node $v$ of the RDG being scheduled must be assigned an instruction that realizes the operation $op[v]$ and a functional unit that executes that instruction. Because each functional unit supports only a limited set $instr_{fu}$ of instructions with varying limitations (for example, the minimum and maximum value of an immediate operand), modeling which instructions of which functional units can be bound to which operations in the RDG is necessary. We represent these binding possibilities by a bipartite graph called the resource graph [TTZ96]: The first partition consists of all operation nodes $v \in V^{op}$, the other partition has a node for each instruction of each functional unit, annotated with the properties of the instruction (in particular, the latency $l_{instr}$ and pipeline interval $\pi_{instr}$. Two nodes are connected if the instruction represented by the head node can be bound to the operation represented by the tail node. Simply put, binding is possible if:

- The instruction realizes the operation $op[v]$ of the operation node $v$. For example, a multiplication by a power of 2 can be realized by a $mul$ instruction or a $shli$ (shift left immediate) instruction.

- The instruction has the same types of operands as the operation node. For example, for an operation node $v$ with $op[v] = \text{add}(\#0, \#1)$ where operand $\#0$ comes from a register (that is, from another operation or input node) and $\#1$ comes from a constant node, then only an instruction matches that takes a register as first operand and an immediate value as second operand\(^2\), as indicated by the placeholders in the instruction templates (see Section 2.1).

- The operation operands satisfy any limitations put on the instruction operands. For example, if an operation “$\text{add}(\#0, \#1)$” with $\#1 = 2^n$ cannot be bound to an instruction $\text{addi} \ rd \ rs \ imm$ if $imm$ may only be within $[0, 2^n)$.

This graph of binding possibilities is generated from the RDG to be scheduled and the architecture description of the targeted TCPA, in particular the set $FU$ of functional units of each PE and their instruction sets $instr_{fu}$. Additionally, the compiler is provided with a database that contains information about which instruction realizes which operations, as well as its operand types, limitations, and instruction properties. Since the details are functional unit- and implementation-specific, we omit any details here.

\(^2\)Note that at the current time, it is an error if not all edges representing the same operand of an operation node have the same type. However, any RDG can be transformed into such a form.
Given the resource graph, the resource constraints $Res$ are formulated such that exactly one instruction is selected for each operation node, but each functional unit is executing at most one instruction per time step. They must take into account the number of available functional units as well as the timing properties of the selected instruction (latency and pipeline rate). Examples of integer linear program formulations of resource constraints used in modulo scheduling are, for example, given in [TTZ96; ED97; Han09].

### 4.9 Register allocation

Code generation (see Chapter 5) requires each dependence, input, and output edge to be annotated with registers used to store the value communicated via the edge from production until consumption. Note that an edge may have different registers for reading and writing (for example, od0 for writing and id1 for reading to allow for inter-processor communication). For convenience, we simply write “value of edge $e$” instead of “value produced/consumed via $e$” or similar in the following.

Before we turn our focus to register allocation itself, recall that an operation node $v$ may have multiple incoming edges for the same operand that need to be disambiguated. An edge is “read” by node $v$ in all iterations $i$ that belong to the edge’s read space.
4.9 Register allocation

Definition 4.2. Given an iteration space $I$, a condition space $I'$, and a vector $d$, the translation $I' + d$ is defined as

$$I' + d := \{ i + d | i \in I' \cap I \wedge i + d \in I \}$$

The translation $I' - d$ is defined analogously.

Definition 4.3 ([WHT19]). Let $e = (v, w)$. The read space $R$ of $e$ is the set of iterations in which the value produced by $v$ is consumed by $w$ via $e$:

$$R[e] \leftarrow I[w] \cap (I[v] + d[e])$$

For each iteration $i$ in the condition space $I[v]$ of $v$, there is exactly one incoming edge $e_j \in E^+(v)$ per operand $j$ with $i \in R[e_j]$. Otherwise, we consider the RDG semantically incorrect.

Similarly, an operation node $v$ may have multiple outgoing edges that represent multiple consumers. An edge is “written” by node $v$ in all iterations $i$ that belong to the edge’s write space.

Definition 4.4 ([WHT19]). Let $e = (v, w)$. The write space $W$ of $e$ is the set of iterations in which the value consumed by $w$ is produced by $v$ via $e$:

$$W[e] \leftarrow I[v] \cap (I[w] - d[e])$$

For each iteration $i$ in the condition space $I[v]$ of $v$, there is at least one outgoing edge $e \in E^-(v)$ with $i \in W[e]$; otherwise, we consider the RDG semantically incorrect. If there is more than one such edge, they must be annotated with the same registers to result in a valid instruction during code generation (see Sections 4.9.5 and 5.2.2).

Example (FIR, continuing from p. 54). Consider edge $e_2 = (v_1, v_2)$ in Figure 3.3 with dependence vector $d[e_2] = (1, 1)^T = 1$. As per the above definitions:

$$R[e_2] \leftarrow I[v_2] \cap ((I[v_1] \cap I) + d[e_2])$$

$$\equiv \{i > 0 \wedge j > 0 \} \cap (\{j = 0 \wedge 0 \leq i < T\} + 1)$$

$$\equiv \{i > 0 \wedge j > 0 \} \cap \{0 < i \leq T \wedge j = 1\}$$

$$\equiv \{0 < i \leq T \wedge j = 1\}$$

$$W[e_2] \leftarrow I[v_1] \cap (I[v_2] - d[e_2])$$

$$\equiv \{j = 0\} \cap (\{0 < i < T \wedge 0 < j < N\} - 1)$$

$$\equiv \{j = 0\} \cap \{0 \leq i < T - 1 \wedge 0 \leq j < N - 1\}$$

$$\equiv \{0 \leq i < T - 1 \wedge j = 0\}$$

\[3\] The intersection with $I$ is necessary to correctly translate the bounds of $I'$. 

85
While operation node $v_1$ produces values onto edge $e_2$ in iterations $\{0 \leq i < T-1 \land j = 0\}$, operation node $v_2$ consumes them in iterations $\{0 < i \leq T \land j = 1\}$. Embedding these spaces into the tiled iteration space $I^*$ yields

\[
R^*[e_2] \leftarrow \{0 < p_1k_1 + j_1 \leq T \land p_2k_2 + j_2 = 1\}
\]

\[
W^*[e_2] \leftarrow \{0 \leq p_1k_1 + j_1 < T - 1 \land k_2 = 0 \land j_2 = 0\}
\]

Register allocation for TCPAs needs to consider not only allocating registers for intra-iteration communication (general-purpose registers), but also for inter-iteration communication (feedback registers) and inter-processor communication (input and output registers). Two aspects determine the type of register that is allocated for an edge: how long a value is alive, and whether inter-processor communication is required. Inter-processor communication is required for an edge $e$ if its annotated dependence vector $d[e]$ has at least one $\delta \neq 0$ in its set of processor displacements $\Delta(d[e])$. This is only the case for $d[e] \neq 0$. To make this more explicit in the RDG, we therefore split each edge $e$ with $d[e] \neq 0$ into $|\Delta(d[e])|$ new edges $e^i$, where $1 \leq i \leq |\Delta(d[e])|$. Each new edge has the same source, sink, and annotations as $e$, but each is additionally annotated with the corresponding processor displacement $\delta[e^i] \leftarrow \delta_i$. Furthermore, the read and write spaces of the original edge are partitioned to reflect the subsets of the tiled iteration space where the corresponding inter-processor communication takes place:

\[
R^*[e^i] \leftarrow R^*[e] \cap D_{d[e],\delta_i}^{s,in}
\]

\[
W^*[e^i] \leftarrow W^*[e] \cap D_{d[e],\delta_i}^{s,out}
\]

where $D_{d[e],\delta_i}^{s,in}$ and $D_{d[e],\delta_i}^{s,out}$ are the input and output domains associated with the processor displacements, as introduced in Section 3.4.3.

**Example** (FIR, continuing from p. 85). Edge $e_2 = (v_1, v_2)$, for which we determined the read and write space in the previous example, is annotated with dependence vector $d[e_2] = (1, 1)$. Assuming the space mapping $\phi$ is two-dimensional, this dependence vector results in four processor displacements:

\[
\Delta\left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) = \{\delta_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \delta_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \delta_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \delta_4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}\}
\]

The edge is therefore split into four edges $e_2^1, e_2^2, e_2^3,$ and $e_2^4$, each annotated with the corresponding processor displacement. After splitting, the edge $e_2^1$ represents intra-processor, but inter-iteration communication because $\delta_1 = 0$ and $d \neq 0$. This is apparent in its newly annotated read and write spaces:

\[
R^*[e_2^1] \leftarrow \{j_1 > 0 \land j_2 = 1\}
\]

\[
W^*[e_2^1] \leftarrow \{0 \leq p_1k_2 + j_1 < T - 1 \land j_1 < p_1 - 1 \land j_2 = 0 \land k_2 = 0\}
\]
By contrast, the edge \( e_4^2 \) represents inter-processor communication from the current PE to the PE below right of it because \( \delta_4 = (1, 1)^T \). This is reflected by the read space describing the “top-left” corner and the write space describing the “bottom-right” corner of the intra-iterationspace:

\[
\mathcal{R}^*[e_4^2] \leftarrow \{ j_1 = 0 \land k_1 > 0 \land j_2 = 0 \land k_2 = 1 \land p_2 = 1 \} \\
\mathcal{W}^*[e_4^2] \leftarrow \{ j_1 = p_1 - 1 \land k_1 < t_1 - 1 \land j_2 = 0 \land k_2 = 0 \land p_2 = 1 \}
\]

After splitting, one or more registers are allocated for each edge \( e = (v, w) \in E \) and annotated as \( \text{reg}^{\text{read}} \), which represents the register used for reading the corresponding operand value of \( w \), and as \( \text{reg}^{\text{write}} \), which represents the register for writing the result of \( v \). Most often, \( \text{reg}^{\text{read}} = \text{reg}^{\text{write}} \), in which case we combine both into \( \text{reg} \). The registers are allocated according to the following classification: If \( e \) is a dependence edge, that is if both \( v \) and \( w \) are operation nodes [Han09]:

- If \( d[e] = 0 \), the edge value is alive for a single iteration and a \( k \)-tuple of general-purpose registers \( \text{reg}[e] \leftarrow (r_1, r_2, \ldots, r_k) \) with \( r_i \in R_{rd} \) is allocated. Multiple registers (\( k > 1 \)) are necessary if the lifetime (see below) of the edge’s value is larger than the initiation interval \( \pi \). We collect these edges in the set \( E_{rd} \).

- If \( d[e] \neq 0 \) and \( \delta[e] = 0 \), the edge value is alive multiple iterations, but only within the current PE. A feedback register \( \text{reg}[e] \leftarrow r \in R_{id} \) of depth \( \lambda d[e]/\pi \) is allocated to keep the value alive for the corresponding number of iterations. We collect these edges in the set \( E_{jd} \).

- If \( d[e] \neq 0 \) and \( \delta[e] \neq 0 \), the edge value is alive across two PEs and requires communication between them. Two registers are allocated: an input register \( \text{reg}^{\text{read}}[e] \leftarrow r \in R_{id} \) for reading on the sink processor \( \phi(k) \) and an output register \( \text{reg}^{\text{write}}[e] \leftarrow r \in R_{od} \) for writing on the source processor \( \phi(k) - \delta[e] \). Additionally, a route from \( \phi(k) - \delta[e] \) to \( \phi(k) \) on the interconnect network is allocated (see Section 4.9.4). We collect these edges into both sets \( E_{id} \) and \( E_{od} \).

Otherwise, if \( e \) is an input edge, an input register \( \text{reg}[e] \leftarrow r \in R_{id} \) is allocated, and the edge is put into \( E_{id} \). If \( e \) is an output edge, an output register \( \text{reg}[e] \leftarrow r \in R_{od} \) is allocated, and the edge is put into \( E_{od} \). For other edge types, no register is allocated.\(^4\)

The conventional approach to register allocation—vertex coloring of an interference graph [Cha82]—applies here, for which several optimal and heuristic solving methods exist. An interference graph is a graph that has a node for each entity that requires a register and an edge between any two nodes whose entities interfere with each other; that is, they cannot share the same register. Finding a \( k \) vertex coloring of the interference graph

\(^4\)Some use-cases are worthy investigating in the future; for example, a constant edge might be assigned a register into which the value is preloaded upon run-time configuration. This solves problems with maximum immediate values in instructions.
graph then corresponds to an allocation of \( k \) registers. Depending on the register type, *interference* is defined differently, as explained in more detail in the following sections. However, for each register type—general-purpose (\( E_{rd} \)), feedback (\( E_{fd} \)), input (\( E_{id} \)), and output (\( E_{od} \))—, the compiler performs the following three steps:

1. **Clustering**: Partition \( E_{rd} \) into edge clusters \( E_i \subseteq E_{rd} \) such that each \( E_i \) contains edges that require the same register, for example due to semantic requirements for code generation. This partitioning is different for each register type. Each cluster \( E_i \) is a node in the interference graph.

2. **Interference checking**: Insert an edge between any two clusters \( E_i \) and \( E_j \) that interfere according to a register-specific interference relation \( E_i \overset{rd}{\leftrightarrow} E_j \).

3. **Coloring**: Find a \(|R_{rd}|\) vertex coloring of the interference graph.\(^5\)

In the context of symbolic modulo scheduling, these three steps for register allocation may either be encoded within the resource constraints \( Res \) or performed a posteriori after scheduling. In the latter case, if register allocation fails, symbolic modulo scheduling is repeated with an increasing initiation interval until successful, or a threshold is reached and considered failed. In either case, if successful, the resulting allocated registers are annotated onto the edges and symbolic compilation proceeds (see Chapter 5).

The following sections formalize clustering and interference relations for each of the register types; we refer to Section 5.4 for a full example.

### 4.9.1 Allocation of general-purpose registers

The set of edges \( E_{rd} \) is partitioned into clusters \( E_i \) with the following properties: (1) All edges in \( E_i \) have the same source node \( v_{E_i} \), and (2) the write space of each edge \( e \) in \( E_i \) intersects the write space of at least another distinct edge \( e' \neq e \) in \( E_i \). Property (1) indirectly states that if two edges have different source nodes, they are not required to have the same register allocated because both source nodes may result in different assembly instructions. Property (2) ensures that all edges of an edge \( e \) are in the same cluster if they require the same register for semantic correctness. Otherwise, an ill-formed instruction would be generated during code generation (see Section 5.2.2).

**Example.** Assume an operation node \( v \) has two outgoing edges \( e_1 \) and \( e_2 \), one to \( w_1 \) and one to \( w_2 \), both with \( d = 0 \) annotated. Intuitively, this means that values produced by \( v \) are consumed by both \( w_1 \) and \( w_2 \). The operation node \( v \) corresponds to a single assembly instruction (see Section 5.2.2) with a single destination register.\(^6\) First, assume that \( \mathcal{W}^* [e_1] \cap \mathcal{W}^* [e_2] = \emptyset \). There is no conflict: if \( i^* \in \mathcal{W}^* [e_1] \), then execute one

---

\(^5\)Finding a minimal coloring is optional, but potentially reduces code size.

\(^6\)The presented compile flow can easily be extended to operations and instructions with multiple results, but we only consider operations with one result in this thesis.
assembly instruction, if $i^* \in W^*[e_2]$, then execute another assembly instruction, which may have a different destination register. However, if $W^*[e_1] \cap W^*[e_2] \neq \emptyset$, the arising conflict within the intersection of both write spaces can be avoided by assigning both edges the same register $r$, meaning the same assembly instruction can be executed. \[ \triangle \]

For each cluster $E_i$, the operation of its associated source node $v_{E_i}$ produces a value at time step

$$t_i = \tau[v_{E_i}] + l[v_{E_i}]$$

in each iteration $i^* \in W^*_{E_i}$, where $W^*_{E_i}$ is the union of the write spaces of all edges in $E_i$. The value is alive until the last sink operation consumes it, which is at time step

$$t'_{E_i} = \max_{(v,w) \in E_i} \tau[w].$$

Consequently, each produced value has a live interval of $l_{E_i} = [t_{E_i}, t'_{E_i}]$. Two edge clusters $E_i$ and $E_j$ therefore interfere if

$$E_i \leftrightarrow E_j \iff W^*_{E_i} \cap W^*_{E_j} \neq \emptyset \land l_i \cap l_j \neq \emptyset.$$

If the lifetime $Life_i = t' - t$, that is the number of time steps between production and consumption, is larger than the initiation interval $\pi$, then $k_i = \lceil Life_i / \pi \rceil$ registers are necessary. This is because after $\pi$ time steps, the produced value is still alive since $Life_i > k_i$, but a new value is produced that must also be stored. However, after $k_i$ iterations, the first register is free again.

### 4.9.2 Allocation of feedback registers

Each edge $e$ in $E_{fd}$ represents a recurring loop-carried dependence, meaning the produced value stays alive for $\lambda/d[e]/\pi$ iterations. Since a new value is produced each iteration, the feedback register to be allocated must have a depth of $\lambda/d[e]/\pi$ to ensure there are enough slots and the produced value reappears after the correct number of rotations. Since the depth of a feedback register is configured prior to and considered static during execution, two edges may only share a feedback register if they result in the same depth. Consequently, $E_{fd}$ is partitioned into clusters $E_i$ with the following properties: (1) All edges in $E_i$ have the same source node $v_{E_i}$, (2) the write space of each edge $e$ in $E_i$ intersects the write space of at least another distinct edge $e' \neq e$ in $E_i$, and (3) all edges in $E_i$ have the same dependence vector $d_{E_i}$. Property (3) in particular ensures that all edges in $E_i$ result in the same feedback register depth.

We consider loop-carried dependences to be alive for the entirety of the $\lambda/d_{E_i}/\pi$ iterations. Accordingly, two edge clusters $E_i$ and $E_j$ interfere if:

$$E_i \leftrightarrow E_j \iff W^*_{E_i} \cap W^*_{E_j} \neq \emptyset \lor d_{E_i} \neq d_{E_j}.$$
4 Symbolic Modulo Scheduling

Note that two distinct dependence vectors may result in the same depth depending on the schedule, meaning the above formulation for interference is pessimistic (that is, it may consider two clusters as interfering even if they could share the same feedback register). However, using the exact condition \( \lambda^i\mathbf{d}_{E_i} \neq \lambda^j\mathbf{d}_{E_j} \) may result in multiple cases because of the parameters in \( \lambda^i \), which complicates the subsequent vertex coloring step.

**Example** (Matrix multiplication, continuing from p. 76). Refer to Figure 3.4. Edges \((v_1, v_2)\) and \((v_2, v_2)\) may share the same feedback register because they have disjoint write spaces and the same dependence vector \( \mathbf{d} = (0, 1, 0)^T \) annotated. By contrast, edges \((v_1, v_2)\) and \((v_3, v_5)\) may not share the same feedback register because they have differing dependence vectors annotated and therefore require different depths. △

### 4.9.3 Allocation of input and output registers

Two cases require the allocation of input and output registers: propagation of uniform dependences across PEs, which corresponds to dependence edges \( e \) with \( \delta[e] \neq 0 \), and streaming data to and from the I/O buffers, which corresponds to output and input edges of the RDG. Register allocation for the former is explained in the next section, which includes the allocation of a propagation channel on the interconnect network.

This leaves register allocation for the input edges \( E_{id} \cap E^{in} \) and the output edges \( E_{od} \cap E^{out} \). Both register types are allocated separately; let us start with the input edges: Each edge \( e \) is its own cluster \( E_i \) because input registers are tied to interconnect ports—even if the register itself could be shared, the interconnect route most likely can not. Moreover, values read from an input buffer are only considered alive for the single iteration they are read in. We therefore define

\[
E_i^{id} \leftrightarrow E_j \iff \mathcal{R}_i^* \cap \mathcal{R}_j^* \neq \emptyset.
\]

In other words, two input registers only interfere if they are read in the same iteration at least once.

To account for the input registers already allocated during propagation channel routing (see next section), the interference graph is constructed for all edges in \( E_{id} \), not only the subset of input edges. Then, each cluster \( E_i \) that represents a dependence edge is pre-colored according to the result of propagation channel routing before performing vertex coloring.

The output edges are allocated analogously using

\[
E_i^{od} \leftrightarrow E_j \iff \mathcal{W}_i^* \cap \mathcal{W}_j^* \neq \emptyset.
\]

### 4.9.4 Routing of propagation channels

In addition to a pair of registers, the communication induced by a dependence edge \( e \) with a processor displacement \( \delta[e] \neq 0 \) requires an interconnect route from an output
4.9 Register allocation

port of the source PE $\phi(k)$ to an input port of the sink PE $\phi(k) + \delta[e]$ for each PE $\phi k$ in the edge’s write space $W^*[e]$. However, with symbolic compilation, the subset of PEs that $W[e]$ will get mapped at run time is unknown at compile time. To still allow compile-time routing, we assume the interconnect network to be homogeneous, that is, all interconnect wrappers to have the same ports and adjacencies. Then, for each dependence edge $e \in E_{id} \cap E_{od}$, a template route is allocated that is replicated across all affected PE pairs during instantiation (see Section 6.1). Since these replicated routes form dedicated communication channels between equidistant pairs of PE in order to propagate data across the TCPA, we call the template route a propagation channel.

**Definition 4.5 ([WWH+]).** A reduced topology graph $T$ is a directed graph that represents the topology of a homogeneous interconnect network in terms of a common interconnect wrapper architecture. The graph contains a node $v$ for each port in the common interconnect wrapper architecture and an edge for each adjacency between two ports, weighted with their inter-processor distance. In particular, there is an edge weighted $(0 \ 0)^T$ from each source port to each of its adjacent sink ports. Additionally, there is an edge from each directional sink port $\location_i^<= \location_j^>$ to its sibling port weighted by $\location: (1 \ 0)^T$ for east, $(0 \ 1)^T$ for south, $(-1 \ 0)^T$ for west, and $(0 \ -1)^T$ for north.

For convenience, we assume $T$ has two polar nodes: Source, connected to all PE output port nodes, and Sink, connected to all PE input port nodes. A propagation channel $\rho$ for a dependence edge $e$ is then a path $(v_1, v_2, \ldots, v_{|e|})$ on $T$ from Source to Sink where the sum of weights equals $\delta[e]$. Routing $k = |\{e \in E: \delta[e] \neq 0\}|$ propagation channels is thus equivalent to solving the $k$ node-disjoint exact-length paths problem, with the relaxation that the paths of two propagation channels $\rho_1$ and $\rho_2$ may share the first $1 \leq r \leq \min(|\rho_1|, |\rho_2|)$ nodes if the write spaces of the corresponding RDG edges are disjoint. This is because if the write spaces are disjoint, the shared nodes (ports) are never occupied at the same time. However, sharing any node (port) but not its predecessors would imply that the port has two connected sources, which is not possible.

The resulting routes and registers—the first node in each $\rho$ corresponds to a PE output register, the last node to a PE input register—are annotated to the corresponding edges in the RDG.

**Example.** Assume an RDG has two edges $e_1$ and $e_2$ with $\delta[e_1] = (0, 1)^T$ and $\delta[e_2] = (1, 1)^T$, that is, two propagation channels need to be allocated. Also assume that $W^*[e_1] \cap W^*[e_2] = \emptyset$. Figure 4.5 depicts the reduced topology graph of an example interconnect architecture. A corresponding allocation of two propagation channels is highlighted with colors. Because the write spaces are disjoint, these two propagation channels may share the first part of the same route. With this allocation, $e_1$ is assigned $r_{\text{read}}[e_1] \leftarrow id0$ and $r_{\text{write}}[e_1] \leftarrow od0$, while $e_2$ is assigned $r_{\text{read}}[e_1] \leftarrow id1$ and $r_{\text{write}}[e_1] \leftarrow od0$. \triangle
4 Symbolic Modulo Scheduling

4.9.5 Allocation of virtual registers

It is frequently the case that at least two outgoing edges of the same operation node have intersecting write spaces but require different registers because they interfere. As stated multiple times above, this case is a semantic error because the resulting assembly instruction would be ambiguous. Such RDGs therefore require to be transformed, for example, by duplicating the offending operation node and scheduling it twice, thus circumventing the problem.

Another solution is specific to TCPAs: using virtual registers. A virtual register is configured at run time with a set of register addresses and, when a value is written to it, writes the value to all registers in the configured set simultaneously. By allocating and assigning conflicting edges the same virtual register for writing, but leaving the original register for reading, a semantically correct RDG is obtained (compare Figure 4.6). The allocated virtual registers need to be configured accordingly. Note that an optimal allocation of virtual registers is considered future work.

4.10 Experiments and discussion

While we proved our method formally, the following experiments aim to illustrate and summarize the presented scientific achievements: For UDAs, symbolic modulo scheduling is often optimal even for an unknown number of tiles and the minimal tile sizes are negligibly small. These aspects are illustrated in Table 4.1.
### Table 4.1: Summary of applying our symbolic modulo scheduling approach to a variety of algorithms

(\(\pi = 1\) if not explicitly shown)

<table>
<thead>
<tr>
<th></th>
<th>Edge, (\pi = 1)</th>
<th>Edge, (\pi = 4)</th>
<th>Euler</th>
<th>FIR</th>
<th>Harris</th>
<th>Elliptic</th>
<th>Mat-Mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimension (n)</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Optimal if number of tiles known</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Average slowdown if unknown (%)</td>
<td>0</td>
<td>0.002</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Largest minimal tile sizes (p^{\text{min}})</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>((3, 1)^T)</td>
<td>None</td>
</tr>
</tbody>
</table>
Figure 4.6: On the left, the outgoing edges of the node \( v \) have different registers, but are assumed to have the same write space. On the right, the two edges have been annotated with the same virtual register \( vd0 \) for writing, but the original register for reading. Since both edges now "write" to the same register, the semantic error has been resolved. (Note that \( vd0 \) needs to be configured at run time to write to \( fd0 \) and \( rd0 \).)

