Code Generation for
Tightly Coupled Processor Arrays

Codegenerierung für eng gekoppelte Prozessorfelder

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

vorgelegt von

Srinivas Boppu
aus Manakondur, Telangana, India
Als Dissertation genehmigt
von der Technischen Fakultät
der Friedrich-Alexander-Universität Erlangen-Nürnberg
Tag der mündlichen Prüfung: 18. December 2015

Vorsitzende des Promotionsorgans: Prof. Dr. rer. nat. Peter Greil

Gutachter: Prof. Dr.-Ing. Jürgen Teich
Prof. Dr. sc.techn. Andreas Herkersdorff
This thesis is dedicated to my wife Anusha Gundu with love and gratitude.
“You need Power, only when you want to do something harmful otherwise Love is enough to get everything done.”
-Charles Chaplin
Acknowledgments

First of all, I would like to thank my doctoral adviser, Prof. Dr.-Ing. Jürgen Teich for giving me this opportunity to pursue PhD at his chair, support, and technical feedback during my research. I would like to thank Prof. Dr. sc.techn. Andreas Herkersdorf for being the co-examiner of my thesis. I would also like to thank our group leader, Dr.-Ing. Frank Hannig for all the scientific discussions, motivation, and constant encouragement in completing this work.

I would like to thank everyone in the chair of Hardware/Software Co-Design for their support, and advice while carrying out this research work. In particular, I would like to thank all members of the InvasIC group—Vahid Lari, Alexandru Tanase, Ericles Sousa, Andreas Weichslgartner, Sascha Roloff, and Shravan Kumar Muddasani for all the technical discussions, and friendly chats during the lunch.

I would also like to thank Infineon Technologies Asia Pacific Pte. Ltd. and Economic Development Board of Singapore for their support during my master studies which paved the way for further pursuing PhD. I would like to thank Naveen Karnati, Srinivas Mudumba, Sugi, Andavar Diravilamani, and Siva Prakash who supported and motivated me to pursue PhD.

I would also like to thank all my friends in India who supported and encouraged me throughout my career. In particular, Sabarigirivasan Shannugam, Dileep Maddukuri, Pranod Manda, Rajanikanth Errabelly, Swetha Madadi, and Kushagra Khorwal for their unconditional support and love.

I would like to thank all my family members who deserve the maximum credit for this work. My father Kanakaiah Boppu and mother Laxmi Boppu always supported me and took extraordinary interest in my education since my childhood. Finally, I would like to thank my wife Anusha Gundu and daughter Aditri Boppu for their encouragement, support, endless patience, and love.

Srinivas Boppu
Erlangen, December 2015
# Contents

1 Introduction
   1.1 Evolution, Challenges, and Trends in System-on-Chip (SoC) Design 5
   1.2 Heterogeneous Multi-Processor System-on-Chip (MPSoC) 7
   1.3 Next Generation Embedded Applications 9
   1.4 Accelerator-based MPSoCs 11
   1.5 Programmable Hardware Accelerators 14
   1.6 Programming Flow 16
   1.7 Problem Definition 20
   1.8 Contributions of the Thesis 21
   1.9 Thesis Organization 23

2 Tightly Coupled Processor Arrays 25
   2.1 Tightly Coupled Processor Array (TCPA) 25
      2.1.1 Processing Element (PE) 26
      2.1.2 Reconfigurable Inter-Processor Network 30
      2.1.3 Configuration Management 33
   2.2 TCPA Editor 34
   2.3 TCPA: Programmable Accelerator Perspective 35
   2.4 Related Work 38
   2.5 Summary 46

3 Compiler Framework 47
   3.1 Loop Parallelization in the Polyhedron Model 47
      3.1.1 Preliminaries: Algorithm Specification 48
   3.2 Compilation Flow 57
      3.2.1 Front End 57
      3.2.2 High-Level Transformations 59
      3.2.3 Partitioning 61
      3.2.4 Space-Time Mapping 64
      3.2.5 Back End 70
Abstract

In this dissertation, we consider techniques for automatic code generation and code optimization of loop programs for programmable tightly coupled processor array targets. These consist of interconnected small light-weight very long instruction word cores, which can exploit both loop-level parallelism and instruction-level parallelism. These arrays are well suited for executing compute-intensive nested loop applications, often providing a higher power and area efficiency compared with commercial off-the-shelf processors. They are ideal candidates for accelerating the computation of nested loop programs in future heterogeneous systems, where energy efficiency is one of the most important design goals for overall system-on-chip design. In order to harness the full compute potential of such an array, we need efficient compiler techniques which can automatically map nested loop programs onto them. Such a compiler framework is essential for increasing the productivity of designers as well as for shortening development cycles. In this context, this dissertation proposes a novel code generation and compaction approach which generates the assembly-level codes for all the processing elements in an array from a scheduled loop nest. The code generation approach itself is independent of the array size, preserves the given schedule, and is independent of the problem size. As part of this compiler framework, we also present a scalable interconnect generation approach where the connections among different processing elements are automatically generated from the same scheduled loop program. Furthermore, we consider the integration of a tightly coupled processor array into a multi-processor system-on-chip: Here, we propose the design of new hardware components such as a global controller, which generates control signals to orchestrate (synchronize) the programs running on the different processing elements, and address generators, which are required to generate the address as well as enable