- Image processing: edge detection as presented in this chapter ("Edge," for two resource configurations), Harris corner detection ("Harris")
- Signal processing: finite response filter ("FIR") as presented in the previous chapter
- Linear algebra: matrix multiplication ("MatMul") as presented in the previous chapter
- General mathematics: Euler method ("Euler"), elliptic digital wave filter ("Elliptic").

These examples are one-, two- or three-dimensional loops. As of writing this dissertation, our work is still the only one capable of performing software pipelining on multi-dimensional loops symbolically.

**Optimal for a known number of tiles**

For each of the columns listed in Table 4.1, we exhaustively compared our symbolic modulo schedules for one space mapping to static, known-to-be latency-minimal schedules for TCPA sizes from \( \hat{t} = (2, 2) \) to \( \hat{t} = (16, 16) \). The iteration space size was kept constant for each experiment. This experiment confirms our proof: All 253 symbolic schedules (one for each \( \hat{t} \)) are latency-minimal if the number of available PEs at run time \( t' \) matches the number \( t \) used during the optimization.
4.11 Related work

Nearly optimal for an unknown number of tiles

Next, for each of the 253 symbolic schedules, we compared the latencies of the 252 cases where \( t \neq t' \) with the statically determined minimum latency. The table shows the arithmetic mean of the latency slowdown in percent, that is, of \( 100 \cdot \frac{L}{L_{\text{opt}}} - 100 \). As can be seen, in all but one experiment, the schedules were latency-minimal for all cases even if the processor array size differed from the one using during optimization. This is because the inter-tile offsets \( \varepsilon \) were optimized to 0, meaning that the number \( t \) of tiles had no impact on the minimum of \( L \) (see Section 4.4).

Negligibly small minimal tile sizes

In the last row of each column, the table shows the largest (in terms of summing all elements) minimal tile size from the 253 static schedules. Only the Harris corner detection resulted in minimal tile size \( p_{\text{min}} = (3, 1)^T \), all others are feasible for any tile size. This can be attributed to all operations having small execution latencies, which is realistic because programmable PEs in TCPAs do in general not support complex instructions. The smaller the latency of an operation, the less impact it has on the schedule because it can be satisfied even if there is little time between production and consumption.

4.11 Related work

In previous pioneering work, we investigated the symbolic parallelization of loop nests, but did not consider software pipelining [TTH13; TWT+14; TWT+15] yet. This means iterations were assumed atomic and enough PE resources available such that they can be initiated in unit time. While the above works culminate in a more general scheduling method (for example, in addition to \textit{locally sequential}, \textit{globally parallel}, also \textit{locally parallel}, \textit{globally sequential} is supported), software pipelining is crucial to optimize the throughput and latency of a loop while taking restricted resources inside PEs to execute the loop body into account. Furthermore, resource constraints were not explicitly considered, including the local allocation of functional units and other resources, which is necessary for finally generating real PE programs. This lead to the development of symbolic modulo scheduling as presented this chapter, which focuses on LSGP, but could be generalized to LGPS.

As for other related work on affinely scheduling parameterized loop nests, in [DR95], Darte et al. study the asymptotic behavior of resulting schedules without tiling. Later, they also formulated closed-form expressions that describe all tight intra-tile schedules for rectangularly tiled loop programs assuming LSGP partitioning\(^7\) [DSR+02]. Their presented method of parallelizing is similar to way symbolic modulo scheduling parallelizes, but does neither take into account how to schedule inter-tile dependencies, nor

\(^7\)We use this fact in Lemma 4.1.
Symbolic Modulo Scheduling

does it explain how software pipelining can be incorporated across tiles. Moreover, no
criteria was so far known for finding the optimum among the closed-form schedules.

Approaches that consider UDAs with resource constraints on a processor array start
with [Thi95] by Thiele et al. While Thiele finds optimal local and global schedules
according to an integer linear program that takes the resource constraints into account,
the resulting schedules require a number of PEs that depends on the loop bounds.
By contrast, Teich et al. [TTZ96; TTZ97] used the LSGP allocation scheme, where, as
previously stated, the number of PEs does not depend on the loop bounds. However,
one of these early approaches was symbolic.

Symbolic (or parametric) tiling is the subject of several other works, all targeting
shared-memory CPUs in a desire to accommodate multi-level memory hierarchies. The
first of these is PrimeTile by Hartono et al. [HBB+09], which generates sequential tiled
code from an imperfectly nested loop nest. Generating parallelized code is much more
involved because tiling may lead to non-affine expressions in the loop bounds, making it
difficult to extract parallelism in a generalized manner [THB+10]. Both Renganarayanan
et al. [RKS’12] and Kim et al. [KR09] (D-Tiling) try to solve this problem by determining
the outset of the iteration space, which is the set of tile origins such that the tiles cover
the entire iteration space of a given loop. Similar to LSGP, one set of nested loops iterates
over the tiles described by the outset, and another set of nested loops iterates over the
iterations within the corresponding tile. Whereas Renganarayanan et al. use the polyhe-
dron model to calculate the outset, Kim et al. rely solely on syntactic transformations.
Both approaches aim for cache coherency and coarse-grained parallelism, but do not
describe scheduling of the tiles, making them unfit for architectures such as TCPAs.

DynTile by Hartono et al. [HBR+10] generates a prolog and epilog for non-full tiles and,
at run time, dynamically schedules tiles in parallel across a wavefront. This softwarc-
controlled, dynamic way of scheduling, together with the presumed atomicity of tile
execution, makes it unsuited for TCPAs as well.

None of the above approaches consider software pipelining of the loop to be par-
allelized; the most frequent realization of software pipelining is modulo scheduling
[RG81]. The majority of modulo scheduling methods only pipeline the innermost level
of a loop, and instead rely on loop permutation to maximize instruction-level parallel-
ism. Symbolic modulo scheduling generalizes this by using affine schedules (recall
that the intra-tile schedules are permutations of each other, making them similar to loop
permutation).

A popular method for applying modulo scheduling to multi-dimensional (nested)
loops was developed by Rong et al. [RTG’07]. They identify the most profitable loop
level according to given optimization criteria and construct a one-dimensional modulo
schedule from the multi-dimensional data dependence graph. This is similar to how
symbolic modulo scheduling handles the intra-tile level—the tight intra-schedules
basically flatten the multi-dimensional tiles onto one dimension—but their approach
considers neither the inter-tile level nor parametric loop bounds.

For more details on specific modulo scheduling methods, we refer to the survey by
Codina et al. [CLG02] and Hannig’s dissertation [Han09], which also situates modulo scheduling within the larger field of scheduling theory. Hannig’s work in particular is relevant for symbolic modulo scheduling because it presents an integer linear program solution to the resource-constrained modulo scheduling problem for ADAs using affine scheduling and LSGP in the context of high-level synthesis—making it very similar to the goal of symbolic modulo scheduling. However, he assumes both the number of PEs to be synthesized and the loop bounds to be fixed a priori. This has the advantage that the synthesized processor array is always optimal with regards to the integer linear program, as opposed to symbolic modulo scheduling, which may be slightly worse than the latency minimum if the number of processing elements is unknown at compile time. However, the flexibility granted by symbolic modulo scheduling is far more important in the context of TCPAs than the negligible loss of optimality. Also, the method is restricted to ILP solvers, whereas symbolic modulo scheduling, or rather the reduction theorem in particular, works with any modulo scheduling approach. This would also allow heuristic approaches, such as iterative modulo scheduling, to be used, which is often more practical for the production use of a compiler.

In summary, as of writing this thesis and to the best of our knowledge, symbolic modulo scheduling is the only method for modulo scheduling of multi-dimensional, parametric loops using symbolic tiling.
Symbolic Mapping

The previous chapter introduced techniques for the resource-constrained modulo scheduling of loop nests given as UDAs symbolically at compile time, thus without the need to know the loop bounds or number of available PEs. At runtime, the resulting symbolic schedule, functional unit binding, register allocation, and propagation channel routes are materialized into concrete programs and configuration data according to the then-known concrete parameter values, a phase we call instantiation (Chapter 6). However, there is still a key puzzle piece missing between symbolic modulo scheduling and instantiation: a suitable representation that describes the above artifacts obtained by symbolic modulo scheduling and from which PE programs and the concrete configuration data of the peripheral components can then easily and efficiently be instantiated. We call this puzzle piece a symbolic configuration and its generation as a whole symbolic mapping. Particularly important is representing the PE programs symbolically, that is, parameterized on the loop bounds and number of PEs. For that, this chapter introduces polyhedral syntax trees, a parameterized generalization of (abstract) syntax trees whose nodes are annotated with polyhedral condition spaces.

Roadmap. First, Section 5.1 describes symbolic configurations and their constituents. One of these is a polyhedral syntax tree, a data structure whose motivation, definition, and generation from an annotated RDG are elaborated in Section 5.2. The last constituent not yet mentioned is a set of access mappings, which map registers to input and output variables and will be introduced in Section 5.3. Finally, Section 5.4 summarizes the steps and full procedure of symbolic mapping.

5.1 Symbolic configuration

Symbolic modulo scheduling (Chapter 4) performs space-time mapping on the RDG of a given UDA; the results are then annotated to the nodes and edges. However, an RDG is not a very suitable representation to directly generate a concrete TCPA configuration from because it does not reflect the component hierarchy of a TCPA well. For example, where and how in an RDG is the information represented that is required to generate I/O buffer configuration data? Instead, we define a data structure called a symbolic configuration that collects all data in a form more amenable to the instantiation of a
TCPA configuration at run time. Its constituents are:

- The space mapping \( \phi \),
- the symbolic schedule vector \( \lambda^* = (\lambda^i, \lambda^k) \),
- a polyhedral syntax tree derived from the given RDG and its annotations (Section 5.2) that represents the PE programs to be generated later,
- a set of propagation routes (Section 4.9.4) from which a concrete interconnect configuration will be generated,
- a set of access mappings (Section 5.3) from which concrete address generator and I/O buffer configurations will be generated, and
- miscellaneous configuration data, such as for virtual registers.

While the space mapping, symbolic schedule vector, propagation routes, and miscellaneous configuration data are simply rearranged from the RDG, both the polyhedral syntax tree and set of access mappings require further work because the necessary information is only implicitly given by an RDG. The following sections elaborate on both, starting with polyhedral syntax trees.

### 5.2 Polyhedral syntax trees

Given an RDG, how can we transform the annotations pertaining to the symbolic space-time mapping into a symbolic representation that facilitates efficient run-time generation of PE programs? In particular, given concrete values for the parameters and a tiled iteration \( \mathbf{i}^* \), the instructions issued in that iteration should easily be derived from it. Furthermore, the representation should encode not only the instructions themselves, but also their order (from scheduling), place of execution (from functional unit allocation), and allocated registers as annotated in the given RDG. These properties closely mirror concrete PE programs (see Section 2.2), making the syntactic representation of a PE program an excellent base to parameterize on the iteration vector \( \mathbf{i}^* \).

#### 5.2.1 Definition

Syntactically, PE programs can be viewed as a hierarchy of fragments.

**Definition 5.1 ([WHT19]).** A fragment \( F \) is any syntactic constituent of a program.

For example, the assembly instruction `addi rd0 rd1 10` can be structured into five meaningful fragments: the mnemonic `addi`, the registers `rd0` and `rd1`, the integer literal `10`, and finally, the entire instruction itself. As these examples imply, a fragment
5.2 Polyhedral syntax trees

usually consists of other fragments, by nature of being a syntactic representation of a program—they can therefore be likened to the nodes in a syntax tree.

Since we desire a symbolic representation, we parameterize fragments on an iteration vector $i$.

**Definition 5.2 ([WHT19]).** A polyhedral fragment $f(i)$ maps an iteration $i \in I$ to a fragment $F$.

For example, assume an instruction writes its result to general-purpose register $rd0$ to propagate it internally within the PE, but to output register $od0$ to communicate it to another PE at the border of the PE’s one-dimensional iteration space:

$$
dest(j = (j)) := \begin{cases} 
    rd0 & \text{if } j < 15 \\
    od0 & \text{if } j = 15 
\end{cases}
$$

From this, we can hierarchically model the instruction itself. For example, assume it simply copies the value of register $rd1$ to the destination register described by $dest$:

$$
imstr(j) = \text{move} \ dest(j) \ rd1
$$

Or, more intuitively, as an actual tree with conditions annotated to the leaves in brackets:

$$
imstr := \text{move} \quad \text{dest} \ \{j < 15\} \quad \text{rd0} \ \{\{j < 15\}\} \\
\quad \text{od0} \ \{j = 15\} \quad \text{src} \ \{\{\text{always}\}\} \quad \text{rd1}
$$

We call this representation a polyhedral syntax tree due to its similarity to abstract syntax trees. In fact, the evaluation of such a polyhedral syntax tree at an iteration $i$ as defined below yields a concrete syntax tree—a polyhedral syntax tree may therefore also be seen as a forest of concrete syntax trees.

**Definition 5.3** (Adapted from [WHT19]). A polyhedral syntax tree is a triple $f = (I, a, G)$ of a condition space $I$, a tuple $a$ of attributes, and a set of children $G$. To avoid ambiguity, we write domain($f$) for $I$, attr($f$) for $a$, and children($f$) for $G$ of $f$. Each child node $g \in G$ is again a polyhedral syntax tree; for clarity, we use the term “node” when focusing on a single node and “polyhedral syntax tree” when emphasizing the tree structure as a whole.

Each node $f$ is of one of two types: If attr($f$) = ($F$), that is, if the tuple $a$ only contains a fragment $F$, then $f$ is a fragment node. Otherwise, $f$ is a meta node that stores implementation-specific syntactic meta-information in its attributes.

All nodes $f$ satisfy the following properties regarding their immediate children:

- All children of $f$ are of the same type, that is, either all are fragment nodes or all are meta nodes. This is independent of the type of $f$ itself.
If the children of $f$ are fragment nodes, their condition spaces must be disjoint:

$$g_1, g_2 \in \text{children}(f), g_1 \neq g_2 \implies \text{domain}(g_1) \cap \text{domain}(g_2) = \emptyset$$

If $f$ has children, its condition space is the union of its children’s condition spaces:

$$\text{domain}(f) = \bigcup_{g \in \text{children}(f)} \text{domain}(g)$$

No two children may have the same attribute values:

$$g_1, g_2 \in \text{children}(f), g_1 \neq g_2 \implies \text{attr}(g_1) \neq \text{attr}(g_2)$$

The evaluation $f(i)$ of a polyhedral syntax tree is the sub-tree where all nodes $g$ with $i \notin \text{domain}(g)$ are removed.

Fragment nodes represent, similarly to terminals in a context-free grammar, the concrete constituents the program is made of; meta nodes, similarly to non-terminals, the idiosyncrasies of the concrete program syntax. To represent PE programs in the desired fashion—that is, such that the polyhedral syntax tree represents the instructions issued in a given iteration, together with scheduling and allocation information—, we define the following kinds of meta and fragment nodes:

1. The root is a meta node $f$ with empty attributes $\text{attr}(f) = ()$.
2. The immediate children of the root node are meta nodes that represent individual functional unit programs. They each have only one attribute: the identifier of the corresponding functional unit $fu$, such as add0. We call these nodes polyhedral functional unit programs.
3. The immediate children of a polyhedral functional unit program are meta nodes and represent a relative time slot of execution. Each has one attribute: the time offset $\tau$. We call these nodes polyhedral time slots.
4. The immediate children of a polyhedral time slot are fragment nodes that represent the mnemonic of the instruction (e.g. add or move). We call these nodes polyhedral instructions.
5. The immediate children of a polyhedral instruction are meta nodes that represent its operands, each with its position $pos$ as attribute. For clarity, we label operand positions with intuitive names, such as dest or src. We call these nodes polyhedral operands.
6. Finally, the immediate children of a polyhedral operand are fragment nodes that represent registers and immediate values, such as rd0 or 12.
Because, using these node kinds, the evaluation of a polyhedral syntax tree at an iteration $i^*$ results in a syntax tree that represents the instructions issued in that iteration, concatenating the instruction sequences in execution order for all $j$ in the intra-tile space $J$ for a PE $k$ yields an unrolled assembly program for that PE. These observations serve as the basis for an efficient algorithm for compact PE program generation in Section 6.2.

**Example.** The following simple polyhedral syntax tree illustrates our conventions for displaying them. We parameterize them on the tiled iteration vector $i^*$ because there may be different polyhedral instructions depending on both the intra- and inter-tile iteration. Each node is represented by its attributes, but fragment nodes are typeset in typewriter font, meta nodes in italic. The leaves are additionally annotated with their condition space in square brackets:\footnote{1}{All other condition spaces are unions thereof, making their annotation redundant.}

$$f(i^* = (j, k)^T) := \begin{array}{c}
\text{dpu0} \quad \tau : 0 \quad \text{move} \quad \text{dest} \quad \text{rd0} \quad \{\{j < p - 1\}\}
\end{array}$$

$$\begin{array}{c}
\text{od0} \quad \{\{j = p - 1\}\}
\end{array}$$

$$\begin{array}{c}
\text{src} \quad \text{rd1} \quad \{\{\text{always}\}\}
\end{array}$$

This particular polyhedral syntax tree represents two instructions (move rd0 rd1 and move od0 rd1), both executed at time offset $\tau = 0$ (relative to the start of iteration $i^*$) on functional unit dpu0. Which of the two instructions is issued depends on the iteration vector $i^*$. The tree is constructed such that for intra-tile iterations at the border of $J$ ($j = p - 1$), the move instruction propagates the value rd1 to the next PE via od0, but for all other intra-tile iterations ($j < p - 1$), the value is propagated internally via rd0.

Let us demonstrate this symbolic nature with an example. To obtain a concrete syntax tree that represents a concrete program, $f$ is evaluated at $i^*$, using, for example, the tile size $p = 4$ and $i^* = (3, 1)^T$:

$$f(i^* = (3, 1)^T) := \begin{array}{c}
\text{dpu0} \quad \tau : 0 \quad \text{move} \quad \text{dest} \quad \text{od0} \quad \{\{j = 3\}\}
\end{array}$$

$$\begin{array}{c}
\text{src} \quad \text{rd1} \quad \{\{\text{always}\}\}
\end{array}$$

After evaluation, $f$ represents the sequence of instructions issued at time step $\lambda^* i^*$, that is, in iteration $j = 3$ of processor $k = 1$. Because $j = 3$ is at the border of $J$, the value in register rd1 is written into od0, that is, communicated to the next PE. If, however, the tile size $p = 8$, the tree evaluates to a different instruction:

$$f(i^* = (3, 1)^T) := \begin{array}{c}
\text{dpu0} \quad \tau : 0 \quad \text{move} \quad \text{dest} \quad \text{rd0} \quad \{\{j < 7\}\}
\end{array}$$

$$\begin{array}{c}
\text{src} \quad \text{rd1} \quad \{\{\text{always}\}\}
\end{array}$$

In this case, $j = 3$ is not at the border of $J$, so the value of rd1 is propagated PE-internally. $\triangleq$
The following operations are necessary for the construction and manipulation of polyhedral syntax trees.

**Definition 5.4** ([WHT19]). Two polyhedral syntax trees $f$ and $g$ are (merge-)compatible, denoted $f \sim g$, if $\text{attr}(f) = \text{attr}(g)$.

**Corollary 5.1** ([WHT19]). Let $f_1$ and $f_2$ be polyhedral syntax trees. Then, for each child $g_1 \in \text{children}(f_1)$, there is at most one child $g_2 \in \text{children}(f_2)$ such that $g_1 \sim g_2$.

**Proof.** This follows directly from Equation (5.1): Because an attribute tuple is unique within the immediate children of a node, if there is a match, there is exactly one.

**Definition 5.5** (Adapted from [WHT19]). Let $f_1$ and $f_2$ be compatible polyhedral syntax trees. The merged polyhedral syntax tree is

$$(f_1 + f_2) := (\text{domain}(f_1) \cup \text{domain}(f_2), \text{attr}(f_1), M \cup \overline{M})$$

where $M$ is the set of all merged, compatible children

$$\{g_1 + g_2 \mid g_1 \in \text{children}(f_1), g_2 \in \text{children}(f_2) : g_1 \sim g_2\}.$$

$\overline{M}$ is the set of all incompatible children

$$\{g_1 \mid g_1 \in \text{children}(f_1) : \nexists g_2 \in \text{children}(f_2) : g_1 \sim g_2\} \cup \{g_2 \mid g_2 \in \text{children}(f_2) : \nexists g_1 \in \text{children}(f_1) : g_1 \sim g_2\}.$$

**Example.** Consider two polyhedral syntax trees $\text{reg}_1$ and $\text{reg}_2$ that both have a meta node $\text{SourceA}$ as their root:

$$\text{reg}_1 := \text{SourceA} \longrightarrow \text{rd0} \{\{i = 0\}\} \quad \text{rd1} \{\{i = 1\}\}$$

$$\text{reg}_2 := \text{SourceA} \longrightarrow \text{rd0} \{\{i = 2\}\} \quad \text{rd2} \{\{i = 3\}\}$$

The two trees are merge-compatible because

$$\text{attr}(\text{reg}_1) = \text{attr}(\text{reg}_2) \iff (\text{SourceA}) = (\text{SourceA}) \implies \text{reg}_1 \sim \text{reg}_2.$$
5.2 Polyhedral syntax trees

\[(\text{reg}_1 + \text{reg}_2) := \text{SourceA} \quad \begin{cases} \text{rd}_0 \{i = 0 \lor i = 2\} \\ \text{rd}_1 \{i = 1\} \\ \text{rd}_2 \{i = 3\} \end{cases}\]

Using the merge operation, we next define the operation \(\text{combine}(a, G)\) that merges all given polyhedral syntax trees \(g \in G\) under a common root node with attribute \(a\):

\[
\text{combine}(a, G) := \sum_{g \in G} (\text{domain}(g), a, \{g\}),
\]

where \(\sum\) represents the merging of multiple nodes.

5.2.2 Generation

Generation of a polyhedral syntax tree from a symbolically modulo-scheduled RDG assumes the following prerequisites from symbolic modulo scheduling: Each operation node \(v \in V^{\text{op}}\) is annotated with its start offset \(\tau[v]\), the selected instruction template \(\text{instr}[v]\) that realizes the node’s operation, and the functional unit \(\text{fu}[v]\) that executes the selected instruction. Each dependence, input, and output edge \(e\) is annotated with its allocated registers \(\text{reg}^{\text{read}}[e]\) and \(\text{reg}^{\text{write}}[e]\). Using these annotations, the generation of a corresponding polyhedral syntax tree is a two-step process: First, a polyhedral instruction is generated for each operation node and its incoming and outgoing edges. These polyhedral instructions are then merged into a single polyhedral syntax tree.

**Generation of polyhedral instructions**

The structure of the polyhedral instruction to be generated for an operation node \(v\) is given by the annotated instruction template \(\text{instr}[v]\). In the following, we assume instruction templates to have the form

\[
\text{mnemo} \quad rd \quad rs_0 \quad rs_1 \quad \ldots,
\]

where \(rd\) represents the destination register and \(rs_j\) the \(j\)-th operand of the realized operation. The generated polyhedral instruction then has the structure

\[
\hat{f} := \text{mnemo} \quad rd \quad rs_0 \quad rs_1 \quad \ldots
\]

Consequently, the first step of polyhedral instruction generation is to determine all polyhedral operands \(rd, rs_0, \ldots\), from the incoming and outgoing edges of \(v\).

Each outgoing edge \(e\) represents a possible destination register for the polyhedral instruction and therefore corresponds to a fragment node \((I, a, G)\), where
• the domain \( I \) is the write space of \( e \);
• the attribute tuple \( a \) only contains the fragment \( \text{reg}_{\text{write}}[v] \) that represents the
  registers allocated for “writing” to the edge;
• the set of children \( G \) is empty.

The polyhedral operand representing \( rd \) is then generated as follows:

\[
f_{rd} := \text{combine}(rd, \{g_e | e = (v, w) \in E\}), \quad g_e := (W^*[e], (\text{reg}_{\text{write}}[e]), \emptyset)
\] (5.2)

Note that only output and dependence edges can be outgoing edges of an operation
node.

Analogously, we can generate the polyhedral operands representing \( rs_j \):

\[
f_{rs_{j,i}} := \text{combine}(rs_{j,i}, \{g_e | e = (w, v) \in (E^\text{in} \cup E^\text{dep})\}),
\] (5.3)

where for input and dependence edges:

\[
g_e := (\mathcal{R}^*[e], (\text{reg}_{\text{read}}[e]), \emptyset)
\] (5.4)

For constant edges:

\[
g_e := (\mathcal{R}^*[e], (c[e]), \emptyset)
\]

**Remark.** If a register annotation is a tuple of multiple registers, as can be the case for
general-purpose registers (see Section 4.9), code generation (Section 6.2) needs to take
care that the registers are cycled through correctly. For example, given the following
polyhedral operand:

\[
f := rs_1 \rightarrow (rd_0, rd_1) \{j > 0\}
\]

In the first iteration of the condition space, \( rd_0 \) is used, in the next \( rd_1 \), then again \( rd_0 \),
and so on, where the temporal order is according to the intra-tile schedule \( \lambda^j \).

Finally, the polyhedral instruction itself is generated by combining all polyhedral
operands:

\[
f_{\text{instr}} := \text{combine(mnemo, } \{f_{rd, f_{rs_{j,i}}, \ldots}\})
\] (5.5)

**Example** (Matrix multiplication, continuing from p. 90). Suppose node \( v_6 \) and its in-
coming and outgoing edges in Figure 3.4 have been annotated as follows after symbolic
modulo scheduling. The node has been annotated with the selected instruction template:

\[
\text{instr}[v_6] = \text{move rd rs}
\]

The only outgoing edge has been annotated with a register and write space:

\[
\text{reg}_{\text{write}}[(v_6, v_7)] = \text{od}0, \quad W^*[(v_6, v_7)] = \{p_3k_3 + j_3 = K - 1\}
\]
The two incoming edges for operand #0 have each been annotated with a register and the read space:

\[
\begin{align*}
\text{reg}((v_4, v_6)) &= \text{rd3}, \quad \mathcal{R}^*[(v_4, v_6)] = \{p_3 k_3 + j_3 = K - 1 \land K = 1\} \\
\text{reg}((v_5, v_6)) &= \text{rd3}, \quad \mathcal{R}^*[(v_5, v_6)] = \{p_3 k_3 + j_3 = K - 1 \land K > 1\}
\end{align*}
\]

First, we look at the destination operand rd. The only outgoing edge is \((v_6, v_7)\), so the corresponding polyhedral operand according to Equation (5.2) is

\[
f_{rd_{v_6}} := rd \longrightarrow \text{od0} \{p_3 k_3 + j_3 = K - 1\}
\]

For the first and only source operand, rs, there are two edges. For \((v_4, v_6)\), according to Equation (5.4), we get the polyhedral register

\[
\text{rd3} \{p_3 k_3 + j_3 = K - 1 \land K = 1\}
\]

For \((v_5, v_6)\), according to Equation (5.4), we get the polyhedral register

\[
\text{rd3} \{p_3 k_3 + j_3 = K - 1 \land K > 1\}
\]

Combining these two according to Equation (5.3) under rs yields the polyhedral operand

\[
f_{rs_{v_6}} := rs \longrightarrow \text{rd3} \{p_3 k_3 + j_3 = K - 1\}
\]

Finally, combining all operands under move according to Equation (5.5) results in the polyhedral instruction

\[
f_{instr_{v_6}} := \text{move} \quad rd \longrightarrow \text{od0} \{p_3 k_3 + j_3 = K - 1\} \\
rs \longrightarrow \text{rd3} \{p_3 k_3 + j_3 = K - 1\}
\]

\[\Delta\]

**Merging of polyhedral instructions**

The top-level polyhedral syntax tree is obtained by merging the polyhedral instructions. First, each polyhedral instruction is extended by the corresponding time offset \(\tau[v]\) and functional unit \(fu[v]\):

\[
\begin{align*}
 f'_{instr_r} &:= (\text{domain}(f_{instr_r}), (\tau[v]), \{f_{instr_r}\}) \\
 f''_{instr_r} &:= (\text{domain}(f_{instr_r}), (fu[v]), \{f'_{instr_r}\})
\end{align*}
\]

Merging all these extended polyhedral instructions yields the polyhedral syntax tree corresponding to the given RDG:

\[
f := \text{combine}(\text{root}, \{f''_{instr_r} \mid v \in V_{op}\})
\]
5 Symbolic Mapping

Example (Matrix multiplication, continuing from p. 106). Assume the operation node \( v_6 \) from the previous example was scheduled at time \( \tau = 3 \) and on functional unit \( \text{dpu0} \). Extending the polyhedral instruction then yields

\[ f''_{\text{instr}} := \text{dpu0} \rightarrow \tau : 3 \rightarrow \text{move} \rightarrow \begin{array}{c} rd \rightarrow \text{od0} \{ \{ p_3k_3 + j_3 = K - 1 \} \} \\ rs \rightarrow \text{rd3} \{ \{ p_3k_3 + j_3 = K - 1 \} \} \end{array} \]

Assuming node \( v_3 \) in Figure 3.4 has the following extended polyhedral instruction:

\[ f''_{\text{instr}} := \text{mul0} \rightarrow \tau : 1 \rightarrow \text{mul} \rightarrow \begin{array}{c} rd \rightarrow \text{rd2} \{ \{ \text{always} \} \} \\ rs_0 \rightarrow \text{rd0} \{ \{ \text{always} \} \} \\ rs_1 \rightarrow \text{rd1} \{ \{ \text{always} \} \} \end{array} \]

Then, the (partial) polyhedral syntax tree is

\[ f := \begin{array}{c} \text{dpu0} \rightarrow \tau : 3 \rightarrow \text{move} \rightarrow \begin{array}{c} rd \rightarrow \text{od0} \{ \{ p_3k_3 + j_3 = K - 1 \} \} \\ rs \rightarrow \text{rd3} \{ \{ p_3k_3 + j_3 = K - 1 \} \} \end{array} \\ \text{mul0} \rightarrow \tau : 1 \rightarrow \text{mul} \rightarrow \begin{array}{c} rs_0 \rightarrow \text{rd0} \{ \{ \text{always} \} \} \\ rs_1 \rightarrow \text{rd1} \{ \{ \text{always} \} \} \end{array} \end{array} \]

The polyhedral instructions for the remaining operation nodes are merged in analogously. \( \triangle \)

5.3 Access mappings

The symbolic configuration must specify how external data is accessed and its relation to the input and output registers annotated to the given RDG. To represent this information, for each input and output edge, an access mapping \( a \) is assembled from the edge’s annotations.

Definition 5.6 ([WWH⁺]). An access mapping is a four-tuple \( a = (\text{reg}, x, \alpha, \mathcal{A}) \) that maps all accesses to input or output register \( \text{reg} \) in iterations \( i \in \mathcal{A} \) to variable element \( x[\alpha(i)] \).

Given an input or output edge \( e \), the indexing function is \( \alpha = (Q[e], d[e]) \), its access space \( \mathcal{A} = I[v] \) (input access) or \( \mathcal{A} = I[w] \) (output access), the register \( \text{reg} = \text{reg}^{\text{read}}[e] \) (input access) or \( \text{reg} = \text{reg}^{\text{write}}[e] \) (output access), and the associated input variable \( x = x[v] \) or output variable \( x = x[w] \). We denote the set of all access mappings in a symbolic configuration \( \Lambda \).
Example (Matrix multiplication, continuing from p. 108). The access mappings $A = \{a_A, a_B, a_C\}$ are as follows:

- $a_A = (\text{reg}_A, x_A, a_A, \mathcal{A}_A)$: Edge $(v_{11}, v_1)$ represents read accesses to the input matrix $A = x_A$. It is indexed using
  $$a_A = (Q[(v_{11}, v_1)], d[(v_{11}, v_1)]) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$
  Its access space is
  $$\mathcal{A}_A = I[v_{11}] = \{j = 0\}$$
  The corresponding input register depends on the allocated register for edge $(v_{11}, v_1)$, but we assume it to be
  $$\text{reg}_A = \text{reg}^{\text{read}}[(v_{11}, v_1)] = \text{id0}.$$

- $a_B = (\text{reg}_B, x_B, a_B, \mathcal{A}_B)$: Edge $(v_8, v_1)$ represents read accesses to the input matrix $B = x_B$. It is indexed using
  $$a_B = (Q[(v_8, v_1)]) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, d[(v_8, v_1)] = 0$$
  Its access space is
  $$\mathcal{A}_B = I[v_8] = \{i = 0\}$$
  The corresponding output register depends on the allocated register for edge $(v_8, v_1)$, but we assume it to be
  $$\text{reg}_B = \text{reg}^{\text{read}}[(v_8, v_1)] = \text{id1}.$$

- $a_C = (\text{reg}_C, x_C, a_C, \mathcal{A}_C)$: Edge $(v_6, v_7)$ represents write accesses to the output matrix $C = x_C$. It is indexed using
  $$a_C = (Q[(v_6, v_7)]) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, d[(v_6, v_7)] = 0$$
  Its access space is
  $$\mathcal{A}_B = I[v_7] = \{k = K - 1\}$$
  The corresponding input register depends on the allocated register for edge $(v_6, v_7)$, but we assume it to be
  $$\text{reg}_B = \text{reg}^{\text{write}}[(v_6, v_7)] = \text{id0}.$$

During instantiation, concrete configuration data for the I/O buffers (in particular the address generators) is generated from the access mappings and the then-known memory layout associated with each input and output variable (see Section 6.5).
5 Symbolic Mapping

5.4 Summary

Given a space mapping $\phi$ (Section 3.4.3) and a TCPA architecture description (Section 2.1), symbolic mapping—the generation of a symbolic configuration from an RDG—comprises the following steps:

1. **Symbolic modulo scheduling** (Chapter 4) determines a tiled schedule vector $\lambda^*$, and annotates the RDG with the information required for the generation of a polyhedral syntax tree: each operation node with a time offset $\tau$, a functional unit $fu$, and an instruction template $instr$; each input, output, and dependence edge with register(s) $reg_{\text{read}}$, $reg_{\text{write}}$ and, if applicable, a propagation channel $\rho$.

2. **Generation of a polyhedral syntax tree** (Section 5.2.2) yields a unique corresponding polyhedral syntax tree by generating one polyhedral instruction for each operation node in the RDG.

3. **Assembling of access mapping** (Section 5.3) finally generates an access mapping from each input and output edge in the given RDG.

Note that overall, these steps need to be repeated for each space mapping and feasible scanning order to yield a set of symbolic configurations from which, at run time, the latency-minimal one is chosen according to the concrete loop bounds and number of available PEs. Since this might result in a large number of symbolic configurations, Chapter 7 presents methods to reduce the number while still guaranteeing given requirements and optimizing a number of secondary objectives like energy consumption.

But before closing the chapter with an example, we would like to emphasize one more time the beauty of being able to perform the generation of a symbolic mapping (including the solution of a number of NP-hard problems) at compile time—therefore producing a high-quality mapping due to no time constraints—and shifting the steps with lower time complexity to the run time, where the loop bounds and number of available PEs have become known.

**Example** (Bit extraction). Consider the following UDA with an iteration space of $I = \{0 \leq i < N\}$:

\[
\begin{align*}
S_1: & \quad x[i] = \text{in} & \text{if } i = 0 \\
S_2: & \quad x[i] = y[i-1] & \text{if } i \geq 1 \\
S_3: & \quad y[i] = x[i] \text{shr} 1 \\
S_4: & \quad \text{bits}[i] = x[i] \text{ and } 1
\end{align*}
\]

This UDA describes the extraction of the $N$ bits of an $N$-bit integer, provided by the scalar input variable $in$, into the one-dimensional output variable $bits$ with the least-significant bit first. For example, assuming $N = 4$ and given $in = 13 = 1101_2$, the elements of $bits$ are

5.4 Summary

We use this simple, rather artificial example to be able to give a comprehensive overview of both all steps of symbolic mapping and, in the next chapter, instantiation of concrete configuration data and code. To recapitulate: the symbolic configuration consists of a space mapping $\Phi$, a tiled schedule vector $\lambda^*$, a polyhedral syntax tree $f$, a set of propagation routes $R$, and a set of access mappings $A$.

Figure 5.1 shows the corresponding RDG before and Figure 5.2 the RDG after symbolic modulo scheduling using the 1-dimensional allocation matrix

$$\Phi = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

which maps the UDA onto a single row of PE. We assume there are two functional units, a1u0 and a1u1, both of which support the following instruction set:

$$\{\text{move rd rs}_0, \text{shri rd rs}_0 \text{ rs}_1, \text{andi rd rs}_0 \text{ rs}_1\}$$

There is only one feasible scanning order:

$$s = (\sigma = (1), \omega = (1))$$

Since this example is 1-dimensional, there is only one tile size parameter, $p$, and one tile count parameter, $t$. With an initiation interval of $\pi = 2$, the resulting symbolic schedule vector is

$$\lambda^* = (\lambda^j = (2) \quad \lambda^k = (2p))$$

Not shown in the RDG is the propagation route allocated for edge $e^2_4$:

$$R = \{\rho[e^2_4], \rho[e^2_4] \leftarrow (\text{pe}^\triangledown_1 \rightarrow \text{east}^<_0 \rightarrow \text{west}^\triangledown_0 \rightarrow \text{pe}^<_0)\}$$

This route will be replicated across all allocated PEs and be used to propagate intermediate results. Note that the first port, $\text{pe}^\triangledown_1$, corresponds to the allocated register $\text{id}_1$ and the last port, $\text{pe}^<_0$, corresponds to the allocated register $\text{id}_0$.

The polyhedral syntax tree generated from the annotated RDG is:

```
\begin{align*}
\text{f := root} & \quad \tau : 0 \quad \text{move} \\
\text{a1u0} & \quad \text{rd} \quad g_1 := \text{rd}0 \{\{pk + j < N\}\} \\
\text{rs} & \quad g_2 := \text{id}0 \{\{j = 0\}\} \\
& \quad g_3 := \text{rd}1 \{\{j > 0 \land pk + j < N\}\} \\
& \quad g_4 := \text{od}0 \{\{j = p - 1 \land k < t - 1\}\} \\
\text{rd} & \quad g_5 := \text{rd}1 \{\{j < p - 1 \land pk + j < N\}\} \\
\text{tau : 1} & \quad \text{shri} \\
\text{rs}_0 & \quad g_6 := \text{rd}0 \{\{pk + j < N\}\} \\
\text{rs}_1 & \quad g_7 := 1 \{\{pk + j < N\}\} \\
\text{rd} & \quad g_8 := \text{od}1 \{\{pk + j < N\}\} \\
\text{tau : 2} & \quad \text{andi} \\
\text{rs}_0 & \quad g_9 := \text{rd}0 \{\{pk + j < N\}\} \\
\text{rs}_1 & \quad g_{10} := 1 \{\{pk + j < N\}\}
\end{align*}
```
5 Symbolic Mapping

\[
\begin{align*}
\text{\(v_1\): } & \text{in} \\
i & = 0
\end{align*}
\]

\[
\begin{align*}
\text{\(v_2\): } & S_1 \\
\text{copy(\(\#0\))} \\
i & = 0
\end{align*}
\]

\[
\begin{align*}
\text{\(e_4\): } & d = (1) \\
\text{\(\#0\)} \\
i & = 0
\end{align*}
\]

\[
\begin{align*}
\text{\(v_3\): } & S_3 \\
\text{shr(\(\#0, \#1\))} \\
0 & \leq i < N
\end{align*}
\]

\[
\begin{align*}
\text{\(v_4\): } & S_2 \ [=} \\
1 & \leq i < N
\end{align*}
\]

\[
\begin{align*}
\text{\(v_5\): } & \text{bits [output]} \\
0 & \leq i < N
\end{align*}
\]

Figure 5.1: RDG corresponding to the bit extraction example before symbolic modulo scheduling. Note that \(\text{in}\) is a scalar value, which is why \(Q\) and \(d\) are both 0-dimensional. Edge \(e_4\) is split during symbolic modulo scheduling because \(|\Delta(d[e])| > 1\).

Finally, the access mappings \(A = \{a_{in}, a_{bits}\}\) are assembled from the RDG:

- \(a_{in}\): Edge \(e_1\) represents accesses to the input scalar \(\text{in} = x_{\text{in}}\). We model scalars as zero-dimensional variables, that means, the indexing function is \(\alpha_{in} = (Q[e_1] = () \in \mathbb{Z}^{0 \times n}, d[e] = () \in \mathbb{Z}^0)\). (For these empty matrices, we assume that \(\mathbb{Z}^{0 \times n}, \mathbb{Z}^n \in \mathbb{Z}^0\)) Its access domain is \(\mathcal{A}_{in} = I[v_1] = \{i = 0\}\), the allocated input register \(\text{reg}_{in} = id\).

- \(a_{bits}\): Edge \(e_7\) represents accesses to the one-dimensional output variable \(\text{bits} = x_{\text{bits}}\). The indexing function is \(\alpha_{bits} = (Q[e_7] = (1), d[e_7] = (0))\). It is written to in each iteration, meaning \(\mathcal{A}_{bits} = I[v_7] = \{i < N\}\). The allocated output register \(\text{reg}_{bits} = od\).

The next chapter continues this example and shows how a concrete TCPA configuration can be instantiated from this symbolic configuration.

5.5 Related Work

APOLLO [IoS16; MCS17] is a speculative loop optimizer that pre-compiles code skeletons [JCD14] and, more recently, code bones [CWC16]. These get assembled at run time,
5.5 Related Work

Figure 5.2: RDG of the bit extraction example after symbolic modulo scheduling. Each operation node has been annotated with a functional unit $fu$, a start offset $\tau$, and an instruction template $instr$, here displayed in the form $fu \ [\tau] \ instr$. Input, output, and dependence edges have been annotated with appropriate registers; edge $e_i^j$ additionally has been annotated with a propagation route $\rho$, which is not shown here.
similar to polyhedral syntax trees, but they lack the capability to represent iteration-dependent condition spaces that are required for assembling modulo-scheduled programs in the compact manner required by PE with small instruction memories.

Regarding intermediate representations of loops in the polyhedral model that contain mapping information, schedule trees represent per-statement affine schedules from which loop code can easily be generated [VGG+14; GVC15]. Schedule trees form the basis for some loop optimization techniques such as loop tactics [CZG+19] and tensor comprehensions [VZT+19]. Loop tactics uses pattern matchers both on the schedule trees and statements themselves to find composable optimizations such as tiling and fusion. Tensor comprehensions are themselves a kind of intermediate representation of tensor computations, in particular in the field of deep learning. While schedule trees are able to represent the affine schedule of an imperfectly nested loop, they are not suitable for the representation of modulo-scheduled assembly programs.
Instantiation is the second phase of symbolic loop compilation. Given a symbolic configuration and an assignment of concrete values to the loop bounds and tile size parameters, instantiation generates concrete program code for all allocated PEs and configuration data for the whole target TCPA. The symbolic configuration to be instantiated is selected from the set of symbolic configurations generated by the preceding phase, symbolic mapping. If run-time requirement enforcement is enabled, an enforcer then selects one satisfying a given set of non-functional requirements (see Chapter 7). Otherwise, one resulting in the lowest latency according to the concrete parameter values is automatically selected (see Section 4.7).

Performing instantiation at run time must be efficient and must be independent of the problem size, that is the number of available PE and the concrete loop bounds. In particular, generating concrete PE programs is the most complex step of instantiation and, using a naïve implementation, has a time complexity that is linear in the number of PEs and would prevent scaling with increasing TCPA size. In this chapter, we present a method that is independent of the number of allocated PEs.

Roadmap. First, Section 6.1 summarizes the flow of instantiation from a symbolic configuration to a concrete configuration (which we call the collection of concrete program and configuration data for the target TCPA). Section 6.2 explains, given the intra-tile schedule vector and polyhedral syntax tree, how the control flow of the given loop with the given mapping is analyzed to yield a set of control flow graphs. This analysis is paramount in ensuring only the set of distinct PE programs is generated. Then, Section 6.3 shows how to allocate the corresponding control signals and generate configuration data for a global controller such that it orchestrates the execution of the PE programs; their generation from the control flow graphs by inserting appropriate branch instructions is described in Section 6.4. Finally, in Section 6.5, the generation of configuration data for the I/O buffers is explained.

6.1 Overview

Instantiation comprises these steps:

1. **Concretization** substitutes all occurrences of parameters in a given symbolic
configuration, for example, in the iteration space $I^*$, the symbolic schedule $\lambda^*$, or the polyhedral syntax tree $f$, with concrete values. Using the concretized schedule, the feedback register depths ($\lambda'd/\pi$, see Section 4.9) are computed.

2. **Program instantiation** is the most complex instantiation step and further divided into three sub-steps:

   a) **Control flow analysis** (Section 6.2) first determines a set of so-called processor classes, that is, a partitioning of $K$ into subsets $P_{pc}$ of PEs that will execute the same PE program. For each processor class $P_{pc}$ and each functional unit $fu$ in the polyhedral syntax tree $f$, a control flow graph $CFG_{pc,fu}$ is generated from the processor class-specific polyhedral syntax tree $f_{pc}$ and the concretized intra-tile schedule $\lambda$. Each node in $CFG_{pc,fu}$ represents an atomic sequence of instructions, and each edge represents a branch annotated with its transition space $T$ (the iterations in which the branch is taken).

   b) **Control signal allocation** (Section 6.3) allocates a set of binary control signals that, for each iteration $j$ in the intra-tile space $J$, encode for all transition spaces across all processor classes and control flow graphs whether $j \in T$. The corresponding control signal values are annotated back to the control flow graph edges and used later for generating branch conditions.

   c) **Program generation** (Section 6.4) generates a PE program for each processor class $P_{pc}$ with a functional unit program for each $CFG_{pc,fu}$: The instruction sequences represented by the nodes are concatenated and branch instructions generated according to the control signal values annotated to the edges.

3. **Periphery instantiation** generates concrete configuration data for the global controller from the allocated set of control signals (therefore also described in Section 6.3). Furthermore, for each access mapping $a \in A$, each involved PE is connected to a memory bank, whose address generator configuration is generated according to the indexing function $a$ (Section 6.5).

4. **Interconnect instantiation** replicates the propagation channels for all allocated interconnect wrappers and incorporates any additional routes from the I/O routing.

Instantiation is, in general, performed at run time. Making run-time instantiation viable requires the above steps to scale well with an increasing number of PEs, which especially matters for the most complex part of instantiation: program instantiation.

**Example** (Bit extraction, continuing from p. 110). Figure 6.1 gives an overview of how instantiation connects a given symbolic configuration with the hardware components within the target TCPA. Assume the bit extraction example shall be mapped onto $t = 3$
PEs with the loop bound parameter $N = 16$. The goal of instantiation is two-fold: (1) generate individual programs such that each PE receives a program that executes the iterations within its assigned tile according to the concretized schedule, and (2) generate configuration data for the remaining TCPA components (interconnect, buffers, global controller) that synchronizes them to the PE programs.

Fortunately, the number of different PE programs is often significantly smaller than the number of PEs. In our example, given concrete parameter values, for the allocation of 3 PEs, there will be 2 PE programs generated from the concretized polyhedral syntax tree. In this case, concretizing the polyhedral syntax tree yields:

$$\lambda^* = (\pi \cdot \pi \cdot p) = (2 \cdot 12).$$

This problem is solved in Section 6.2 on control flow analysis. Rather than pure PE-local control flow, a global controller is used in TCPAs to propagate control signals using the control layer of the interconnect to the individual PEs (see the green part in Figure 6.1). In Section 6.3, we explain how to generate a minimal number of such control signals.

Next, from the access mappings, configuration data for the I/O buffers is produced. In this particular case, one I/O buffer provides the input data, and three I/O buffers receive the output data from the individual PEs (see the yellow part in Figure 6.1). Finally, the interconnect configuration data is generated from the given propagation channels as well as any routes generated during the instantiation of the access mappings (see the purple part in Figure 6.1).
Figure 6.1: (From [WWH⁺]) Relationship between the symbolic configuration and the TCPA hardware components involved for the bit extraction example for the case of $t = 3$ and $N = 16$. Each color represents a different group of components: Green represents the configuration data for the global controller and corresponding propagation of the generated control signals on the control layer of the interconnect network (depicted by dashed lines and round ports), which steer the control flow of the three allocated PEs. Blue represents the two generated PE programs for the three allocated PEs. Yellow represents the configuration data for the I/O buffers and address generators, which synchronizes in- and outputting of data to and from the PEs. Purple represents the configuration data for the data layer of the interconnect network (depicted by solid lines and rectangular ports) to establish the necessary data propagation routes.
Example (Matrix multiplication, continuing from p. 109). There are six feasible scanning orders, meaning symbolic mapping generates six symbolic configurations. Suppose a symbolic configuration is selected that contains the following symbolic schedule vector:

$$\lambda^* = (1 \ p_1 \ p_2 p_1 \ p_1 \ (p_2 - 1)p_1 + 1 \ (p_3 - 1)p_2 p_1 + 1)$$

Assuming the tile sizes are calculated from a given number of available PEs (see Section 3.4.3) to be

$$p_1 = 256, p_2 = 256, p_3 = 1024,$$

the concretized tiled schedule vector is

$$\lambda^* = (1 \ 256 \ 65536 \ 256 \ 65281 \ 67043328)$$

The depth feedback registers allocated for edges with dependence vector $$d = (0, 1, 0)^T$$ then is

$$\lambda^* d = (1 \ 256 \ 65536) \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = 256$$

6.2 Control flow analysis

Generating a program for each PE separately does not scale to arbitrary TCPA sizes—fortunately, due to the regularity of loops, large subsets of PEs usually execute the same program. Thus, by identifying regions of PEs that will be assigned a unique instruction sequence and generating one distinct program only for each identified region, program instantiation scales well because the number of distinct programs across PEs is bounded even if the number of PEs keeps increasing. But is it possible to determine whether multiple PEs will be configured with the same program without actually generating their programs? Yes, by reasoning not about the PE programs themselves, but rather the polyhedral syntax tree: We consider the programs of two PEs $$k_1$$ and $$k_2$$ equal if specialization yields the same polyhedral syntax tree for both. Specialization is defined in the following and is based on the intra-tile domain of a tile $$k$$. (Note that in the following, we use tile and PE interchangeable; for the reasoning, refer to Section 3.4.3.)

Definition 6.1 ([WHH⁺]). Given a tiled condition space$$^1$$ $$I^* \subseteq J \oplus K$$, the function

$$\text{split} : k, I^* \mapsto \widehat{J} = \{ j \in J \mid (j^T, k^T)^T \in I^* \}$$

returns the intra-tile domain $$\widehat{J}$$ of a tile $$k \in K$$ within $$I^*$$, that is, the set of iterations $$j$$ within tile $$k$$ that lie in $$I^*$$.

$$^1$$Or, more accurately: a condition space embedded into the tiled iteration space.
Definition 6.2 ([WHW^+]). Given a polyhedral syntax tree $f$ with a tiled condition space $I^* = \text{domain}(f) \subseteq J \oplus K$, specialization for a tile $k \in K$, denoted $f \triangleright k$, recursively maps the condition spaces of all children of $f$ to the intra-tile domain of $k$ within $I^*$:

$$f \triangleright k := (\text{split}(k, \text{domain}(f)), \text{attr}(f), \{ g \triangleright k \mid g \in \text{children}(f) \})$$

In other words, specialization transforms a polyhedral syntax tree that describes the PE programs for all PEs into a new polyhedral syntax tree that only represents the program of the single PE $k$. A specialized polyhedral syntax tree only refers to intra-tile iteration variables anymore, which we make clear in the notation: after specialization, a polyhedral syntax tree is evaluated at intra-tile iterations, for example $f(j)$.

Example (Bit extraction, continuing from p.116). Multiple leaves of the concretized polyhedral syntax tree $f^{\text{bit}}$ have the tiled condition space $\text{domain}(g_1) = \{6k + j < 16\}$. Depending on $k$, this tiled condition space has two different intra-tile domains:

- $\text{split}(k = (0)^T, \{6k + j < 16\}) = \{j < 16\} \equiv \{\text{always}\}$
- $\text{split}(k = (1)^T, \{6k + j < 16\}) = \{j < 16\} \equiv \{\text{always}\}$
- $\text{split}(k = (2)^T, \{6k + j < 16\}) = \{12 + j < 16\} \equiv \{j < 4\}$

These correspond to full tiles ($k = (0)^T$ and $k = (1)^T$) and partial tiles ($k = (2)^T$). Overall, specialization for $k = (0)$ yields:

\[ f \triangleright (0) := \text{root} \]

\[ \text{alu0} \xrightarrow{\tau: 0} \text{alu0} \quad \tau: 1 \xrightarrow{\text{shri}} \text{alu0} \]

\[ \text{alu0} \xrightarrow{\tau: 0} \text{move} \quad \text{src} \quad \text{dest} \quad g_1 := \text{rd} \emptyset \{\text{always}\} \]

\[ g_2 := \text{id} \emptyset \{j = 0\} \]

\[ g_3 := \text{rd} 1 \{j > 0\} \]

\[ g_4 := \text{od} \emptyset \{i = 5\} \]

\[ g_5 := \text{rd} 1 \{j < 5\} \]

By contrast, specialization for $k = (2)$ yields:
6.2 Control flow analysis

Note that node $g_4$ can be removed because its intra-tile domain is empty; the program for the corresponding PE will therefore never contain the instruction shri od0 rd0 1. Similarly, all condition spaces are constrained to $j < 4$ because $k = (2)$ is a partial tile with “empty” iterations. Also note that all condition spaces now are subsets of the intra-tile space $J$. △

**Example** (Matrix multiplication, continuing from p. 117). The following is an excerpt from the concretized polyhedral syntax tree representing the PE programs of the matrix multiplication example:

\[
\begin{align*}
    f^{mm} := root & \quad \text{dpu0} \quad \tau: 0 \quad \text{move} \\
    g^1 & := \text{id0} \quad \{j = 0\} \\
    g^2 & := \text{rd0} \quad \{j < 4\} \\
    g^3 & := \text{od1} \quad \{j < 4\} \\
    g^4 & := \text{rd0} \quad \{j < 4\} \\
    g^5 & := \text{id0} \quad \{j = 0\} \\
    g^6 & := \text{fv0} \quad \{j > 0 \land j < 4\} \\
    g^7 & := \text{al0} \quad \{j < 4\} \\
    g^8 & := \text{al1} \quad \{j < 4\} \\
    g^9 & := \text{al1} \quad \{j < 4\} \\
    g^{10} & := \text{al0} \quad \{j < 4\}
\end{align*}
\]

Given, for example, any $k \in \mathcal{K}$, the intra-tile domain of $g_1^{mm}$, $g_2^{mm}$, and $g_3^{mm}$ are:

- $\text{split}(k, \text{domain}(g_1^{mm})) = \{\text{always}\}$
- $\text{split}(k, \text{domain}(g_2^{mm})) = \{j_2 = 0\}$
- $\text{split}(k, \text{domain}(g_3^{mm})) = \{j_2 > 0\}$

Since all three only depend on intra-tile iteration variables, their intra-tile domains are equal to their original condition spaces. For the given partial polyhedral syntax tree, we therefore have $f^{mm} = f^{mm} \bowtie k$. △

### 6.2.1 Determination of processor classes

Using specialization, we partition the inter-tile space $\mathcal{K}$ into a set $PC$ of processor classes $P_{pc}$, each of which is a set of PEs that result in an identical specialized polyhedral syntax tree and thus program. In other words, each processor class represents one distinct PE program and $|P_{pc}|$ is equal to the number of PE programs requiring instantiating. When is $f \bowtie k_1 = f \bowtie k_2$ for two PEs $k_1 \neq k_2$? Since specialization only transforms
condition spaces, and all condition spaces in a polyhedral syntax tree are unions of children's condition spaces (compare Definition 5.3), the two specialized polyhedral syntax trees are equal if \( \text{domain}(g \cdot k_1) = \text{domain}(g \cdot k_2) \) for all leaves \( g \) of \( f \). We denote the leaves of \( f \) using leaves(\( f \)) in the following. Specialization maps condition spaces to their corresponding intra-tiled domain, meaning we must investigate for which subsets of \( K \) two tiles \( k_1 \) and \( k_2 \) result in the same intra-tiled domain \( \widehat{\mathcal{J}} \) within a tiled condition space \( I^* \).

**Definition 6.3 ([WWH⁺])**. Given a tiled condition space \( I^* \subseteq K \oplus J \), the set of tiles with the same intra-tiled domain \( \widehat{\mathcal{J}} \), called its *inter-tiled domain* \( \widehat{\mathcal{K}} \), is given by the function

\[
\text{tiles}: \widehat{\mathcal{J}}, I^* \mapsto \widehat{\mathcal{K}} = \{ k \mid k \in K \land \text{split}(k, I^*) = \widehat{\mathcal{J}} \}.
\]

We call \( \widehat{I} = (\widehat{\mathcal{J}}, \widehat{\mathcal{K}}) \) the *intra-tile pattern* of \( \widehat{\mathcal{K}} \) within \( I^* \). We denote the set of all intra-tile patterns of \( I^* \) as \( \nabla I^* \), which always corresponds to a partitioning of \( K \).

**Example** (Bit extraction, continuing from p. 120). Given

\[
I^* = \text{domain}(g_1) = \{6k + j < 16\},
\]

there are \( |\nabla I^*| = 2 \) intra-tile patterns:

\[
\begin{align*}
\text{\( \widehat{\mathcal{I}}_1 \)} & = (\{0 \leq j < 6\}, \{k < 2\}) \\
\text{\( \widehat{\mathcal{I}}_2 \)} & = (\{0 \leq j < 4\}, \{k = 2\})
\end{align*}
\]

In other words, if \( k < 2 \), then iterations \( i^* = (j, k)^T \) with \( 0 \leq j < 6 \) are elements of the tiled condition space; if \( k = 2 \), then only iterations \( i^* = (j, 2)^T \) with \( 0 \leq j < 4 \) are elements of the tiled condition space.

**Example**. Let the following square iteration space \( I \) and non-orthogonal condition space \( I_{\text{diag}} \) be given:

\[
I = \{0 \leq i_1, i_2 < 1024\}, \quad I_{\text{diag}} = \{i_1 > i_2\}
\]

Assuming \( t_1 = t_2 = 4 \) tiles, the tiled iteration space is

\[
I^* := \{0 \leq j_1, j_2 < 256 \land 0 \leq k_1, k_2 < 4\}.
\]

Accordingly, \( I_{\text{diag}} \) embedded into \( I^* \) is

\[
I^*_{\text{diag}} = \{256 \ast k_1 + j_1 > 256 \ast k_2 + j_2\}
\]

This non-orthogonal condition space results in \( |\nabla I^*| = 2 \) intra-tile patterns:

\[
\begin{align*}
\text{\( \widehat{\mathcal{I}}_1 \)} & = (\{j_1 > j_2\}, \{k_1 = k_2\}) \\
\text{\( \widehat{\mathcal{I}}_2 \)} & = (\{0 \leq j_1, j_2 < 256\}, \{k_1 > k_2\})
\end{align*}
\]
Now, however, assume $I$ is tiled into $t_1 = 8$ and $t_2 = 4$ tiles instead:

\[
I^* := \{0 \leq j_1 < 128 \land 0 \leq j_2 < 256 \land 0 \leq k_1 < 8 \land 0 \leq k_2 < 4\}
\]

\[
I^\text{diag} = \{128 \cdot k_1 + j_1 < 256 \cdot k_2 + j_2\}
\]

In this case, we obtain $|I^*| = 3$ intra-tile patterns:

\[
I^*_1 = (\{j_1 > j_2\}, \{k_1 = 2k_2\})
\]

\[
I^*_2 = (\{j_1 > j_2 - 128\}, \{k_1 = 2k_2 + 1\})
\]

\[
I^*_3 = (\{0 \leq j_1 < 128, 0 \leq j_2 < 256\}, \{k_1 > 2k_2 + 1\})
\]

Clearly, different tiling parameters may lead to different intra-tile patterns of the same condition space and thus to different PE programs—one of the reasons control flow analysis must take place at runtime.

Inferring from the definition of an intra-tile pattern in Definition 6.3, if both $k_1$ and $k_2$ are in the inter-tile domain $\mathcal{K}$ of the same intra-tile pattern $\mathcal{I} = (\mathcal{J}, \mathcal{K})$, then specialization of $g$ results in domain($g \triangleright k_1$) = domain($g \triangleright k_2$). Consequently, the determination of processor classes depends only on the inter-tile domains: If $k_1$ and $k_2$ are in the same inter-tile domain for each leaf of $f$, they result in the same specialized polyhedral syntax tree and thus processor class. Figure 6.2 illustrates this relation between processor classes and intra-tile patterns, as well as why two PEs share the same program if the specialized polyhedral syntax trees are equal. To formalize this visual intuition, let $\mathbb{K} := \{\mathcal{K}_1, \mathcal{K}_2, \ldots\}$ be the set of all inter-tile domains annotated to the leaves of $f$:

\[
\mathbb{K} = \left\{ \mathcal{K} \mid g \in \text{leaves}(f) \land (\mathcal{J}, \mathcal{K}) \in \text{domain}(g) \right\}.
\]

For each PE $k \in \mathcal{K}$, there is a partitioning of $\mathbb{K}$ into two subsets: $\mathbb{K}^+_k$, containing all $\mathcal{K}_i$ such that $k \in \mathcal{K}_i$, and $\mathbb{K}^-_k$, containing all $\mathcal{K}_i$ such that $k \notin \mathcal{K}_i$. If this partitioning is equal for two PEs $k_1$ and $k_2$, then by definition, both $k_1$ and $k_2$ belong to the same inter-tile domain for each leaf of $f$. Given a specific PE $k$, the set $\mathcal{P}_k$ of PEs with the same partitioning is

\[
\mathcal{P}_k = \bigcap_{\mathcal{K} \in \mathbb{K}^+_k} \mathcal{K} \cap \bigcap_{\mathcal{K} \in \mathbb{K}^-_k} \overline{\mathcal{K}}.
\]

The set of processor classes then is $PC = \{\mathcal{P}_k \mid k \in \mathcal{K}\}$.

**Example** (Bit extraction, continuing from p. 122). Assuming the same tiling parameters as before, there are three distinct inter-tile domains arising from the condition spaces annotated to the leaves of $f$:

\[
\mathbb{K} = \{\mathcal{K}_1 = \{k < 2\}, \mathcal{K}_2 = \{k \leq 2\}, \mathcal{K}_3 = \{k = 2\}\}
\]
wecandetermineallotherPEswiththesamepartitioning:

This means all PEswith

As illustrated in Figure 6.2, there are two processor classes overall: $PC = \{ P_1 = \{ k < 2 \}, P_2 = \{ k = 2 \} \}$.
6.2 Control flow analysis

**Algorithm 1** Partition $\mathcal{D}$ into subsets $D = \{D_1, \ldots\}$ according to conditions $C = \{C_1, \ldots\}$

\[
\begin{align*}
\text{partition } \mathcal{D} & \text{ into subsets } D = \{D_1, \ldots\} \text{ according to conditions } C = \{C_1, \ldots\} \\
D & \leftarrow \{D\} \quad \triangleright \text{ start with full space } \mathcal{D} \text{ as “partitioning”} \\
\text{for } C \in C \text{ do} & \quad \triangleright \text{ for each condition, partition current set of subsets} \\
D' & \leftarrow \emptyset \\
\text{for } D' \in D \text{ do} & \quad \triangleright \text{ partition each subset according to condition into up to two subsets} \\
\text{if } D' \cap C \neq \emptyset & \text{ then } D' \leftarrow D' \cup \{D' \cap C\} \\
\text{if } D' \cap C \neq \emptyset & \text{ then } D' \leftarrow D' \cup \{D' \cap C\} \\
\text{end for} \\
D & \leftarrow D' \\
\text{end for}
\end{align*}
\]

6.2.2 Determination of kernel classes

Next, for each processor class $\mathcal{P}_{pc}$, a control flow graph for each functional unit in the specialized PST $f_{pc} = f \triangleright k, k \in \mathcal{P}_{pc}$ is constructed. However, generating compact programs necessitates exploiting repetition, but successive iterations may overlap in time due to the software pipelining introduced by symbolic modulo scheduling.

**Example** (Bit extraction, continuing from p. 122). Suppose we fully unroll the functional unit program described by alu1 of $f_1 = f \triangleright (0)^T$ into pseudo-assembly (time offset $\tau$ in brackets):

\[
\begin{align*}
[0] & \quad \text{nop} \quad \triangleright \text{ j = 0 starts (prolog)} \\
[1] & \quad \text{shri fd0 rd0 1} \\
[2] & \quad \text{andi od1 rd0 1} \quad \triangleright \text{ j = 1 starts --\} \\
[1] & \quad \text{shri fd0 rd0 1} \\
\text{-- repetition} \quad \triangleright & \text{...} \\
[2] & \quad \text{andi od1 rd0 1} \quad \triangleright \text{ j = 4 starts |} \\
[1] & \quad \text{shri fd0 rd0 1} \quad \triangleright & \text{...} \\
[2] & \quad \text{andi od1 rd0 1} \quad \triangleright \text{ j = 5 starts |} \\
[1] & \quad \text{shri od0 rd0 1} \\
[2] & \quad \text{andi od1 rd0 1} \quad \triangleright \text{ epilog starts}
\end{align*}
\]

Observe that iterations overlap, and that between $1 \leq j \leq 4$, the same two instructions repeat. A timing-equivalent *compact* program is the following, arranged by iteration:

\[
\begin{align*}
\text{nop} & \quad \triangleright \text{ j = 0} \\
\text{L1: andi od1 rd0 1 ; shri fd0 rd0 1 ; goto L1 if j <= 4} & \quad \triangleright 1 \leq j \leq 4 \\
\text{andi od1 rd0 1 ; shri od0 rd0 1} & \quad \triangleright j = 5 \\
\text{andi od1 rd0 1} & \quad \triangleright \text{ epilog}
\end{align*}
\]

The $\pi = 2$ instructions that repeat originate from different evaluations of $f_1((j)^T)$—andi from iterations $0 \leq j \leq 3$ and shri from iterations $1 \leq j \leq 4$. But how can we find such a compact program without unrolling the program?  
\[\triangle\]
In a modulo-scheduled functional unit program, the next iteration is started every \( \pi \) time steps. The execution of the program is therefore equivalent to a sequence of assembly kernels that have a length of \( \pi \) slots each. Each slot may be occupied by one instruction (otherwise, it is an implied \textit{nop}). For example, in the compact program example above, each line is a kernel, showing that kernels may repeat. Since a kernel represents the \( \pi \) instructions issued from the start of one iteration \( j \) until the start of the temporarily next iteration \( \text{succ}(j) \), there is an unambiguous mapping from iterations \( j \) to kernels; we say a kernel is issued (or executed) in iteration \( j \). Each kernel is executed atomically and there is no overlap between kernels, so control flow only changes between kernels, that is, each \( \pi \) time steps. We use this observation to reformulate the problem: How do we determine all distinct kernels of a functional unit program from a polyhedral syntax tree? How do we construct a corresponding control flow graph that exploits repeating kernels?

To answer this, we first look at the formation of kernels in a functional unit program, visualized in Figure 6.3. Let \( f_{pc, fu} \) be the polyhedral functional unit program associated with \( fu \). Then, at the start time \( t(j) \) of each iteration \( j \), the sequence of instructions described by the evaluation \( f_{pc, fu}(j) \) is issued. Each child \( g \) of \( f_{pc, fu} \) represents a polyhedral time slot with temporal offset \( \tau[g] \) relative to \( t(j) \), from which we can compute into which future iterations it overlaps: Because after \( \pi \) timesteps, the next iteration starts, any instruction at offset \( \tau[g] \) is executed within slot \( (\tau \mod \pi) \) of the kernel issued in iteration \( \text{succ}(j, \lfloor \tau / \pi \rfloor) \). Here, \( \text{succ}(j, n) \) is the \( n \)-th successor of \( j \) according to the intra-tile schedule \( \lambda_j \). Since \( g \) is issued whenever \( j \) is in \( g \)'s condition space \( J_g = \text{domain}(g) \), the instruction at \( \tau[g] \) therefore occupies the slot of the kernel issued in all iterations that are the \( \lfloor \tau[g] / \pi \rfloor \)-th successor of an iteration in \( J_g \) (compare the red instructions in Figure 6.3), given by

\[
\text{succ}(J_g, n) := \{ j = \text{succ}(j', n) \mid \forall j' \in J_g \}, \quad n = \lfloor \tau[g] / \pi \rfloor.
\]

**Example** (Bit extraction, continuing from p. 125). Consider the following polyhedral functional unit program:

\[
\begin{align*}
g_1 := & \tau: 1 \quad \text{shri} \quad \text{dest} \quad \text{rd}0 \quad \{0 \leq j < 5\} \\
g_2 := & \tau: 2 \quad \text{andi} \quad \text{srcA} \quad \text{rd}0 \quad \{0 \leq j < 6\} \\
\end{align*}
\]

First, consider the polyhedral time offset \( g_1 \). Its time offset \( \tau = 1 \) is smaller than the initiation interval \( \pi = 2 \) and therefore does not overlap into future iterations. This is
6.2 Control flow analysis

visualized by the green boxes and the blue box in Figure 6.3. Now consider the polyhedral time offset \( g_2 \): Its condition space, by definition, is the union of all condition spaces of its leaves:

\[
\text{domain}(g_2) = \bigcup_{g' \in \text{leaves}(g_2)} \text{domain}(g') = \{0 \leq j < 6\}
\]

That means the instruction represented by \( g_2 \) is executed in iterations \( 0 \leq j < 6 \) with a time offset of \( \tau = 2 \). Since the time offset is not smaller than the initiation interval \( \pi = 2 \), the instruction overlaps into the future \( \tau / \pi = 1 \) iteration(s). For example, the iteration after \( j = 0 \) is \( j = 1 \), the one after \( j = 5 \) is \( j = 6 \). We can therefore shift \( g_2 \)'s domain to represent the kernels it is executed in:

\[
\text{succ}(\text{domain}(g_2), 1) = \{1 \leq j < 7\}
\]

The instruction is therefore executed in slot \( \tau \mod \pi = 0 \) of all kernels of iterations \( \{1 \leq j < 7\} \). This is visualized by the red boxes in Figure 6.3.

\[\triangle\]

**Example.** For multi-dimensional examples—that is, for the vast majority of real-world applications—the successor relation is more complicated. For example, consider a condition space \( J' = \{0 \leq j_1 < 16 \land j_2 = 0\} \).

Let \( s = (\sigma(1, 2), \omega = 1) \) be the scanning order associated with the schedule from the selected symbolic configuration (first in positive \( j_1 \)-direction, then in positive \( j_2 \) direction). Assuming the intra-tile iteration space is

\[
J = \{0 \leq j_1 < 16 \land 0 \leq j_2 < 16\},
\]

then:

\[
\text{succ}(J', 1) = \{(1 \leq j_1 < 16 \land j_2 = 0) \lor (j_1 = 0 \land j_2 = 1)\}
\]

This is because the temporally next iteration after \( j = (15, 0)^T \) is \( j = (0, 1)^T \) according to the scanning order \( s \).

\[\triangle\]

Using this knowledge, we *fold* all children of \( f_{pc, fu} \) into \( \pi \) slots to obtain a transformed polyhedral syntax tree \( f'_{pc, fu} \) that does not describe the sequence of assembly instructions issued at the start of iteration \( j \), but that instead describes the \( \pi \) slots of the *kernel* issued at the start of iteration \( j \). The folding operation is elaborated in Algorithm 2 and visualized in Figure 6.3.

Folding introduces “iterations” \( j \) where \( j \notin J \), thus pseudo-iterations that contain the epilog of the pipelined functional unit program; we therefore call \( E_g = J_g \setminus J \) the *epilog space* of \( g \). In the following, \( J_{pc, fu} = J \cup E_{f'_{pc, fu}} \) denotes intra-tile iteration space including the epilog space and \( E \) the union of all individual epilog spaces.

**Example** (Bit extraction, continuing from p.126). After folding \( f_{l, alu1} \), we obtain a transformed tree where the second level represents the slot index instead of the time offset \( \tau \):

127
Algorithm 2 Given $\pi$, fold PST $f_{fu}$ representing a functional unit program into $f'_{fu}$

$G \leftarrow \emptyset$

for $g \in \text{children}(f_{fu})$ do

$G' \leftarrow \text{children}($

$\text{offset}(g, \lfloor \tau[g]/\pi \rfloor))$

$g' \leftarrow \left(\bigcup_{g'' \in G'} \text{domain}(g''), (\tau[g] \mod \pi), G'\right)$

> new node with slot index as attribute

$G \leftarrow G \cup \{g'\}$

end for

$f'_{fu} \leftarrow (\bigcup_{g \in G} \text{domain}(g), \text{attr}(f), G)$

> condition space of a parent node is union of children's

function $\text{offset}(f, n)$

> recursively offset condition spaces in $f$ by $n$ iterations

if $\text{children}(f) = \emptyset$ then

return $(\text{succ}(\text{domain}(f), n), \text{attr}(f), \emptyset)$

> copy $f$, but with offset condition space

else

$G \leftarrow \emptyset$

for $g \in \text{children}(f)$ do

> offset children condition spaces recursively

$G \leftarrow G \cup \{\text{offset}(g, n)\}$

end for

return $(\bigcup_{g \in G} \text{domain}(g), \text{attr}(f), G)$

> copy $f$, but with offset children

end if

end function
6.2 Control flow analysis

Figure 6.3: (From [WWH+]) Formation of kernels in \( f_{1, alu} \) with \( \pi = 2 \). Each column represents an iteration \( 0 \leq j \leq 6 \) (\( j = 6 \) being a "pseudo-iteration" corresponding to the epilog) and the boxes within a column the instructions executed in the \( \tau = 2 \) time steps until the next iteration starts. Above, the instructions are arranged according to the evaluation \( f_{1, alu} ((j)) \). In particular, the instruction with time offset \( \tau = 2 \) overlaps with iteration \( j + 1 \), having its condition space shifted from \( \{0 \leq j \leq 5\} \) to \( \{1 \leq j \leq 6\} \) and giving rise to the epilog space \( \mathcal{E} = \{j = 6\} \). Below, the same instructions are depicted after being arranged into kernels corresponding to the evaluation \( f'_{1, alu} ((j)) \) of the folded PST (see main text). A kernel class \( Q_i \) is a set of iterations issuing the same kernel. For example, \( Q_2 \) is red \( \cap \) green \( \cap \) blue (color corresponds to box color).

For example, the andi instruction, which was originally at offset \( \tau = 2 \), now resides at slot 0, but with a shifted condition space that reflects the overlapping into the next iteration (compare Figure 6.3). This makes the pipelined program’s epilog space \( \mathcal{E}_{1, alu} = \{j = 6\} \). △

The folded polyhedral syntax tree \( f'_{pc, fu} \) of a functional unit gives rise to a set \( QC_{pc, fu} \) of kernel classes, that is a partition of \( \mathcal{F} \cup \mathcal{E}_{pc, fu} \) into subsets \( Q_{pc} \) of iterations in which the same kernel is issued. These are determined analogously to processor classes using Algorithm 1.

Example (Bit extraction, continuing from p. 127). For \( f'_{1, alu} \), we obtain 4 kernel classes:

\[
QC_{1, alu} = \{Q_1 = \{ j = 0 \}, Q_2 = \{ 1 \leq j < 4 \}, Q_3 = \{ j = 4 \}, Q_4 = \{ j = 5 \} \}
\]

The kernel \( q_1 \) of \( Q_1 \), for example, is obtained by evaluating \( f'_{1, alu} \) at any \( j \in Q_1 \):
6 Instantiation

\[ f'_{1,alu1}(0) := \text{shift} \rightarrow \text{slot} \rightarrow 1 \rightarrow \text{shri} \rightarrow \text{srcA} \rightarrow \text{rd}\_0 \rightarrow \text{srcB} \rightarrow 1 \]

Converted into an instruction sequence, this evaluation yields the following kernel (slot index in brackets):

\[
\begin{align*}
[0] & \text{nop} \\
[1] & \text{shri rd1 rd0 1}
\end{align*}
\]

(Note that slot 0 has no associated node in \( f'_{1,alu1} \)—we assume an implied nop in such cases.)

6.2.3 Generation of control flow graphs

Finally, because all functional units of a PE execute individual programs in parallel, an individual control flow graph \( CFG_{pc,\text{fu}} \) is constructed from the set of kernel classes \( QC_{pc,\text{fu}} \) for each processor class \( P_{pc} \) and functional unit \( \text{fu} \) using Algorithm 3. The algorithm inserts a node for each kernel class and an edge \( e \) between each pair of kernel classes \( Q_i \) and \( Q_j \) between which control flow passes from \( Q_i \) to \( Q_j \). The edge \( e \) is annotated with the transition space \( T \), that is, the set of iterations \( j \) where control flow passes from \( Q_i \) to \( Q_j \). Figure 6.4 shows the CFG for processor class \( P_1 \) and functional unit \( alu1 \) of the Bit extraction example.

**Algorithm 3** Generate control flow graph \( CFG \) from kernel classes \( QC \) and PST \( f' \)

\[
\begin{align*}
V & \leftarrow \{v_1, v_2, \ldots, v_{|QC|}\}, E \leftarrow \emptyset \quad \triangleright \text{one node for each kernel class} \\
\text{for } Q_i \in QC & \text{ do} \\
q[v_i] & \leftarrow f'(Q_i), Q[v_i] \leftarrow Q_i \quad \triangleright \text{annotate kernel and kernel class to node} \\
\text{for } Q_j \in QC & \text{ do} \\
T & \leftarrow Q_j \cap \text{succ}(Q_i, 1) \quad \triangleright \text{(note: self edges represent repetition)} \\
\text{if } T & \neq \emptyset \text{ then} \\
E & \leftarrow E \cup \{e = (v_i, v_j)\} \quad \triangleright \text{insert edge} \\
T[e] & \leftarrow T \quad \triangleright \text{annotate transition space} \\
\text{end if} & \\
\text{end for} & \\
\text{end for} & \\
\text{CFG} & \leftarrow (V, E)
\end{align*}
\]

The constructed control flow graph \( CFG_{pc,\text{fu}} = (V, E) \) represents the functional unit program to be generated: In any iteration \( j \in J_{pc,\text{fu}} \) there is exactly one node \( v \in V \) where \( j \in Q[v] \), representing the kernel \( q[v] \) to be started at time step \( t(j) \). Among its outgoing edges, there is either exactly one edge \( e = (v, w) \in E \) where \( j \in T[e] \), or it is the temporally last kernel to be executed. Node \( w \in V \) represents the kernel \( q[w] \) issued
6.3 Control signal allocation

For each iteration \( j \in J_{\text{pc, fu}} \), there is exactly one node \( v \in V \) of \( \text{CFG}_{\text{pc, fu}} = (V, E) \) with \( j \in Q[v] \). The kernel represented by that node has \( \deg(v) \) branch targets, but only one target \( w \) will be branched to: the only one with \( j \in T[(v, w)] \). Consequently, some entity—in the case of a TCPA, the global controller (GC)—must track the current iteration and signal to the PEs for which edges \( e \in E^+(v) \) the current iteration \( j \) is in their transition space \( T[e] \) in order to select to which target to branch. It does this by

Figure 6.4: (From [WWH+]) Control flow graph for processor class \( P_1 = \{k < 2\} \) and functional unit \( \text{alu1} \) in the running example. Each node represents a kernel, and each edge is annotated with its transition space \( T \), that is, in which iterations the branch represented by the edge is taken (note that it is taken in the time step \( \pi - 1 \) relative to the start time of the kernel). For kernels with more than one outgoing edge, the global control signals \( CS \) and assignments \( c \) are also annotated (Section 6.3).

Remark. Algorithm 3 generates CFGs with arbitrary outdegree, that is, an arbitrary number of branch targets. While TCPAs support a configurable number of simultaneous branch targets, it is usually set to 2 or other low numbers. Therefore, the outdegree of the control flow graph must be reduced accordingly; some solutions are presented in [Bop15]. Note that program size is traded off against the complexity of the transition spaces. There are two extremes: (1) No node in the control flow graph is duplicated and the program size is minimal, but the transition spaces are complex and require many control signals with complex conditions (that is, arbitrary Presburger sets); (2) The number of control signals is minimized, using only simple conditions, but many nodes in the control flow graph are duplicated to split complex conditions into simple ones (for example, a union is split into its convex constituents and divided among multiple edges and thus nodes). Finding an optimum given certain criteria (for example, satisfying both instruction memory size and control signal constraints) while still keeping time efficiency in mind is considered future work.

6.3 Control signal allocation

in the next iteration. Hence, the outgoing edges of \( v \) represent the set of branch targets and the transition spaces the branch conditions. Next, these branch conditions need to be encoded using a set of control signals.

Remark. Algorithm 3 generates CFGs with arbitrary outdegree, that is, an arbitrary number of branch targets. While TCPAs support a configurable number of simultaneous branch targets, it is usually set to 2 or other low numbers. Therefore, the outdegree of the control flow graph must be reduced accordingly; some solutions are presented in [Bop15]. Note that program size is traded off against the complexity of the transition spaces. There are two extremes: (1) No node in the control flow graph is duplicated and the program size is minimal, but the transition spaces are complex and require many control signals with complex conditions (that is, arbitrary Presburger sets); (2) The number of control signals is minimized, using only simple conditions, but many nodes in the control flow graph are duplicated to split complex conditions into simple ones (for example, a union is split into its convex constituents and divided among multiple edges and thus nodes). Finding an optimum given certain criteria (for example, satisfying both instruction memory size and control signal constraints) while still keeping time efficiency in mind is considered future work.
generating (binary) control signals that encode this information for all outgoing edges of all nodes and all generated control flow graphs.

**Definition 6.4 ([WWH⁺]).** A control signal is a function

\[
\text{cs}(j) : \mathcal{J} \cup \mathcal{E} \mapsto \{0, 1, -\}
\]

that maps an intra-tile iteration \(j\) to 0, 1, or don’t-care (represented by \(-\)). A control signal is called **partial** if it maps at least one \(j\) to \(-\).

To better contextualize this in terms of hardware: This means that the global controller outputs either a 0 or a 1 (for don’t-cares, it does not matter which) for each configured control signal, depending on the current intra-tile iteration \(j\). The value of the control signal is then passed by each PE to its direct neighbors, thus propagating the control signal across the TCPA to steer the control flow of the individual programs using branch instructions (see below).

For each node \(v \in \mathcal{V}\), there are \(N_v = \lceil \log_2 \deg^+(v) \rceil\) control signals required to encode the \(\deg^+(v)\) outgoing edges of \(v\). Each outgoing edge is given a unique assignment \(c[e] \leftarrow (c_1, \ldots, c_{N_v})\) with \(c_i \in \{0, 1, -\}\) such that these assignments do not overlap for any two outgoing edges of \(v\). These assignments can, for example, be determined using binary decision diagrams [Bop15]. From these assignments, for each node \(v\), we build \(N_v\) partial control signals:

\[
\forall v \in \mathcal{V}, 1 \leq i \leq N_v : \text{cs}_{v,i}(j) := \begin{cases} 
1 & \text{if } \exists e = (v, w) \in \mathcal{E} : c_1[e] = 1 \land j \in \mathcal{T}[e] \\
0 & \text{if } \exists e = (v, w) \in \mathcal{E} : c_1[e] = 0 \land j \in \mathcal{T}[e] \\
- & \text{else}
\end{cases}
\]

Intuitively, this control signal is 1 whenever the outgoing edge \(e\) with \(j \in \mathcal{T}[e]\) (recall that there can only be one) has a 1 in its assignment at the given index; it is 0 when the assignment has a 0 at the given index. If it has a don’t care at the given index or there is no outgoing edge with \(j \in \mathcal{T}\), then the value of the control signal does not matter (don’t-care).

**Example** (Bit extraction, continuing from p. 127). Only node \(q_2\) in Figure 6.4 requires a control signal because it is the only node with more than one outgoing edge. First, we need to give its two outgoing edges unique assignments \(c\), for example

\[
c[(q_2, q_2)] \leftarrow (0) \text{ and } c[(q_2, q_3)] \leftarrow (1).
\]

Since there are only two outgoing edges, \(N_{q_2} = \lceil \log_2 2 \rceil = 1\) “bit” suffices to distinguish them and no don’t-cares are required. Next, \(N_{q_2} = 1\) control signal local to \(q_2\) is constructed:

\[
\text{cs}_{q_2,1} := \begin{cases} 
0 & \text{if } j \in \mathcal{T}[(q_2, q_2)] \equiv 0 \leq j < 4 \\
1 & \text{if } j \in \mathcal{T}[(q_2, q_3)] \equiv j = 4 \\
- & \text{else}
\end{cases}
\]
6.3 Control signal allocation

We can interpret this control signal within the context of program execution as follows: Assume the kernel given by \( q_2 \) is currently being executed. In the last cycle of its execution, if the control signal is 0, then kernel \( q_2 \) is repeated, if it is 1, executed passes to kernel \( q_3 \). The control signal cannot be \(-\) (don’t-care) in this case because then, \( q_2 \) would not be currently executing.

Across all control flow graphs \( CFG_{pc,fu} \), there is now a large set of partial control signals \( cs_{pc,fu,v,i} \) that is local to the corresponding node \( v \). The number of these local control signals usually far exceeds the maximum number \( C \) of control signals supported by the global controller, which we call global control signals. We therefore combine the local, partial control signals using an interference graph where each control signal \( cs_{pc,fu,v,i} \) is a node and two control signals \( cs_1 \) and \( cs_2 \) interfere if

\[
\exists j \in J \cup E : cs_1(j) \neq cs_2(j) \land cs_1(j) \neq - \land cs_2(j) \neq -.
\]

A \( C \)-coloring of the vertices then corresponds to the allocation of \( C \) global control signals \( cs_i \). For each \( cs_i \), the global controller is configured with its one-domain, that is, the subset of \( J_{pc,fu} \) where \( cs_i(j) = 1 \); for all other iterations 0 is output. Additionally, each node \( v \) in each control flow graph is annotated with the \( N_v \)-tuple \( CS[v] \) of global control signals that were allocated for the node’s local control signals; this information is necessary for program generation, in particular branch condition generation.

Example (Bit extraction, continuing from p. 132). The local control signal from the previous example is represented by the following global control signal:

\[
cs_1(j = (j)) := \begin{cases} 
1 & \text{if } j = 4 \\
0 & \text{else} 
\end{cases}.
\]

It is annotated to the control flow graph node: \( CS[q_2] \leftarrow (cs_1) \).

In terms of hardware, the global controller is configured to have one active control signal that outputs 1 if \( j = 4 \), and 0 otherwise. This value is then propagated using the control layer of the interconnect network across all 3 allocated PEs, starting from PE \( k = 0 \), as visualized in green in Figure 6.1. Note that each control signal is delayed in cycles by exactly the same amount of cycles neighbor PEs are delayed in their execution of the same branches according to a given schedule. By construction, the correct synchronization of the control flow of multiple PEs executing the same program, but at a delayed number of cycles, is thereby guaranteed.

Note that for instantiation at run time, fast heuristics such as greedy graph coloring are preferable because graph coloring in general is NP-complete. We consider it future work to investigate if control signal allocation can be—at least partially—performed at compile time. After control signal generation, the generated control flow graphs now contain all information necessary for program generation.
6 Instantiation

6.4 Program generation

Each processor class \( \mathcal{P}_{pc} \) requires the generation of one PE program, which is simply a container for the functional unit programs in that processor class. Generating a PE program therefore requires generating the program for each functional unit from its control flow graph \( CFG_{pc, fu} \).

In orthogonal instruction processing (Section 2.1), each instruction in a functional unit program is a pair of a functional instruction, specifying the operation, and a branch instruction, specifying the control flow. While the functional instructions are explicit in the kernels \( q[v] \) annotated to the nodes in \( CFG_{pc, fu} \), corresponding branch instructions remain to be generated. The instructions in slots \( 0 \ldots \pi - 2 \) of a kernel \( q[v] \) are each combined with a next branch instruction because each kernel is executed atomically and in order. However, for the last instruction, the one in slot \( \pi - 1 \), a conditional multi-target branch instruction must be generated that selects the target according to the allocated global control signals \( CS[v] = \{cs_1, \ldots, cs_{N_v}\} \) and the assigned values \( c[e] \) for all outgoing edges \( e \in E^+(v) \). The subsequent concatenation of all kernels (starting with \( q[v] \) where \( v \) is the start node, that is, has no incoming edges) yields the functional unit program. Algorithm 4 summarizes these two steps.

**Example** (Bit extraction, continuing from p. 133). We obtain the following functional unit program for functional unit \( alu1 \) in processor class \( \mathcal{P}_1 \) (the comments show some applicable optional simplifications):

\[
\begin{align*}
q1: & \text{nop} / \text{next} \\
    & \text{shri rd1 rd0 1 / goto q2} \quad \text{// can be simplified to 'next'} \\
q2: & \text{andi od0 rd1 1 / next} \\
    & \text{shri rd1 rd0 1 / if ic0 jmp q3, q2} \\
q3: & \text{andi od0 rd1 1 / next} \\
    & \text{shri od1 rd0 1 / goto q4} \quad \text{// can be simplified to 'next'} \\
q4: & \text{andi od0 rd1 1 / next} \\
    & \text{nop / halt} \quad \text{// can be merged into previous instruction}
\end{align*}
\]

The programs for \( alu0 \) (not shown) and \( alu1 \) together form the PE program for \( \mathcal{P}_1 \).

6.5 I/O access instantiation

The last step is the instantiation of configuration data for the I/O buffers and address generators (Section 2.1) from the access mappings \( A \). Recall that an access mapping \( a = (\text{reg}, x, \alpha, \mathcal{A}) \) maps accesses within iterations \( i \in \mathcal{A} \) to \( x[\alpha(i)] \) to \( \text{reg} \). We call a particular access in an iteration \( i \) an access instance \( a[i] \).

**Example** (Bit extraction, continuing from p. 134). The access mapping \( a_{\text{bits}} \) maps all write accesses to \( od0 \) with \( i < N \) to \( \text{bits} [Q_{\text{bits}} i + d_{\text{bits}}] \), that is, \( \text{bits} [i] \). These write accesses
Algorithm 4 Generate functional unit program from annotated $CFG = (V, E)$

```plaintext
program ← []
for $v \in \text{topological\_sort}(V)$ do  # begin with start node
  targets ← $\{w \mid (v, w) \in E\}$
  for slot := 0 to $\pi - 1$ do
    if slot = $\pi - 1$ then  # if it is the last instruction in the kernel
      branch ← MAKE_BRANCH($v$, targets)  # make conditional branch
    else
      branch ← next  # otherwise, go unconditionally to the next instruction
    endif
    instr ← ($q[v].\text{instruction}[slot]$, branch)
    append(program, instr)
  endfor
endfor

function MAKE_BRANCH($v$, targets)
if $|\text{targets}| = 0$ then
  return halt  # last node → halt execution
elseif $|\text{targets}| = 1$ then
  return goto target  # target is the single element of targets
elseif $|\text{targets}| > 1$ then
  return if $cs_1[v], \ldots, cs_{N_v}[v] \text{ jmp target}(\text{targets}, 2^{N_v} - 1), \ldots, \text{target}(\text{targets}, 0)$  # target(\text{targets, i}) gives $w$ such that $e[(v, w)] = i$
endif
end function
```
correspond to instruction `andi od0 rd0 1` in the previous examples: Each time `od0` is written, the value is stored in `bits[i]`.

Tiling distributes access instances `a[(j, k)^T]` in `A^* ⊆ K_a ⊕ J` across multiple PEs `K_a`, possibly making them concurrent since the PEs run in parallel. Consequently, for each `k ∈ K_a` of each access mapping `a ∈ A`, two parts must be instantiated:

1. A connection between an I/O buffer bank and the port corresponding to reg of PE `k`, which entails finding a free bank and a route between the bank and the PE on the interconnect.

2. The configuration of the allocated memory bank’s address generator, consisting of the coefficients of an affine address function derived from `a` and the memory layout of `x` that maps the intra-tile iteration `j` to an address, and the intra-tile domain `\tilde{J} = \text{split}(k, A^*)` in the access space, required to generate an enable signal.

The generation of the above two parts makes the time complexity of this step linear in the number of involved PEs `|K_a|` in the general case. However, if input and output variables are only accessed at the borders of the processor array, routing becomes unnecessary because the border PEs have a direct connection to the I/O banks. This allows the compiler to generate the above two parts already at compile time.

**Example** (Bit extraction, continuing from p. 134). See Figure 6.1.

**Example** (Matrix multiplication, continuing from p. 121). Using the given specification of the matrix multiplication algorithm, all allocated PEs require output access to individual memory banks to write the output variable `C` since `A_C = I`. This not only makes I/O access generation linear in the number of PEs, which we want to avoid, but also restricts the maximum number of allocated PEs because there is only a limited number of I/O buffer banks and interconnect routes. One solution is to transform the UDA such that the output only takes place at one of the borders, for example by propagating the results of `C` at the end. Another solution is to choose a 1-dimensional space mapping; however, this decreases parallelism.

After I/O access instantiation, all concrete configuration data to configure the target TCPA and execute the given loop has been created.

### 6.6 Experiments and discussion

First, note that we are not interested in evaluating the quality of the instantiated mapping in terms of performance metrics because we already gave optimality results in Section 4.6 and the instantiated PE programs obey both the symbolic schedules and the register allocation, meaning instantiation maintains optimality.
We rather experimentally show that the time complexity of program instantiation does not in general depend on the number of PEs and that a symbolic configuration is a space-efficient representation. In particular, we perform symbolic mapping and instantiation on the following UDAs: bit extraction, FIR filter, matrix-matrix multiplication, and a convolutional layer within a CNN (6-dimensional loop). The choice is based on the intention to cover a variety of both application domains and dimensionality. For each symbolic configuration, we instantiated six concrete configurations corresponding to six tilings: three resulting in a 1-dimensional region of PEs (1, 16, and 32 PEs), and three resulting in a 2-dimensional region (4 × 4, 8 × 8, and 32 × 32)\(^2\). For each instantiation run, we measured the mean execution time of program instantiation and the size of the concrete configuration.

Since we want to show the independence of the compilation times from the problem size—that is, number of available PE and the loop bounds—we are mainly interested in normalized execution times, summarized in Table 6.1. Each row represents one of the examples and contains, for each of the six tilings, the execution time of program instantiation normalized to the execution time of program instantiation in the case of a single tile (1 PE). The number of resulting processor classes is given in parentheses. As clearly visible, the execution time of program instantiation is roughly linear in the number of processor classes and not in the number of PEs, as is, for example, evident for the matrix multiplication example: Program instantiation for both 4 × 4 = 16 and 32 × 32 = 1024 PEs takes about equally as long because both have two processor classes, meaning two programs need to be instantiated. The instantiation phase, therefore, effortlessly scales to the ever-increasing number of PEs. (Note that some tilings may result in more complex control flow analysis, as is, for example, seen in the convolution example, where instantiation for 4 × 4 PEs takes about four times as long as for 1 × 16 PEs, despite having only double as many processor classes. However, it is still independent of the number of PEs.)

Table 6.2 shows the size of each concrete configuration normalized to the size of the symbolic configuration it was instantiated from. Excluding the bit extraction example, all concrete configurations by themselves were already larger than the symbolic configuration. Consequently, run-time instantiation significantly saves memory even if only a small number of concrete configurations were necessary at run time. For example, storing the symbolic configuration for the CNN example saves about 95% space compared to storing both concrete configurations for 4 × 4 and 8 × 8 PEs.

Finally, let us qualitatively compare symbolic loop compilation to techniques for mapping loops onto CGRAs because it also involves NP-hard problems, in particular modulo scheduling. In relevant state-of-the-art works, such as HyCube [KMM+17], compilation times are reported in the range of minutes to hours, depending on the complexity of the loop program and architecture. With our approach, although not directly comparable—our parallelization technique is vastly different by exploiting not

\(^2\)Both loop bounds and tile sizes were chosen appropriately to result in these PE regions.
### 6 Instantiation

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1D (# PEs)</th>
<th>2D ($R \times C$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\pi$</td>
<td>1</td>
</tr>
<tr>
<td>Bit extract (1D)</td>
<td>2</td>
<td>1.00 (1)</td>
</tr>
<tr>
<td>FIR filter (2D)</td>
<td>2</td>
<td>1.00 (1)</td>
</tr>
<tr>
<td>Matrix multiplication (3D)</td>
<td>2</td>
<td>1.00 (1)</td>
</tr>
<tr>
<td>CNN Convolution (6D)</td>
<td>1</td>
<td>1.00 (1)</td>
</tr>
</tbody>
</table>

Table 6.1: Relative execution times of program instantiation for a set of mappings of various loop programs. Each row represents a symbolic configuration of the listed algorithm and each column the instantiation for one of six tilings, three resulting in a one-dimensional and three in a two-dimensional PE allocation. For each instantiation, the execution time relative to the execution time of the first column is listed; the number of processor classes is listed in parentheses. The table clearly shows that the time complexity of instantiation is roughly proportional to the number of processor classes and not to the number of processing elements.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1D (# PEs)</th>
<th>2D ($R \times C$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\pi$</td>
<td>1</td>
</tr>
<tr>
<td>Bit extract (1D)</td>
<td>2</td>
<td>0.77</td>
</tr>
<tr>
<td>FIR filter (2D)</td>
<td>2</td>
<td>1.17</td>
</tr>
<tr>
<td>Matrix multiplication (3D)</td>
<td>2</td>
<td>1.00</td>
</tr>
<tr>
<td>CNN Convolution (6D)</td>
<td>1</td>
<td>1.05</td>
</tr>
</tbody>
</table>

Table 6.2: The size of each generated concrete configuration normalized to the size of the symbolic configuration it was instantiated from.

only innermost loop parallelism, but also potential parallelism in all outer loops—these potentially minutes to hours to determine a high-quality mapping are spent only once during symbolic mapping at compile time. For any mapping with different parameter values, only instantiation needs to be performed, whose execution time is negligible (in the range of milliseconds to seconds, depending on the complexity of the loop program) and, as shown, scales well with the number of PEs. By contrast, not using symbolic loop compilation requires solving the NP-hard problems over and over again, accruing hours and hours.
6.7 Related work

We know of no other work that can generate symbolic code for loop accelerators—which rely on cycle-accurate scheduling—whose tiling and size can be chosen at run time.

Run-time code generation is often mentioned in the context of so-called auto tuning.\(^3\) Auto tuning optimizes mapping parameters such as tile sizes by repeatedly running a given loop and measuring performance (or other metrics), in the end only keeping the parameter values that lead to the best performance. Making it possible to change the mapping at run time allows auto tuning approaches to drastically improve their execution time. One example based on the polyhedral model is APOLLO [IoS16], as briefly introduced in Section 5.5. But not only auto tuning, but utilizing run-time information in general has been shown to be beneficial for performance [SAG+13]. For example, PolyJIT [SAG+19] combines the usage of run-time knowledge (such as the number of available threads) with multi-versioning and just-in-time compilation to dynamically optimize polyhedral loop nests. Multi-versioning describes approaches where multiple versions of a code region are compiled at compile time and the “best” one is selected conditionally at run time (see, for example, [JLC11]). However, all of the mentioned approaches target general-purpose processors.

Generation of loop code for a number of processors unknown at compile time for targets other than TCPAs or CGRAs has been the focus of several other works. Konstantinidis et al. [KKR+14] generate parallelized code for GPUs by symbolically tiling a loop at compile time compiling CUDA kernels at run time. Tensor comprehensions [VZT+19] use a just-in-time compiler to generate optimized CUDA code for GPUs from DAGs that represent deep learning kernels. DynTile [HBR+10] targets general-purpose multi-cores by symbolically tiling a loop at compile time and then scheduling and executing the pre-compiled programs at run time. However, none of these approaches apply to processor arrays like TCPA because they provide neither cycle-accurate scheduling, tight synchronization, nor compact tile-specific programs. Instead, they rely on features of their target architectures to circumvent these problems.

\(^3\)An early example is ATLAS [WD98], which accelerates linear algebra operations and is often cited when explaining auto tuning.
Run-time Requirement Enforcements of Loops

Run-time requirement enforcement is an adaptive technique that leverages flexibility in application mapping to guarantee bounds on non-functional properties, such as latency and energy consumption. For example, by adjusting the number of allocated PEs of a loop application at run time, a certain latency may be achieved while minimizing energy consumption. As such, run-time requirement enforcement is a natural application of symbolic loop compilation.

However, this flexibility is not only a boon, but also a curse: Given requirements and input data, how many processing elements are best to guarantee the requirements and optimize other non-functional properties? What functional units? What schedule should be used? Or even, which frequency should the TCPA run at? Since run-time requirement enforcement is, as the name implies, performed at run time by an enforcer, the non-functional properties of the enforcer itself must be taken into account. However, the enforcer has to comb through a huge search space to find the proverbial needle in a haystack. As a compromise, we propose two approaches to reduce the size of the search space: architecture-specific reduction, where we compute part of the mapping at run time from a static part of the mapping, and coverage-based reduction, where we select a subset of mappings at compile time that maximizes the probability that a set of given requirements will be satisfied at run time.

Roadmap. First, Section 7.1 formalizes run-time requirement enforcement with a focus on loop applications. Next, two complementary techniques for reducing the search space are proposed: architecture-specific reduction in Section 7.2, and coverage-based reduction in Section 7.3. Finally, we motivate the effectiveness of the two reduction approaches in a case study in Section 7.4 to base further work on.

7.1 Run-time requirement enforcement on loops

We consider requirements as absolute, interval-constrained bounds on an application’s non-functional properties, similar to [TGR*16]. Examples for loop applications include latency requirements (the loop must execute within 20 ms), throughput requirements (the loop must process at least 25 frames per second), and energy requirements (the loop
must use at most a certain amount of energy). Formally, we define a requirement on a loop program as follows.

**Definition 7.1** (Adapted for loop programs from [TGR+16]). Let \( p \) denote a non-functional property of the execution of a loop program \( \mathcal{L} \). A requirement \( r_p \) on \( p \) is an interval or corridor

\[
r_p := [p_{\text{lower}}, p_{\text{upper}}].
\]

Now, the actual values of a non-functional property \( p \) usually depend both on the loop bounds \( b = u - l \) (upper minus lower bounds\(^1\) in each dimension) and its mapping \( m \) onto the target architecture; we denote this dependency by \( p(m, b) \).

**Definition 7.2** ([WT18]). A loop program mapping \( m = (m_1, m_2, \ldots) \in M \subset M_1 \times M_2 \times \ldots \) is a tuple of mapping parameters (such as allocation, schedule, frequency).

**Example.** For TCPAs, we model a mapping as a tuple \( m = (\mu, \hat{R}, \hat{C}, f) \) of a symbolic mapping \( \mu \in M_{\text{sym}} \) (Section 4.7), a PE allocation \( \hat{R} \times \hat{C} \), and a frequency \( f \in F \) where \( F \) is the set of frequencies the target TCPA supports. To recapitulate: A symbolic mapping \( \mu \) contains both a space mapping \( \phi \) and a symbolic schedule \( (\lambda^*, \tau) \).

**Example.** A commercial processor array is the MPPA-256 from Kalray [dDAB+13]. The MPPA-256 is subdivided into 16 clusters with 16 cores each and also provides programmable I/O-clusters. A mapping could therefore be a tuple \( (\gamma_1, \gamma_2, \ldots, \iota) \), where \( \gamma_i \) is the number of allocated cores in cluster \( i \), and \( \iota \) the number of allocated I/O-clusters.

**Definition 7.3** ([WT18]). A mapping \( m \) of a loop program \( \mathcal{L} \) satisfies a requirement \( r_p \) for bounds \( b \), formally \( m \models_b r_p \), if \( p(m, p) \in r_p \) for the entire execution of \( \mathcal{L} \) with bounds \( b \). A set of requirements \( R = \{r_{p_1}, r_{p_2}, \ldots\} \) is satisfied if all its requirements are satisfied:

\[
m \models_b R \iff \forall r \in R: m \models_b r.
\]

This definition implies the capability to determine a non-functional property’s value from the mapping \( m \) and loop bounds \( b \). For example, for a latency requirement on TCPAs, we can use the precomputed linear schedule \( \lambda^* \); other architectures might use the result of WCET analyses or similar.

To enforce that a loop program satisfies a set of given requirements for bounds \( b \), we must find a mapping at run time that satisfies them.

---

\(^1\)Note that for non-rectangular iteration spaces, \( u \) and \( l \) are the bounds of the rectangular hull of the iteration space.
7.1 Run-time requirement enforcement on loops

Definition 7.4 ([WT18]). Run-time requirement enforcement on loops involves determining a mapping $m$ of a loop program $L$ at run time such that $m \models_b R$ for a given set $R$ of requirements and loop bounds $b$:

\[
\text{given } L, R, b \\
\text{find } m \models_b R
\]

An entity performing run-time requirement enforcement on loops is called a (loop) enforcer.

An enforcer may freely choose among all mappings that satisfy $R$. This freedom can be leveraged to optimize other non-functional properties $P$. A prime example is energy—the program must stay within the latency corridor and consume as little energy as possible.

Definition 7.5 ([WT18]). Run-time requirement enforcement on loops with secondary objectives describes run-time requirement enforcement methods on loops that, while satisfying a set of requirements $R$, also minimize a set of other non-functional properties $P = \{p_1, p_2, \ldots\}$ according to objective functions $f_1, f_2, \ldots$:

\[
\text{given } L, R, P, b \\
\text{minimize } f_1(p_1, m, b), f_2(p_2, m, b), \ldots \\
\text{such that } m \models_b R
\]

If we limit the enforcement problem to the selection of a mapping out of a discrete set $M$ of all available mappings at run time (some might be unavailable because resources might have been allocated by other applications), we can formalize the notion of enforcement for loops as follows.

Definition 7.6 ([WT18]). Let $L$ be an $n$-dimensional loop program with requirements $R$, secondary objectives $P$, and a set of valid mappings $M$. An enforcement function for $L$, $R$, and $P$ is a function $e : B \mapsto M$ that maps loop bounds $b$ to a valid mapping $m \in M$ such that

\[
m \models_b R \land m = \arg\min_{m \in M} (f_1(p_1, m, b), f_2(p_2, m, b), \ldots).
\]

A naive algorithm realizing an enforcement function could look like this:

**Static inputs:** $L$, $R$, $P$; **run-time inputs:** $M$, $b$

**Output:** $m$

$M_{\text{feasible}} \leftarrow \emptyset$

for all $m \in M$ do

if $m \models_b R$ then

$M_{\text{feasible}} \leftarrow M_{\text{feasible}} \cup m$
First, all mappings from $M$ that satisfy $R$ are collected into $M_{\text{feasible}}$. From these mappings, a mapping that optimizes the secondary objectives $P$ is then selected (the optimization method “optimize” for $P$ can be freely chosen). Note that how to react to non-satisfiable requirements is implementation-specific.

**Example.** As a more concrete example of an enforcer for TCPAs, let us look at the inverse problem of Section 4.7. There, we investigated how to find the latency-minimal mapping $m$ among a set of symbolic mappings $M$. Here, we instead want to find a symbolic mapping $\mu \in M_{\text{sym}}$ and a region of PEs $\tilde{R} \times \tilde{C}$ that satisfy a given upper bound $L_{\text{upper}}$ on the latency $L$ given the loop bounds $b$:

$$m' := (\mu, \tilde{R}, \tilde{C}) \models_b [0, L_{\text{upper}}],$$

where we also call $m'$ a mapping (as opposed to a symbolic mapping) and we assume that the worst-case execution times for enforcement and configuration of the target TCPA have already been subtracted from $L_{\text{upper}}$. As one possible solution, Algorithm 5 iteratively determines the minimum number $\tilde{R} \cdot \tilde{C}$ of PEs to satisfy the given requirement by evaluating the latency function $L$ (Section 4.4). The algorithm iterates over the available PE allocations $\tilde{R}, \tilde{C}$, ordered in ascending number of PEs $\tilde{R} \cdot \tilde{C}$, and then over the set of symbolic mappings $M_{\text{sym}}$. Using the concrete loop bounds $b$, it calculates the tile counts $t$ and tile sizes $p$ from $m' := (\mu, \tilde{R}, \tilde{C})$. Using the calculated $t$ and $p$, it then calculates the latency $L$. If the calculated latency $L$ for the current $m'$ is smaller than the upper bound $L_{\text{upper}}$, this mapping is returned. Due to the order the loop iterates in, we have already found a mapping that achieves the latency bound with the minimum number of PEs.

However, in many cases, the number of mappings $|M|$ might be prohibitively large for exhaustive search at run time, especially inside an embedded real-time system, which is the main domain of TCPAs. For example, fine-grained dynamic frequency scaling might result in a very high number of supported frequencies. In the next sections, we therefore propose two synergizing design-time approaches for reducing $|M|$. The first is mapping-specific and statically computes part of $m \in M$ from the requirements (Section 7.2). The second is general and chooses an adequately reduced subset of $M$ (Section 7.3) by maximizing the coverage (Section 7.3.1) of the subset.
Algorithm 5 Determining a minimal number of PEs to satisfy a given latency requirement $r_L$

**Inputs:** set of symbolic mappings $M^{sym}$, latency bound $L_{upper}$, concrete loop bounds $b$

**Output:** mapping $m'$ such that $m' := (\mu, R, C) \mid b \in [0, L_{upper}]

for $(R, C) = (1, 1), (1, 2), (2, 1), \ldots, (R, C)$ do
  for $\mu = (\Phi, \lambda^*, \tau) \in M^{sym}$ do
    Set tile counts $t$ from $\Phi$ and $R, C$ (see Section 3.4.3)
    Compute tiles sizes $p$ from $b$ and $t$ (see Section 3.4.2)
    Compute latency $L(\lambda^*, \tau, t, p)$ (see Section 4.4)
    if $L(\lambda^*, \tau, t, p) \leq L_{upper}$ then
      return $m' := (\mu, R, C)$
    end if
  end for
end for

no mapping found — error

7.2 Architecture-specific mapping reduction

Given a mapping $m = (m_1, m_2, \ldots) \in M$, the maximum\(^2\) cardinality of $M$ is the product of the cardinalities of all mapping parameter sets $|M_1| \cdot |M_2| \cdot \ldots$

**Example.** Assume we want to map a matrix multiplication loop nest onto a TCPA of size $8 \times 8$ that supports the frequencies 100–1000 MHz in 100 MHz steps. The algorithm is three-dimensional and suppose there are 6 feasible schedules. From Section 3.4, we know the number of possible mappings:

$$|M| = \binom{3}{2} \cdot (8 \cdot 8) \cdot 6 \cdot 10 = 11,520.$$ 

By restricting the mapping in architecture- and application-specific ways, for example, only allowing full rows or columns to be allocated and pre-selecting the space mapping, we can reduce this number considerably to

$$|M| = (8 + 8) \cdot 6 \cdot 10 = 960.$$ 

Note, however, that this number might still be prohibitively large to exhaustively evaluate at run time. $\triangle$

\(^2\)Some combinations of the parameters might be invalid.
As a remedy, a sub-tuple \( m_{\text{compute}} \) of the mapping parameters can often be computed from the remaining parameters \( m_{\text{fixed}} \) to satisfy a set of requirements \( R \). For example, for a latency requirement, we can compute the minimum frequency required to satisfy \( R \) given the allocation and schedule.

**Definition 7.7 ([WT18]).** Given a loop program \( \mathcal{L} \) with requirements \( R \) and secondary objectives \( P \), let \( m \in M \subset M_1 \times M_2 \times \ldots \) be a mapping. Assume \( m \) is split into two sub-tuples \( m_{\text{fixed}} \in M_{\text{fixed}} \) and \( m_{\text{compute}} \in M_{\text{compute}} \). A function \( c_{R,P} : M_{\text{fixed}} \rightarrow M_{\text{compute}} \) is called a mapping completion function if

\[
m_{\text{compute}} = c_{R,P}(m_{\text{fixed}})
\]

such that, given bounds \( b \), together \( m_{\text{compute}} \) and \( m_{\text{fixed}} \) form a mapping that satisfies \( R \) and minimizes \( P \):

\[
m = (m_{\text{compute}}, m_{\text{fixed}}) \mid \left[ b \right] \quad R
\quad \wedge \quad m = \arg \min_{m \in M} (f_1(p_1, m, b), f_2(p_2, m, b), \ldots).
\]

**Example.** Assume we have a single latency requirement \( r_L = [5, 10] \) ms and a single secondary objective to minimize, namely energy \( p_E \). At run time, acceleration of a loop is requested with bounds \( b \) derived from the input size. We want to find a mapping \( m' = (\mu, \widehat{R}, \widehat{C}, f) \). We choose \( M_{\text{compute}} = F \) (the frequency) and \( M_{\text{fixed}} = M_{\text{sym}} \times \Gamma \), where \( \Gamma \) is the set of available PE allocations \( \widehat{R}, \widehat{C} \).

We want to determine \( m_{\text{compute}} = f \) for \( m_{\text{fixed}} = (\mu, \widehat{R}, \widehat{C}) \) such that \( L \in r_L \). Assume the latency \( L(m_{\text{fixed}}, b) \) in cycles is \( 10^6 \). Then

\[
f_{\text{max}} = \frac{L(m_{\text{fixed}}, b)}{5 \text{ ms}} = \frac{10^6}{0.005 \text{ s}} = 200 \text{ MHz}.
\]

Similarly, \( f_{\text{min}} = 100 \) MHz. Given an energy model, the enforcer can now choose the energy-minimizing frequency \( f \in [f_{\text{min}}, f_{\text{max}}] \).

Instead of using the full set of valid mappings \( M \), the enforcer can limit the search space to \( M_{\text{fixed}} \) because \( M_{\text{compute}} \) can be computed from it. Hence, if there are multiple partitionings into \( m_{\text{fixed}} \) and \( m_{\text{compute}} \) (for example, we could instead determine \( \lambda \) from frequency \( f \)), choose the partitioning such that the cardinality \( |M_{\text{fixed}}| \) is minimized.

**Example.** By computing the frequency \( f \) from \( \mu, \widehat{R}, \widehat{C}, \) and \( b \), the number of mappings for the matrix multiplication example can be reduced from 960 to just 96. Determining \( \lambda^* \) from \( f \), however, would only reduce it to 160.

By substituting \( m \) by \( (c_{R,P}(m_{\text{fixed}}), m_{\text{fixed}}) \) in the enforcement function \( e \), we can update its implementation:

**Static inputs:** \( \mathcal{L}, M_{\text{fixed}}, R, P; \) **run-time inputs:** \( B \)
Output: \( m \)

\[ M_{\text{feasible}} \leftarrow \emptyset \]

\textbf{for all} \( m_{\text{fixed}} \in M_{\text{fixed}} \) \textbf{do}

\[ m \leftarrow (c_{R,P}(m_{\text{fixed}}), m_{\text{fixed}}) \]

\textbf{if} \( m \models_R R \) \textbf{then}

\[ M_{\text{feasible}} \leftarrow M_{\text{feasible}} \cup m \]

\textbf{end if}

\textbf{end for}

\textbf{if} \( M_{\text{feasible}} \neq \emptyset \) \textbf{then}

\[ m_{\text{opt}} \leftarrow \text{optimize}(M_{\text{feasible}}, P) \]

\textbf{return} \( m_{\text{opt}} \)

\textbf{else}

\textbf{return} \( \text{None} // \text{issue error} \)

\textbf{end if}

Especially hardware implementations, but also software implementations, of an enforcer might only allow considering \( k \) mappings at run time in order to guarantee correct enforcement\(^3\). Still, even when reducing the cardinality of \( M \) by computing parts of them mapping from \( M_{\text{fixed}} \), the remaining number of mappings might still be greater than \( k \). For this case, we propose to, at compile time, reduce the number of available mappings by a technique called \textit{coverage optimization}. (Note that in the following, we denote the set of available mappings as \( M \) because coverage optimization is independent of the concepts introduced in this section. However, coverage optimization can also be applied to \( M_{\text{fixed}} \) to further reduce the number of mappings to be considered at run time.)

\subsection*{7.3 Coverage-based mapping reduction}

The smaller the maximum number of mappings \( k \) an enforcer can select from, the less likely there is a mapping satisfying the requirements for given loop bounds \( b \in B \). Here, \( B \) denotes the set of loop bounds for a given loop. This likelihood also depends on the composition of the \( k \)-subset: Some compositions are clearly inferior to others because they either do not satisfy the requirements for all \( b \in B \) or because on average, the secondary objectives are “worse.” Considering soft requirements—not satisfying the requirements is merely undesirable, but not a critical failure—some compositions are better because they satisfy the given requirements for a larger number of bounds \( b \).

\textbf{Example}. Suppose \( M \) contains only three mappings \( m_1, m_2, \) and \( m_3 \). The mapping \( m_1 \) allocates \( 1 \times 1 \), mapping \( m_2 \) also allocates \( 1 \times 2 \), and mapping \( m_3 \) allocates \( 4 \times 4 \) PEs. Further assume \( k = 2 \), that is, we have to choose a 2-subset of \( M \) for a given

\(^3\)Recall that in order to guarantee requirements and secondary objectives, the non-functional properties of the enforcer itself need to be taken into account.
enforcer. If, for example, we choose $M_1 = \{m_1, m_2\}$, then this subset is unlikely to satisfy latency requirements for large loop bounds because both $m_1$ and $m_2$ only allocate few PEs. However, if we choose $M_2 = \{m_1, m_3\}$, latency requirements are more likely to be satisfied because $m_3$ utilizes more parallelism. In this sense, $M_2$ is a "better" subset than $M_1$. Still, the question remains: Is there a mapping in $M_2$ that satisfies the given requirements for each possible loop bound? △

We formalize this notion into the following definition to later use for finding optimal $k$-subsets.

**Definition 7.8 ([WT18]).** The coverage of a set of valid mappings $M$ with respect to a given set of requirements $R$ and given loop bound space $\mathcal{B}$ is

$$cov_{\mathcal{R}, \mathcal{B}}(M) := \frac{|\mathcal{B}_{\text{feasible}}(M)|}{|\mathcal{B}|},$$

where $\mathcal{B}_{\text{feasible}} \subseteq \mathcal{B}$ is the subset of loop bounds where $M$ satisfies the requirements:

$$\mathcal{B}_{\text{feasible}}(M) := \{b \mid b \in \mathcal{B} \land \exists m \in M : m \models b R\}.$$

Ideally, even with only a $k$-subset, we still reach the same coverage as the full set of mappings. Note that if the coverage is not 1 in the case of hard requirements even for the full set $M$ of mappings, then the loop bound space $\mathcal{B}$ needs to be restricted (compare [TGR'16]).

### 7.3.1 Determining coverage

For hard requirements, $cov_{\mathcal{R}, \mathcal{B}}(M)$ is required to be 1. A hard requirement allows no violation at any time during any execution [TGR'16]. There are two possibilities to guarantee this: First, analytically. Computing the coverage analytically is both architecture- and requirement-specific, which is why we omit it from the discussion here; however, for TCPAs and symbolically defined non-functional properties such as the latency $L$, we consider analytical coverage determination future work. Second, by exhaustively iterating over all $b \in \mathcal{B}$ and counting for how many $b$ the set of mappings $M$ satisfies the given set of requirements $R$. Obviously, depending on $|\mathcal{B}|$, exhaustive checking may be intractable, but in many cases, the allowed input sizes and thus loop bounds may be restricted a priori (see the example below).

However, for soft requirements, where infrequent violations of a requirement are considered tolerable, we propose a probabilistic approach instead to lower the computational complexity: Definition (7.1) is equivalent to the weighted sum

$$cov_{\mathcal{R}, \mathcal{B}}(M) := \frac{1}{|\mathcal{B}|} \sum_{b \in \mathcal{B}} H(b),$$

where

$$H = \begin{cases} 1 & \text{if } \exists m \in M : m \models b R \\ 0 & \text{else} \end{cases}$$

(7.3)
This weighted sum, and thus the coverage, can be estimated by Monte Carlo integration with importance sampling given a probability density function $Pr : \mathcal{B} \mapsto [0, 1]$ for the loop space $\mathcal{B}$. How might $Pr$ look like?

**Example.** Assume that in a two-dimensional image processing pipeline, we know beforehand that only the following image resolutions are ever processed: VGA, XGA, and Full HD. The loop bound space therefore is

$$\mathcal{B} = \{(640, 480), (1024, 768), (1920, 1080)\}.$$

Out of four images, two will be VGA, one will be XGA, and one will be in Full HD. The probability density function $Pr$ then is:

$$Pr(B) = \begin{cases} 0.5 & \text{if } B = (640, 480) \\ 0.25 & \text{if } B \in \{(1024, 768), (1920, 1080)\} \end{cases}.$$

This scenario is likely, for example, if only few applications use the pipeline and user interactivity is minimal, and can easily be sampled exhaustively for hard requirements.

**Example.** A TCPA provides hardware acceleration for calculating matrix products to many applications, but the size of matrices varies widely. However, assume only sizes between $4 \times 4$ and $256 \times 256$ are supported in a particular implementation. The two matrices must agree in their inner sizes, so we have a span of loop bounds

$$\mathcal{B} = \{ [4...256] \times [4...256] \times [4...256] \}.$$

Run-time tracing of the expected applications indicates that most matrices are around size $32 \times 32$. Matrix multiplication is a three-dimensional loop; therefore, we use a three-dimensional normal distribution with mean $\mu = (32, 32, 32)$ and variance $\Sigma$:

$$Pr := N_3(\mu, \Sigma) \quad \triangle$$

Given $Pr$, $\mathcal{B}$, and a number of iterations $N$ (not to be confused with the number of iteration executed by the loop), Monte Carlo integration is realized by the following algorithm:

```plaintext
covered ← 0
for $i := 1$ to $N$ do
    $b$ ← choose randomly from $\mathcal{B}$ with $Pr$
    if $M \xrightarrow{b} R$ then
        covered ← covered + 1
    end if
end for
$\text{cov}_{R,B}(M) \leftarrow \frac{\text{covered}}{|\mathcal{B}|}$
```

149
7 Run-time Requirement Enforcements of Loops

According to the law of large numbers, this estimate will eventually approach the actual \( \text{cov}_{R,B}(M) \) with \( N \) approaching \( \infty \).

Using the above techniques, we can evaluate the coverage of a \( k \)-subset of \( M \). However, besides coverage, the secondary objectives \( P \) also characterize the quality of a \( k \)-subset of mappings. How can we find the overall “best” \( k \)-subset of mappings and what does “best” mean in the first place?

### 7.3.2 Optimal subset selection

“Best” intuitively means to find a \( k \)-subset that first optimizes the coverage (or ensures a coverage of 1 for hard requirements), then secondary objectives.

**Definition 7.9 ([WT18])**. Given a loop program \( \mathcal{L} \) with valid mappings \( M \) and loop bound space \( B \), a \( k \)-subset \( M_1 \) of \( M \) is optimal with respect to a set of requirements \( R \) and secondary objectives \( P \) if there is no other, distinct \( k \)-subset \( M_2 \) of \( M \) that has a better coverage or has the same coverage but better secondary objectives.

How can we find such a \( k \)-subset of mappings \( M^k \)? We distinguish two cases:

1. \( |B| \leq k \): Selecting the subset is trivial. For each \( b \in B \), determine the mapping that maximizes coverage and include it in \( M_k \). The subset \( M_k \) has the same coverage as the full set of mappings \( M \).

2. \( |B| > k \): We must find an optimal subset among all subsets with a coverage of 1 (hard requirements), or an optimal subset that maximizes coverage over \( B \) (soft requirements). Both versions are combinatorial problems and thus NP-hard.

To solve (2), we decided to extend kofnGA, which is a genetic algorithm proposed by Wolters [Wol15] to find \( k \)-subsets that are optimal in a single objective. For details, refer to [WT18].

### 7.4 Case study and discussion

To further motivate run-time requirement enforcement for loops, we implemented both an architecture-specific run-time requirement enforcer according to Section 7.2 for TCPAs with \( m_{\text{compute}} = (f) \) and \( m_{\text{fixed}} = (\mu, \hat{R}, \hat{C}) \), as well as coverage-based mapping reduction according to Section 7.3 using a genetic algorithm (as presented in [WT18]).

With these implementations, the following experiments serve to show the capabilities and advantages of run-time requirement enforcement for loop nests over a range of loop bounds \( B \):

- Run-time requirement enforcement can improve the quality of non-enforced non-functional properties.
The mapping $m \in M$ that optimizes a set of secondary objectives depends on the actual loop bounds $b \in B$.

For soft requirements, using only a $k$-subset of mappings does not significantly affect coverage depending on $k$.

For ease of comprehension, we chose a simple example that also reflects our most common use cases: enforcing the latency property $p_L$ of a two-dimensional Sobel edge detector on a $4 \times 4$ TCPA while minimizing the energy consumption property $p_E$. There exist $|M^{\text{sym}}| = 2$ symbolic mapping candidates and $|\Gamma| = 16$ available PE allocations $\tilde{R}, \tilde{C}$. The latency property $p_L$ is equal to the latency $L$ given in Equation (4.6). The energy property $p_E$ we model rather naively, but practically, as

$$p_E = (t_r \cdot t_c) \cdot \pi (\det(P) - 1) + L_{\text{local}} \cdot f.$$

where $P$ is the tiling matrix (see Sections 3.4.2 and 3.4.3) and $L_{\text{PE}}$ denotes the number of cycles a PE is active. This model only serves to compare two loop executions; it is not meant to model the actual energy consumption. To summarize:

$$R = \{r_L\}, \ P = \{p_E\}, \ |M_{\text{fixed}}| = |M^{\text{sym}}| \cdot |\Gamma| = 32$$

For demonstration purposes, we set $r_L = [2 \cdot 10^{-2}, 5]$ ms. For easy visualization, we choose the loop bound space to be $B = \{(1 \ldots 256, 1 \ldots 256)^T\}$ and $Pr$ to be the uniform distribution.

While we showcase our approach for only an image processing algorithm here, any algorithm that is mapped using the techniques from the previous chapters can be enforced analogously.

**RRE can improve non-enforced non-functional properties**

We sampled a sequence of $N = 10^5 > |B|$ loop bounds $b$ from $B$ using $Pr$ and for each, ran two scenarios of the Sobel detector: First, regardless of the loop bounds $b$, always use the mapping with PE allocation $\tilde{R} = 4, \tilde{C} = 4$, the first symbolic mapping $\mu_1$ in $M^{\text{sym}}$, and frequency 100 MHz. Second, our architecture-specific run-time enforcer chooses the mapping among all $|M| = 32$ mappings that both satisfies requirement $r_L$ and minimizes the secondary objective $p_E$. Across all runs, we then calculated the arithmetic mean of $p_E$ and coverage separately for both scenarios.

Figure 7.1 contrasts the measurements for both scenarios: We see that run-time requirement enforcement, even in this simple example, not only improves the estimated energy consumption by 8%, but also obtains a higher coverage by almost 5%. This is because a single mapping cannot always satisfy the requirements for all run-time loop bounds, and also might not be energy-minimal.
7 Run-time Requirement Enforcements of Loops

![Graphs showing normalized mean of energy and normalized coverage with and without RRE.](image)

Figure 7.1: (From [WT18]) Using run-time requirement enforcement of latency for a Sobel filter loop nest can lower the estimated average energy consumption (a) and can improve the coverage of the loop bound space (b).

### The optimality of a mapping depends on \( b \)

Exhaustively for each \( b \in \mathcal{B} \), we let our enforcer choose the optimal mapping from all possible \( |M| = 32 \) mappings. Figure 7.2 visualizes the result similar to a heat map; each color represents a different mapping. We can gather several conclusions from this: We can clearly see the large white regions where the requirements cannot be satisfied by any mapping. Also, more importantly, while there seem to be some patterns, which of the 32 mappings is optimal varies widely depending on \( b \). Furthermore, not every mapping is optimal for equally as many loop bounds \( b \). Figure 7.3 shows how often each of the 32 mappings was selected by the enforcer during the \( N = 10^5 \) requested loop executions. Some mappings are even never selected as optimal, which supports our idea of reducing the cardinality of \( M_{\text{fixed}} \) without sacrificing coverage.

### 7.5 Related work

Run-time requirement enforcement, as described in this chapter, denotes a class of techniques to enforce a set of non-functional property requirements during program execution [TGR+16; TPK+20; TMP+20]. Closely related is run-time enforcement, “a technique dedicated to ensure that a run of a system satisfies a given desired property” [Fall10]. Run-time enforcement approaches model an application as a sequence of (abstract) actions called an execution sequence. Here, the two techniques differ in two major ways:

First, in run-time enforcement approaches, properties solely pertain to characteristics of the execution sequence (for example, two actions A and B may not be executed at the same time). Therefore, run-time enforcement techniques do not allow to provably enforce absolute timing, energy consumption or other requirements on non-functional properties of a cyber-physical system implementation. Second, run-time enforcement
7.5 Related work

(a) Optimal mappings for each $b = (b_1, b_2)^T \in B$

(b) Optimal mapping for each $b \in (64\ldots128, 64\ldots128)^T \subset B$

Figure 7.2: (From [WT18]) Depending on the actual loop bounds $b \in B = \{(1\ldots256, 1\ldots256)^T\}$, a different mapping is optimal with respect to secondary objectives. Each color represents one of the 32 mappings in $M_{\text{fixed}}$. (a) visualizes the entire loop bound space $B$, (b) shows the details of a subset.

Figure 7.3: (From [WT18]) Percentage of each of the 32 mapping candidates in $M_{\text{fixed}}$ being an optimal solution according to Figure 7.2.
approaches enforce properties by modifying the execution sequence. For example, Pinisetty et al. recently proposed run-time enforcement techniques for synchronous reactive systems in which synthesized enforcement automata modify erroneous inputs and outputs such that properties are satisfied [PRS+17a; PRS+17b]. In contrast, our approach finds a mapping prior to loop execution that guarantees bounds on non-functional properties. Doing so at run time enables the system to dynamically react to changes in the environment such that other non-functional properties are optimized.

Singh et al. [SSK+13] survey methods of optimizing non-functional properties by mapping applications onto multi-core systems and identify three classes: design-time mapping, run-time mapping, and hybrid mapping. Design-time mapping decides the mapping at design time, run-time mapping at run time, and hybrid mapping uses design-time analyses for run-time mapping. Consequently, run-time requirement enforcement belongs to either run-time or hybrid mapping. In this paper, we present a hybrid approach: using a design-time analysis, we determine a subset of mappings from which the enforcer can select at run time.

What separates our work from previous work on hybrid mapping to optimize a set of non-functional objectives? First, we focus on run-time enforcement of for-loops, which to the best of our knowledge, we are the first to do. Loops are a major source of uncertainty for design-time analyses but most work uses task graphs (e.g., [QP15]), eschewing favorable properties: for-loops are very regular, allowing a more accurate characterization of non-functional properties.

Second, we focus on a specific class of loop accelerators, namely processor arrays, whereas previous work mostly targets multi-core or heterogeneous multi-processor systems (see [SDM+17] for a survey) without considering details of mapping to specific resources such as accelerators. However, processor arrays have much simpler architectures and thus are very amenable to predicting non-functional properties, facilitating run-time enforcement.

Third, to be applicable to real product implementations, we want to enforce absolute values on, for example, timing, power, and energy.
This thesis presented symbolic loop compilation, the first full-fledged solution to compiling loops for a TCPA in a way that allows for dynamic mapping at run time, that is, without knowing the loop bounds and number of PEs at compile time. This is achieved by a separation into two phases: symbolic mapping, which generates a set of symbolic configurations, and instantiation, which generates concrete programs and configuration data for the target TCPA from one of the symbolic configurations. While symbolic mapping frontloads all NP-hard problems to compile time to ensure high-quality mapping results (that is, a low initiation interval, reusing of registers and interconnect resources, and so on), instantiation scales well with increasing TCPA size.

Symbolic mapping in particular further consists of symbolic modulo scheduling, the first solution to modulo-scheduling multi-dimensional loops using symbolic tiling. In this thesis, we further showed how to incorporate instruction selection, register allocation, and propagation channel routing into symbolic modulo scheduling. The results of symbolic modulo scheduling are compiled into a set of symbolic configurations. Part of a symbolic configuration is a polyhedral syntax tree, which is a novel, space-efficient program representation parameterized in the loop bounds and number of PEs.

Instantiation generates concrete programs and configuration data from a symbolic configuration. The symbolic configuration to instantiate is either selected to be the latency-minimal one, or, if run-time requirement enforcement is used, by a loop enforcer. The most complex part of instantiation, program instantiation, generates the PE programs from the given polyhedral syntax tree in a manner whose time complexity is not proportional to the number of PEs, therefore scaling well with increasing TCPA sizes.

As an application of symbolic loop compilation and the dynamic mapping capabilities it enables, this thesis further formalized run-time requirement enforcement for loops with TCPAs as a specific target architecture example. We also showed methods to reduce the number of mappings that need to be considered during enforcement; this aids in achieving predictability in the enforcer itself.
8 Conclusion

8.1 Outlook and further work

Although symbolic loop compilation, as presented in this thesis, already provides a powerful framework for TCPAs, we envision further improvements in the future.

8.1.1 Run-time conditions

At the time of writing, symbolic loop compilation does not yet support algorithm classes with data-dependent run-time conditions. Since many algorithms exhibit data-dependent behavior, it is important to extend symbolic modulo scheduling to these algorithm classes. For example, Hannig introduced dynamic piecewise linear algorithms [HT04], which, as the name implies, incorporate run-time conditions. For example:

\[ S_1: x[i] = \begin{cases} a[i] & \text{if } a[i] \geq b[i] \\ b[i] & \text{else} \end{cases} \]

In the corresponding RDG, run-time conditions are reflected by two new types of nodes: a \textit{condition node} that computes the condition \((a[i] \geq b[i])\) in the example, and a \textit{merge node} that, depending on the computed condition, selects one of the two paths (either \(a[i]\) or \(b[i]\) in the example).

Within symbolic loop compilation, we see two ways to support dynamic piecewise linear algorithms:

- Schedule both branches to execute in parallel and schedule a conditional move instruction associated with the merge node. While this is the simpler solution, it would require hardware support in the form of a functional unit that implements a conditional move instruction, for example \texttt{move rd0 rd1 rd2 rc0}. In this example, either \(rd1\) or \(rd2\) is copied to \(rd0\), depending on whether the control register \(rc0\) is 0 or 1. The value of \(rc0\) would be set by the operation realizing the condition node.

- Schedule both branches to be mutually exclusive and execute only the correct branch using conditional branching. This way is more natural for sequential programs such as those executed by PEs. However, because the branch unit of a functional unit only supports a limited number of branch targets, these resource constraints need to be incorporated into symbolic modulo scheduling and control flow analysis. For example, if there are not enough simultaneous branch targets, then the initiation interval might need to be increased.

The two approaches are not mutually exclusive, and there are situations in which either one would be the better choice. Either way, polyhedral syntax trees and program instantiation need to be extended to provide support for run-time conditions.
8.1 Outlook and further work

8.1.2 TCPA-specific optimizations

RDGs are an excellent representation for structural transformations that do not change the semantics of the represented UDA, but influence the properties of the mapping. There are many potential optimizations, such as transformations that reduce pressure on resources or that help to decrease the initiation interval. For example, it might make sense to duplicate nodes and calculate a value twice, or to remove propagation nodes (since propagation is often a redundant move instruction, as opposed to in hardware, where it is just a wire).

This extends to virtual registers: What exactly is an “optimal” allocation of virtual registers? If the initiation interval needs to be increased because of an insufficient number of virtual registers, or because the virtual registers do not support sufficient concurrent writes, might it make sense to transform the RDG instead?

It is also an unsolved problem to minimize the program size during symbolic mapping. This is non-trivial because during compilation, the concrete programs are not known yet. This is a problem because if the instruction memory of the target TCPA is not large enough, instantiation might fail.

8.1.3 Further work on run-time requirement enforcement on TCPAs

This thesis merely scratched the surface of the potential power of run-time requirement enforcement. Besides further research into run-time requirement enforcement itself, we envision further research into its applications and larger system design.

Take the LU decomposition as an example, which decomposes a square matrix $A$ into the product of two matrices $L$ and $R$:

$$A = L \cdot U, \quad A, L, U \in \mathbb{R}^{N \times N},$$

where $L$ is a lower triangular matrix and $R$ is an upper triangular matrix. To parallelize the LU decomposition, it can be divided recursively into blocks, for example $3 \times 3$:

$$\begin{pmatrix} A_1 & A_2 & A_3 \\ A_4 & A_5 & A_6 \\ A_7 & A_8 & A_9 \end{pmatrix} = \begin{pmatrix} L_1 & L_2 & L_3 \\ L_4 & L_5 & L_6 \\ L_7 & L_8 & L_9 \end{pmatrix} \begin{pmatrix} U_1 & U_2 & U_3 \\ U_4 & U_5 & U_6 \\ U_7 & U_8 & U_9 \end{pmatrix}$$

These subproblems are interdependent; Figure 8.1 shows the dependence graph for this division into $3 \times 3$ blocks. Clearly, the execution time is dominated by the critical path (nodes marked in bold borders). This allows run-time enforcement to consider further options than simply the mapping on a singular TCPA: For example, given a bound on the execution time, the nodes along the critical path could be accelerated on a TCPA, while the non-critical nodes are executed in parallel on multi-core processors.
Figure 8.1: Dependence graph of the block-recursive LU decomposition with $3 \times 3$ blocks. The critical path is marked by thick borders. Blue nodes calculate the LU decomposition of the submatrix $A_i$, green blocks solve for $L_i$ or $U_i$, and yellow blocks calculate the shown difference of matrices.
This appendix introduces the common notational conventions used throughout the thesis. Notation only used within certain chapters and sections will, however, be defined there.

**Linear algebra**

\( \mathbb{Z} \) is the set of integers, \( \mathbb{N} \) the set of natural numbers including 0. Scalar variables and scalar functions are denoted using lower-case Latin or Greek letters, such as \( a \) and \( \alpha \). Vectors and functions resulting in vectors are denoted using bold lower-case Latin or Greek letters, such as \( \mathbf{a} \) and \( \alpha \), and their elements as scalar variables with a subscript, for example

\[
\mathbf{a} = (a_0, a_1, \ldots)^T.
\]

Whether a vector is a row or a column vector should be clear from the context. Similarly, matrices are denoted using bold upper-case Latin or Greek letters, such as \( \mathbf{A} \) and \( \Phi \), their rows or columns as vectors with a single subscript, and their elements as scalar variables with two subscripts. For example:

\[
\mathbf{A} = (\mathbf{a}_1, \mathbf{a}_2, \ldots) = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots \\ a_{2,1} & a_{2,2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \end{pmatrix} = (a_1, a_2, \ldots)^T.
\]

When represented as vectors, a matrix is either a column vector of row vectors or a row vector of column vectors.

We also write \((\mathbf{a}, \mathbf{b})\) or \((\mathbf{a}, \mathbf{b})^T\) to denote the concatenation of two vectors. It should be clear from the context whether the result is a matrix or a vector, and whether it is a row or column vector. In particular, we write \((\mathbf{a}, \mathbf{b})^T\) instead of \((\mathbf{a}^T, \mathbf{b}^T)^T\) to concatenate two column vectors into another column vector to avoid typographic clutter.

**Algorithms and polyhedra**

We denote recurrence algorithms (see Section 3.1) using upper-case Fraktur letters, such as \( \mathfrak{A} \) (Fraktur A) or \( \mathfrak{U} \) (Fraktur U). An equation of a recurrence algorithm is denoted by
**A Notation**

$S$ with a subscript, for example $S_1$. Indexed variables (see Section 3.1) are denoted using lower-case names, such as $x$, $y$, or $\text{bits}$.

In the polyhedral model, iterations of a loop are modeled as $n$-dimensional integer points (or vectors). By convention, we use $i$, $j$, and $k$ to denote such iteration vectors. The set of iterations a loop iterates through is called its iteration space. We denote iteration spaces and other sets of integer points using calligraphic letters, such as $I$ or $\mathcal{L}$. A superscript asterisk, as in $I^*$, marks that a set of integer points refers to either a tiled iteration space or to a set embedded into a tiled iteration space (see Section 3.4.2).

**Directed graphs**

This thesis makes heavy use of directed graphs. In general, we refer to the set of nodes as $V$ and the set of edges as $E$. Subsets use the same letters, but with super or subscripts, such as $V^{\text{top}}$ or $E_1$. Nodes are denoted using $v$ or $w$, possibly with a subscript, such as $v_1$ and $w_2$. Edges are denoted analogously, but using the lower-case letter $e$. Each edge is a tuple $e = (v, w)$ that consists of the head or sink node $w$ and the tail or source node $v$. We differentiate between incoming edges of a node $v$—where $v$ is the head node—and the outgoing edges of a node $v$—where $v$ is the tail node. The set of incoming edges is denoted $E^+(v)$, the set of outgoing edges $E^-(v)$. Furthermore, the cardinalities of these sets define the indegree $\deg^+(v) = |E^+(v)|$ and outdegree $\deg^-(v) = |E^-(v)|$ of a node $v$. 
Synthesis and Run-time Configuration Flows

B.1 Synthesis flow of a TCPA

As Chapter 2 established, TCPAs are highly configurable with many synthesis-time parameters at all levels of the component hierarchy. In addition, there are currently two synthesis targets: hardware from a RTL (register-transfer level) description and a cycle-accurate simulator from C++ sources\(^1\). The flow to synthesize a concrete TCPA, shown in Figure B.1, is therefore designed with both flexibility and usability in mind.

The starting point is a hierarchical architecture description of all synthesis-time parameters of a TCPA in XML. XML is not only human-readable and widely supported—parsers exist for all major programming languages—, it also reflects the hierarchical structure of a TCPA very well. The following excerpt describes the top level of a 4 × 4 TCPA\(^2\):

```
<TcpaArchitecture>
  <Size>
    <Rows>4</Rows>
    <Columns>4</Columns>
  </Size>
  <!-- further common architecture description -->
  <ProcessorArray>
    <!-- processor array architecture description -->
  </ProcessorArray>
  <Periphery>
    <!-- periphery architecture description -->
  </Periphery>
</TcpaArchitecture>
```

The architecture description is fed to tcpasynth, a tool that generates sources for the two targets in accordance with the description—VHDL for the hardware target and C++ for the simulator target. To improve code quality and maintainability, the

---

\(^1\)This simulator was developed in tandem with symbolic loop compilation, but for brevity, the interested reader is referred to [WHT17] for details.

\(^2\)For the full specification of an XML architecture description, refer to the accompanying tool documentation.
sources of both targets are developed and optimized manually, but use built-in language features for compile-time parameterization to represent the synthesis-time parameters: generics in VHDL and templates in C++. This way, tcpasynth needs to only generate assignments for these generics (as a VHDL library) or template parameters (as a C++ header file), which not only eliminates many sources of bugs inherent to generating sources completely from scratch, but also decouples hardware development from the code generation.

Finally, using a synthesis tool (such as Xilinx Vivado) for the RTL description or a C++ compiler for the simulator sources, the respective synthesis output—hardware or simulator—is generated.

**Remark.** The reference compiler implementing symbolic loop compilation that was developed for this thesis also takes the architecture description as an input because it describes necessary compile-time knowledge such as the number of available registers, interconnect topology, and so on.
B.2 Run-time generation and instantiation flow of configuration data

Figure B.2 shows the tool flow and its multiple paths from the generation of a symbolic configuration by the compiler until the configuration of a TCPA with a binary run-time configuration. The binary representation can either be translated from a concrete configuration in XML format using the tcpaconf tool at compile time or by instantiating the binary representation of a symbolic configuration within the TCPA driver (see Section 2.3.3) at run time. The binary representation of a symbolic configuration can be generated by tcpaconf from a symbolic configuration in XML format at compile time. Given concrete loop bounds and number of PEs, tcpaconf can also instantiate a symbolic configuration into a concrete XML configuration at compile time. Finally, a symbolic configuration in XML format is generated by tcpac, the compiler described in this thesis, after successfully symbolically compiling a loop program.

Similar to the architecture description of a TCPA, the following XML excerpt shows parts of a concrete run-time configuration:

```xml
<ProcessingElement>
  <Region name="pc_0">
    <Element row="0" column="0" />
  </Region>
  <Program file="pc_0.asm" />
  <Registers>
    <Data>
      <FeedbackRegister index='0' length='16' />
    </Data>
    <Control>
      <InputRegister index='0' length='16' />
    </Control>
    <Registers>
  </ProcessingElement>
```

In this example, the top-left PE is configured with the program specified in pc_0.asm. Both fd0 and ic0 are configured with depth 16.
Figure B.2: Tool flow when configuring a TCPA. A TCPA is configured using a binary representation of a concrete configuration that contains configuration data for all involved components. In the context of this dissertation, the binary concrete configuration is instantiated from a binary symbolic configuration, which in turn is the artifact generated by symbolic loop compilation (that is, by tcpac, the compiler described in this dissertation).
Tightly Coupled Processor Arrays: Related Work

Three classes of accelerators share design aspects and goals with TCPAs: MPPAs (massively parallel processor arrays), CGRAs, and systolic (CNN) accelerators. All three use a grid of many interconnected computing elements to accelerate loops in a highly parallelized fashion; their distinction, however, is not precisely defined in the literature. For the sake of structuring the following discourse, we classify them based on architecture, intended application, and mapping methods:

- If an accelerator is a systolic array that is specifically designed to accelerate CNNs, disregarding other loop applications, we consider it a systolic CNN accelerator. This narrower scope is reflected in the mapping flow: Starting from a CNN description (provided by frameworks such as TensorFlow or Caffe), each layer of the network is mapped to the accelerator in a layer-specific way.

- If an accelerator’s processing elements are programmable or even general-purpose processors and it has a wider scope than only accelerating loops, we consider it an MPPA. MPPAs often use open or proprietary thread-based and/or message-passing parallelizing frameworks to map not only loops, but large parts of an application onto them.

- If an accelerator’s processing elements are reconfigurable fixed-function functional units and it cannot operate without a host processor, we consider it a CGRA. CGRAs usually only accelerate loop kernels, which are mapped by embedding the data flow graph of a loop’s body onto the CGRA.

TCPAs share aspects with all three, but do not fully belong to any of the classes. The next sections present commercial and academic examples from all three classes and compare them to TCPAs, in particular regarding the three goals of high performance, energy efficiency, and predictability.
C Tightly Coupled Processor Arrays: Related Work

C.1 Massively parallel processor arrays

Compared to most MPPAs, the PEs of a TCPA are much more limited: there are neither caches and nor local random-access memories, which entails, for example, a lack of a stack, interrupts, and multi-threading. MPPAs also usually have NoCs instead of circuit-switched or fixed interconnects to facilitate far-distance communication. Although these design choices make MPPAs more powerful in general, allowing a wider range of applications, they also hinder the achievement of predictability and reduction of energy consumption. In fact, some MPPAs are developed with high-performance computing in mind—such as the SW26010 as a part of the TaihuLight supercomputer [FLY+16]—, but for the sake of comparison, we focus on MPPAs that target low-power embedded systems.

C.1.1 Commercial

Following the stagnation of processor frequencies in the mid-2000s, commercial interest for MPPAs started to grow. Most of these early ventures have been discontinued, for example the picoChip picoArray [DTP+05] (acquired by Intel), the Ambric Am2045 [BJW07] (company closed down), and the Tilera TILEPro64 [BEA+08] (acquired by Mellanox Technologies). However, many others are still commercially available as of 2020. Here, we focus on two MPPAs for embedded systems: the Coherent Logix HyperX hx3100 and the Kalray MPPA-256.

The HyperX hx3100 started production in 2011 and is advertised by its developer Coherent Logix as a "low-power, high-performance" accelerator for embedded systems [IDS12]. It contains $10 \times 10$ 32-bit processors for digital signal processing embedded into a NoC comprised of $11 \times 11$ routers (see Figure C.1). Each core runs with up to 500 MHz, supports both fixed- and floating-point operations, and has 4 kB of instruction memory. Data is stored in the routers, each of which provides 8 kB of data memory. At the borders, the routers are connected to high-speed I/O interfaces with a bandwidth of up to 104 Gb/s, emphasizing the need for high data bandwidths also seen in TCPAs.

According to Coherent Logix, it was important to have an integrated design flow that “preserves the [existing] software stack as-is” [IS11]. As such, Coherent Logix provides mapping tools for the widely used languages C, Simulink, and MATLAB. Regardless of implementation language, an application is partitioned into tasks and mapped to disjoint subsets of the processor array that are pipelined to run in parallel. Figure C.2 shows an example: a digital video stabilization application written in C is partitioned into four tasks and mapped differently sized regions of the hx3100. This realization achieves 30 frames per second on an 800 $\times$ 600 video stream while consuming about 2.5 W [HTB+11]. While this falls in the advertised lower-power range of 25 mW to 2.5 W, there have been no investigations into predictability of the platform.

Despite emphasizing the existing software stack, Coherent Logix also acknowledged that the “programmability of a many-core processor is generally perceived by indus-
try as difficult and inefficient” and proposed a mapping method based on data flow graphs [TH13]. Each node is mapped to a region of processing elements, communication channels are routed between them, and C code parallelized with MPI is generated.

The second example, the MPPA-256 developed by Kalray, started production in 2013 and contains 256 cores divided into 16 clusters with 16 cores each [dDAB13] (see Figure C.3). Cores within a cluster communicate using shared memory, while clusters communicate over a torus-connected NoC. Each core is a 32-bit VLIW processor with five execution units: two ALUs, a multiply-and-accumulate unit that supports floating-point operations, a load-store unit, and a branch-and-control unit [IDG17]. These units communicate using a shared register file of 64 32-bit general-purpose registers. While this architecture resembles that of PEs in a TCPA, an MPPA-256 core additionally has dedicated data and instruction caches as well as direct access to the shared cluster memory.

The NoC also connects the 16 clusters to four I/O subsystems, one at each border. Each I/O subsystem offers four additional cores and controllers for high-speed connections to DDR memory, Ethernet, and the PCI Express bus, emphasizing the need for a high I/O bandwidth.

Kalray offers two high-level approaches for mapping applications to the MPPA-256: POSIX-based C++ for CPU-like and OpenCL for GPU-like programming [CNJ15]. If using POSIX-based C++, the application must be manually split into processes, one for each cluster, which in turn may spawn additional threads. The latter, that is, intra-tile parallelization, may be automated to a degree by annotating the code with OpenMP directives. However, for inter-cluster communication, a proprietary but POSIX-compliant library is used.
Regarding predictability, Kalray states that to mitigate the uncertainty inherent to shared-memory multi-core systems with caches (as each cluster is), the cores are designed to be timing compositional with LRU caches and opcodes that bypass the caches. On the cluster level, each core has a private bus to the shared cluster memory. On the chip level, the NoC offers service guarantees, such as minimum bandwidth and maximum latency [dDim15; dDG17]. Finally, on the I/O level, the DDR controller offers service guarantees and special predictability configurations as well [dDim16].

As for energy efficiency, Kalray published a case study where de Dinechin et al. implemented a library for modular arithmetic on the MPPA-256 and measured its performance and power consumption in comparison to two other implementations: for a system with two Intel E5-2650 à 8 cores, and for the Nvidia GeForce GTX TITAN X GPU [IDG*17]. Reportedly, the MPPA-256 performed much better than the twin-CPU system in both operations per second and joule per operation. Compared against the GPU, however, the MPPA-256 performed only a third to a half as many operations per second, but achieved a much lower joule per operation measurement.

### C.1.2 Academic

Probably due to the prohibitive cost of developing and producing large-scale chips, far fewer academic MPPAs exist. The two largest examples are the 167-core computational platform [TCM*09] in 2009 and the KiloCore [BSP*17a] in 2017, both from the University of California, Davis.

The larger and more recent example, the KiloCore, features 1000 cores arranged in a $32 \times 32$ grid, the last row replacing 24 processors with 12 memory modules (see
C.1 Massively parallel processor arrays

Figure C.3: (a) The MPPA-256 from Kalray has $16 \times 16 = 256$ cores arranged into 16 shared-memory clusters. The clusters are connected with a NoC and connected to a high-bandwidth I/O subsystem at the borders. (b) Each cluster contains 16 cores with 2 MB of shared memory. (Source: [IDG+17])
Figure C.4: The KiloCore from the University of California, Davis (figure taken from [BSP*17a]) provides a grid of 1000 cores, supplemented by 12 memory modules and interconnected via a circuit-switched and a routed network.

Each core is a simple 16-bit processor with $128 \times 40$ bit (640 bytes) of instruction memory and 512 bytes of data memory, which is addressed using three address generators. A core runs on average at 1.78 GHz, although the clocking scheme allows each core to run as fast as it can. The cores are connected to two communication networks: via a circuit-switched network with 28.5 Gb/s, made for high-speed and low-latency communication, and a NoC with 9.1 Gb/s per router port for high fan-in/fan-out communication and administrative messaging. In- and output ports connecting to either the circuit-switched network or the NoC are situated at the corners of the processor array.

Programming the KiloCore comprises multiple steps and starts with a C, C++, or assembly implementation of the application [BSP*17b]. The implementation undergoes two manual partitionings: serial and parallel. Serial partitioning puts consecutive tasks into a pipeline, while parallel partitioning parallelizes tasks by replicating them. Finally, a custom tool maps the tasks to the cores under several constraints, such as avoiding faulty processors and implementing various optimizations.

The team behind the KiloCore simulated its power consumption for four different applications and compared the results to multiple Intel CPUs and nVidia GPUs. When operating at 1.1 V, they measured a geometric mean improvement of 9.4 for throughput per Watt, with absolute values between 3.12 W (sorting 100-byte records) and 15.4 W (AES).
C.1.3 MPPAs: Conclusion

These examples show that MPPAs can outperform both CPUs and GPUs in terms of energy efficiency and predictability. But they also show that to achieve that, careful attention to the design of data networks and I/O interfaces is required to achieve the necessary high I/O bandwidth to not starve the accelerator. Similar care went into the design of TCPAs’ interconnect and I/O interface. Furthermore, they also show that programming a large number of cores is difficult, especially regarding the large scope of MPPAs: all three examples have custom mapping flows with mostly manual parallelization. In contrast, TCPAs have a more limited scope—loop programs—and their mapping flow, described in this thesis, includes fully automatic parallelization.

C.2 Coarse-grained reconfigurable arrays

For a detailed overview of CGRAs that were developed between 1990 and 2016, we refer to the survey by Wijtvliet et al. [WWC16]. Most are academic projects, although some commercialized examples exist, such as the PACT XPP [BEM’03; Tec]. According to the authors of the survey, a coarse-grained reconfigurable array is defined by: (1) temporal reconfigurability on the loop level or above and (2) spatial reconfigurability on the functional unit level or above. In other terms, the accelerator is reconfigured each time a different loop is accelerated (temporal reconfigurability), but the granularity of reconfiguration is at the functional unit level and not, for example, at the gate level (spatial reconfigurability). This definition implies most CGRAs to be a grid of interconnected fixed-function functional units. Accordingly, TCPAs are not CGRAs in the strict sense by this definition, but share the intention of accelerating loops and the temporal reconfigurability at the loop level. While this definition is rather narrow, the following two examples showcase different approaches to CGRA design.

The first example is ADRES (architecture for dynamically reconfigurable embedded system [MVV’03]), a popular CGRA template whose architecture design space was explored [BBK’07] and enhanced accordingly [BBD’08]. Similar to TCPAs, ADRES is a template for a class of accelerators that can be instantiated from an XML description [MVV’02]. ADRES leverages the need for a host CPU by sharing resources between a VLIW core and a reconfigurable matrix in a processor/co-processor relation. The reconfigurable matrix comprises a mesh of interconnected, reconfigurable functional units interspersed with local register files; between the VLIW core and the reconfigurable matrix, there is an additional pair of register files connected to a subset of the functional units. Because of the resource sharing, ADRES may run in either VLIW or CGRA mode: In VLIW mode, the functional units that are connected to the shared register files act as the functional units of the VLIW core to execute general-purpose application code. In CGRA mode, the reconfigurable matrix accelerates loop kernels with all available functional units.

A common problem in CGRAs is that a simple circuit-switched mesh makes routing
Figure C.5: The ADRES CGRA shares resources between a VLIW core and a reconfigurable matrix of reconfigurable functional units. A subset of the functional units is connected to a pair of shared register files. These functional units are either used as the functional units of the VLIW core or as parts of the reconfigurable matrix. (Source: [MVV+03])
C.2 Coarse-grained reconfigurable arrays

Figure C.6: The HyCUBE CGRA connected each processing element to a router that contains a crossbar between all four cardinal directions. Using this NoC, longer routes between processing elements can be utilized, improving mapping. (Source: [KMM+17])

more difficult if only local neighbor-to-neighbor communication is allowed. That is why HyCUBE [KMM+17], a recently proposed CGRA, offers a NoC instead. In HyCUBE, each processing element is connected to a router that contains a crossbar switching between all four cardinal directions. By using clock-less repeaters, this allows multiple hops within a single cycle. This is similar to TCPAs that allow many different topologies for its interconnect network, also allowing multiple hops in a single cycle.

Following from their simpler architecture, CGRAs accelerate only loop kernels—that is, the body of a loop—and leave bookkeeping, data fetching, and so on to a host CPU. The kernels are mapped to CGRAs using a variation of the resource-constrained modulo scheduling problem where a kernel is represented as a data flow graph and the CGRA as a MRRG (modulo routing resource graph). In a modulo routing resource graph, each node represents a port of either a functional unit or a register, and each edge represents a connection between these ports. This representation was pioneered by Mei et al. to exploit iteration-level parallelism when mapping loop kernels to ADRES (see below) [BVV+03]; in particular, they solved mapping the data flow graph to the modulo routing resource graph using simulated annealing. Other authors use integer linear programming [CA18; WA19] or graph-theoretic methods such as graph embedding [PFK+06] and graph-minor mapping [CM12]. Finally, some work claims that using edge-centric approaches—prioritizing routing over placement—are better suited because routing is the more difficult on CGRAs [PFM+08]. In contrast, the methods described in this thesis instead parallelize whole loops by tiling (see Section 3.4.2).

In conclusion, CGRAs have simpler processing elements than TCPAs, which narrows their parallelization scope to the iteration-level. This increases the reliance on a host CPU while also neglecting other levels of parallelism, especially loop-level parallelism of executing multiple parts of a loop in parallel. This is reflected in their mapping flow, which, in contrast to TCPAs, concentrates on mapping the body of a loop.
C.3 Systolic arrays for accelerating CNNs

CNNs are computationally expensive and spend most of their run time performing convolutions and other linear algebra algorithms, making them excellent targets for dedicated acceleration. This suitability combined with their popularity lead to the development of both commercial and academic CNN accelerators, of which many are systolic arrays. However, even more so than many other loop applications, CNNs are often memory-bound, a characteristic that heavily influences the architecture of these accelerators. The three following examples, two commercial and one academic, showcase three architectures that are distinct in their memory design, but have a systolic array at the core.

C.3.1 Commercial

Google’s TPU (tensor processing unit) has been employed in data centers since 2015, making the report from 2017 [JYP+17] a reliable summary of its merits. As illustrated in Figure C.7, the TPU’s main computational component is a systolic array with 256 8-bit MAC units connected to three different on-chip buffers: First, an 8 MB weight buffer that loads weights directly from an off-chip DDR3 memory, Second, 4 MB of accumulators that store the partial sums from the systolic array and propagate them to dedicated components for applying activation functions and pooling. Third, a 24 MB feature map buffer that stores both input feature maps coming from the host, as well as intermediate feature maps coming from the systolic array. These buffers are sized such that no spilling or reloading happens during normal operation. Likewise, the connections between the components are 256 bytes wide to provide the necessary high bandwidth within the chip.

In conclusion, the TPU solves the problem by providing large, fast, and dedicated-function on-chip buffers. While this might raise concerns regarding energy efficiency, Jouppi et al. compared the power/Watt of the TPU to both an Intel Haswell CPU and an nVidia K80 GPU [JYP+17]. The TPU had a better power/Watt measurement than the CPU by a factor of 17 to 34, and better than the GPU by 14 to 16.

The second example is the accelerator published by the Watson Research Center of IBM [FSZ+18]. Instead of providing large and data-specific flat on-chip memories, this accelerator instead provides a hierarchy of unified scratchpads. Still, as its heart, there is a systolic array with 512 16-bit floating-point functional units arranged as a torus, as well as an additional row of 32-bit MAC units. Each unit has a 16-deep FIFO. The systolic array is connected to two 8 kB scratchpads, one for each orientation, which are both connected to an additional 2 MB scratchpad connected to the host. They offer a combined bandwidth of 384 GB/s. The functional units support a small set of common instructions, most importantly, multiply-and-accumulate and some non-linear activation functions such as ReLU.

They use DeepMatrix [VCS+17] to explore mappings of an input DNN and find the performance-optimal partitioning onto the accelerator.
C.3 Systolic arrays for accelerating CNNs

Figure C.7: Google’s Tensor Processing Unit features a systolic array of 256 MAC units whose inputs and outputs are streamed over high-bandwidth connections from and to on-chip buffers. (Source: [JYP+17])

Figure C.8: CNN accelerator from IBM. (Source: [FSZ+18])
C.3.2 Academic

MIT’s Eyeriss [CKE+16] uses an array of $12 \times 14$ processing elements using a multi-level memory hierarchy to optimize data access times (and reduce energy consumption). The processing elements are connected using a NoC to enable point-to-point and single-cycle data delivery. There are three separate NoCs, one for each of the data types (input, output, partial sum). In addition, Eyeriss uses compression techniques specific to CNNs to further improve energy efficiency. Each processing element is in principle a MAC-unit with each $224 \times 16$ bits of scratchpad memory for input feature maps and filter weights, as well as $24 \times 16$ bits of scratchpad memory for partial sums.

The next level of the memory hierarchy is a global buffer whose $108$ kB of memory is shared across the filter weights, input feature maps, and partial sums. The global buffer is connected to an off-chip DRAM with a width of 64 bits; both input and output feature maps additionally use run-length encoding in hardware.

In 2019, the architecture was updated to better support sparse and compact deep neural networks [CYE+19]. Instead of the flat multi-cast NoC used in the original architecture, here, a hierarchical mesh is used where the global buffer is split up and, together with the processing elements, put into clusters.

C.3.3 Conclusion

The three examples show three aspects specific to CNN accelerators: (1) special attention must be paid to memory design because most CNNs are memory-bound, (2) CNNs allow computation at lower precision, for example 8 or 16 bits, and (3) dedicated realizations of non-linear activation functions make sense. While TCPAs are more general than these accelerators, when accelerating CNNs with a TCPA, the synthesis-time parameters may be chosen accordingly: choose fitting I/O buffer sizes, choose an appropriate data width, and include functional units for non-linear activation functions.
C.4 Conclusion

What distinguishes TCPAs most from these other classes of accelerators is their scope: TCPAs target the acceleration of an entire loop nest at all its levels of parallelism. In contrast, MPPAs have a wider scope—they accelerate whole applications—, making automatic parallelization significantly more difficult. TCPAs offer a more comprehensive parallelization approach, as described in this thesis, if only for parts of an application that fits the loop model. In addition, TCPAs also have more hardware features specific to loop acceleration, such as the global controllers. CGRAs have a narrower scope—they accelerate only the body of a loop—which disregards at least the loop-level parallelism. This absolves mapping for CGRAs of dealing with symbolic parallelization since the loop bounds are inconsequential for the CGRA. However, TCPAs further unburden the host CPU by taking more responsibility. Finally, systolic CNN accelerators are more specific in which loops they accelerate and hence feature many neural network-specific components and mapping infrastructure.
Construction of a Reduced Dependence Graph

For a given UDA (uniform dependence algorithm), the corresponding RDG (reduced dependence graph) is derived as follows. First, the set of nodes $V$ is constructed:

1. Insert an input node $v$ for each $x_{in} \in X_{in} \setminus X_{constant}$, annotated with the input variable $x[v] \leftarrow x_{in}$ and $I[v] \leftarrow \{\}$.

2. Insert a constant node $v$ for each constant $c \in X_{constant}$, annotated with its value $c[v] \leftarrow c$. (The annotations $x$ and $I$ are not necessary for constants.)

3. Insert an output node $v$ for each $x_{out} \in X_{out}$, annotated with the output variable $x[v] \leftarrow x_{out}$ and $I[v] \leftarrow \{\}$.

4. For each equation $S_i$:
   a) Insert an operation node $v$, annotated with the defined indexed variable $x[v] \leftarrow x_i$, the operation $op[v] \leftarrow op_i$, and the condition space $I[v] \leftarrow I_i$.

   b) If $x_i \in X_{out}$ is an output variable, annotate $I[w] \leftarrow I[w] \cup I_i$ to the corresponding output node $w$.

   c) For each input variable $y_{i,j} \in X_{in} \setminus X_{constant}$ used on the right-hand side of $S_i$, annotate $I[w] \leftarrow I[w] \cup I_i$ to the corresponding input node $w$.

Then, the set of edges $E$ is constructed:

1. For each equation $S_i$ where $x_i$ is an output variable, insert an output edge $e = (v, w)$ from the operation node $v$ corresponding to $S_i$ to the output node $w$ corresponding to $x_i$. Annotate the edge with the indexing matrix $Q[e] \leftarrow Q_i'$ and offset $d[e] \leftarrow d'_i$.

2. For each equation $S_i$ with corresponding operation node $w$ and for each used variable $y_{i,j}$:
   a) If $y_{i,j}$ is a non-constant input variable, insert an input edge $e = (v, w)$ from $v$ to the input node $w$ corresponding to $y_{i,j}$. Annotate the edge with the indexing matrix $Q[e] \leftarrow Q_{i,j}$ and offset $d[e] \leftarrow d_{i,j}$, as well as the argument position $pos[e] \leftarrow j$. 

179
b) If $y_{i,j}$ is a constant $c$, insert a constant edge $e = (v, w)$ from $v$ to the constant node $w$ corresponding to $c$. Annotate the edge with the argument position $pos[e] \leftarrow j$.

c) Otherwise, insert a dependence edge $e_k = (v_k, w)$ to $w$ from each operation node $v_k$ whose corresponding equation $S_k$ defines at least one instance of $y_{i,j}$ used in $S_i$, that is for each $v_k$ with

$$(I[w] - d_{i,j}) \cap I[v_k] \neq \emptyset.$$  

Annotate each edge with the dependence vector $d[e_k] \leftarrow d_{i,j}$ and the argument position $pos[e_k] \leftarrow j$. 

Ein Compiler zur symbolischen Code-Erzeugung für eng gekoppelte Rechenfelder
Diese Dissertation stellt die symbolische Schleifenübersetzung vor, welche der erste durchgehende Ansatz ist, um verschachtelte Schleifen symbolisch auf eng gekoppelte Rechenfelder (engl. tightly coupled processor arrays, TCPAs) abzubilden. TCPAs sind eine Klasse von Schleifenbeschleunigern basierend auf einem Gitter von Rechenelementen. Symbolische Schleifenübersetzung ist:

- **Durchgehend**, weil sie *alle* Schritte der Übersetzung abdeckt, inklusive der Raumzeitabbildung, Codeerzeugung und der Erzeugung aller weiteren notwendigen Konfigurationsdaten für die beteiligten Hardwarekomponenten. Durchgehende Übersetzung ist besonders wichtig für Beschleuniger wie TCPAs, weil eine manuelle Abbildung von Schleifen nicht nur schwierig und aufwändig, sondern vor allem auch fehleranfällig ist.

- **Symbolisch**, weil weder die Schleifengrenzen noch die Anzahl verfügbarer Rechenelemente als bekannt vorausgesetzt während der Übersetzung – sie können also zur Laufzeit frei gewählt werden. Dies ist besonders wichtig für das ressourcenwahre Programmieren, bei dem die Anzahl verfügbarer Ressourcen erst zur Laufzeit bekannt ist.


Im Rahmen dieser beiden Phasen stellt diese Dissertation folgende Beiträge vor:

- **Symbolisches Modulo-Scheduling** ist eine Technik, um Modulo-Scheduling mit Ressourcenbeschränkungen zu lösen, falls die Schleifengrenzen und Anzahl verwendeter Rechenelemente unbekannt sind. Unter anderem zeigen wir, dass dieses Problem latentz-minimal gelöst werden kann, falls die Anzahl verwendeter Rechenelemente im Voraus bekannt ist und beinahe latentz-minimal, falls sie nicht bekannt ist.
**German Part**

- *Polyedrische Syntaxbäume* sind eine platz-effiziente, parametrisierte Darstellung von Programmvarianten, aus denen die notwendigen konkreten Programme zur Laufzeit erzeugt werden können.

- Die *Instantiierung* beeinhaltet Methoden zur Erzeugung von konkreten Programmen und Konfigurationsdaten mit einer Zeitkomplexität, die nicht von den Schleifengrenzen oder der Anzahl verwendeter Rechenelemente abhängt.

- *Run-time requirement enforcement für Schleifen* ist eine Technik, die die durch symbolische Schleifenübersetzung gegebene Flexibilität ausnutzt, um gegebene Anforderungen an nicht-funktionale Eigenschaften zu erfüllen, indem die Abbildung einer Schleife dynamisch angepasst wird; man stelle sich etwa vor, dass die Anzahl verwendeter Rechenelemente so bestimmt wird, dass eine gegebene Latenzschanke eingehalten wird.

Zusammenfassend erlauben die in dieser Dissertation vorgestellten Methoden erstmals die durchgehende symbolische Übersetzung von verschachtelten Schleifen auf TCPAs. Ohne diese Methoden müsste eine gegebene Schleife jedes Mal vollständig neu übersetzt werden, wenn sich die Schleifengrenzen oder die Anzahl verwendeter Rechenelemente ändern. Dies würde die dynamische Abbildung zur Laufzeit unpraktikabel machen sowie selbst konventionelle Übersetzung übermäßig zeit- und platzintensiv.


[BBD+08] Frank Bouwens, Mladen Berekovic, Bjorn De Sutter, and Georgi Gaydadjiev. Architecture enhancements for the ADRES coarse-grained reconfigurable array. In International Conference on High-Performance Embedded Architec-


Bibliography


Bibliography


[PFM*08] Hyunchul Park, Kevin Fan, Scott Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In International Conference on Parallel Ar-


Author’s Own Publications


**Acronyms**

**ADA** (affine dependence algorithm) a recurrence algorithm with exclusively affine indexing functions 37–39, 97

**CGRA** (coarse-grained reconfigurable array) a loop accelerator consisting of a grid of simple, usually non-programmable processing elements 4, 139, 165, 171–173, 177, 203

**CNN** (convolutional neural network) a kind of artificial neural network 17, 22, 165, 174, 176, 177

**FU** (functional unit) a hardware component of a TCPA PE that executes functional programs and contains both an instruction memory and a branch unit 17–19, 27, 44, 46

**MPPA** (massively parallel processor array) an accelerator consisting of a grid of general-purpose processors, usually connected via a network-on-chip 165, 166, 168, 171, 177

**MPSoC** (multi-processor system-on-chip) an entire system with multiple processors on a single chip 30, 31

**MRRG** (modulo routing resource graph) a graph representation of the resources in a CGRA 173

**NoC** (network-on-chip) a routed network that connects on-chip components such as processors and memory 22, 29–31, 166–170, 173, 176

**OIP** (orthogonal instruction processing) a PE architecture where each functional unit executes its own program 12, 16, 18, 25, 26


**RCMSP** (resource-constrained modulo scheduling problem) the problem of finding a latency-minimal, cyclic linear schedule under given resource constraints 63–66, 72, 73, 76–83

**RDG** (reduced dependence graph) a graph representation of a loop program where dependences between equations are made explicit by edges between nodes that represent operations 8, 33, 42, 44–46, 49, 55, 59, 61, 63–67, 69, 70, 72, 76, 82–86, 90–92, 99, 100, 105, 107, 108, 110–113, 156, 157, 179
Acronyms

**RTL (register-transfer level)** a representation of a digital circuit as registers and combinatorial circuits 161, 162

**SoC (system-on-chip)** an entire system on a single chip 29


**TDMA (time division multiple access)** a method for sharing communication channels in a timing-predictable manner 23

**UDA (uniform dependence algorithm)** a recurrence algorithm with exclusively uniform indexing functions 8, 33, 34, 37–40, 42, 44–46, 50, 52–56, 60, 61, 64–69, 81, 82, 92, 96, 99, 110, 111, 136, 137, 157, 179

**VLIW (very long instruction word)** an instruction word containing the individual instructions for multiple functional units 11
access instance, 134
access mapping, 99, 108
adjacency matrix, 21
adjacent (interconnect), 20, 21, 91
affine dependence algorithm, 37
affine schedules, 59
alive (value), 86
anytime instruction processing, 17
approximate computing, 17
architecture description, 161
array single-assignment form, 35
auto tuning, 139
binding possibility, 83
branch instruction, 18, 26, 134
claim (Invasive Computing), 30
complex feasibility constraint, 66, 71, 72, 79
condition node, 156
condition space, 34, 85, 119, 122
constant edge, 45
constant node, 45
control (semantics), 16
control signal, 7, 12, 132
corridor (requirement), 142
coverage, 144, 148
coverage optimization, 147
data (semantics), 16
data signal, 12
definition (variable), 34
dependence, 59
dependence edge, 45
dependence vector, 38, 45, 50, 53, 64, 67, 71, 72, 75, 89, 180
directional ports, 19
domain (RDG), 44
dynamic piecewise linear algorithm, 40, 156
enforcement function, 143
enforcer, 115, 141, 143, 146, 147
everilog space, 127
evaluation (polyhedral syntax tree), 101, 102, 124
feasibility constraint, 63, 64, 66, 72, 79
feasible (schedule), 64
feedback register, 15, 27, 87, 89
finite impulse response filter, 37
fold (polyhedral syntax tree), 127
forward communication only, 73
fragment, 100
fragment node, 101
free schedule, 59
functional instruction, 26, 134
general-purpose register, 6, 15
global allocation, 49, 56
global control signal, 133
hard requirement, 148
head (node), 160
indegree, 160
index space, 34
index variables, 34
index vector, 34
indexed variable, 34
indexing function, 28, 34, 44, 53, 108
indexing matrices, 37
indexing offsets, 37
infect (Invasive Computing), 30
initiation interval, 28, 62, 77, 79, 81, 87, 88
inner loop parallelization, 49
input channels, 27
input domain, 58, 86
input edge, 45, 108
input node, 45
input register, 15, 27, 86, 108
input variable, 39, 42, 53, 108
instantiation, 3–5, 25, 99
instruction template, 18, 105
integer polyhedron, 40
inter-tile dependence vector, 53, 73
inter-tile domain, 122
inter-tile feasibility constraint, 65, 72, 73
inter-tile schedule vector, 61, 72, 73
interconnect wrapper, 12, 19, 28, 91
interference, 87, 88
internal variable, 39
intra-tile dependence vector, 53, 70
intra-tile domain, 119
intra-tile feasibility constraint, 65, 66, 70
intra-tile pattern, 122
intra-tile schedule vector, 28, 61, 66, 70, 80, 115
intra-tile space, 51
invasive (Invasive Computing), 30
invasion controller, 31
Invasive Computing, 30
invasive TCPA, 31
issue (instruction), 103
iteration (vector), 33, 34
iteration space, 33, 34, 160
iteration variables, 34
kernel, 126, 129
kernel class, 129, 130
latency, 59, 63, 64, 76, 141, 151
linear schedule, 67
linearly bounded lattice, 40, 41
live interval, 89
local allocation, 49, 64, 82, 83
local control signals, 133
localization, 38
locally parallel, globally sequential, 95
locally parallel, globally sequential (LGPS), 49
locally sequential, globally parallel, 7, 95
locally sequential, globally parallel (LSGP), 49, 65
loop enforcer, 143
loop extraction, 8
loop tactics, 114
main scanning direction, 70, 72
mapping completion function, 146
memory bank, 23
merge node, 156
merge-compatible, 104
meta node, 101
minimal tile size, 63, 66, 80–82, 93
minimal tile sizes, 66
module (I/O buffer), 23
modulo scheduling, 62
monohedral tessellation, 49
one-domain, 133
operation node, 44, 64, 83, 105
orientation (interconnect), 21
outdegree, 160
outer loop parallelization, 49
output channels, 27
output domain, 58, 86
output edge, 45, 108
output node, 45
output register, 15, 86, 108
output variable, 39, 42, 53, 108
outset, 96
parameter (iteration space), 35
partial control signal, 132
partition (I/O buffer), 23
partitioning, 49
PE port, 19
piecewise linear algorithms, 37
piecewise regular algorithms, 38
polyhedral, 105
polyhedral fragment, 101
polyhedral functional unit program, 102
polyhedral instruction, 102
polyhedral model, 8, 33
polyhedral operand, 102, 106
polyhedral syntax tree, 4, 5, 99–101, 105, 116, 120, 121, 129
polyhedral time slot, 102
Presburger language, 41
Presburger set, 33
processor classes, 116, 121
programmable accuracy, 17
projection (allocation), 49
propagation channel, 27, 90, 91, 110, 116
read space, 84, 85, 107
recurrence algorithm, 33, 34
reduced dependence graph, 33, 42, 64, 83, 99, 105, 179
reduced resource-constrained modulo scheduling problem, 66, 76
reduced topology graph, 91
reduction theorem, 63, 66, 80
referentially transparent, 35
regular iterative algorithms, 38
requirement, 142
resource constraint, 62, 63, 79
resource graph, 83
resource-constrained modulo scheduling, 50, 62
resource-constrained modulo scheduling problem, 62, 63
retreat (Invasive Computing), 30
run-time requirement enforcement, 2
run-time requirement enforcement on loops, 143
run-time requirement enforcement on loops with secondary objectives, 143
satisfy (requirement), 142
schedule, 35, 59
semantics (data/control), 12
sequential scanning, 66
short dependences, 50
simple feasibility constraint, 66, 71, 72, 79
single instruction, multiple data, 17
sink (node), 160
sink (port), 21
slot (kernel), 126
soft requirement, 148
software pipelining, 50, 61
source (node), 160
source (port), 21
space mapping, 56, 82, 100, 110, 142
space-time mapping, 35, 46
specialization, 119, 120
statement instances, 33
symbolic, 50
symbolic configuration, 3, 4, 25, 99, 108, 115, 163
symbolic loop compilation, 3, 115, 141
symbolic mapping, 3, 99
symbolic modulo scheduling, 4, 8, 50, 62, 63, 82, 99, 105
symbolic tiling, 49, 74
systems of affine recurrence equations, 37
systems of uniform recurrence equations, 33, 34, 38
tail (node), 160
TCPA, 1, 11
temporal distance, 27
tensor comprehensions, 114
three-address code, 17
tight (schedule), 66
tile space, 51
tiled dependence vector, 53, 76
tiled feasibility constraints, 65
tiled iteration space, 51, 76
tiling, 49
Index

tiling matrix, 51
transition space, 116, 131

unfolded dependence graph, 45
uniform dependence, 38
uniform dependence algorithm, 33, 38, 99
unions of linearly bounded lattices, 41
use (variable), 34

variable instance, 34
virtual register, 15, 92

weakly programmable processor array, 7
write space, 85, 91, 106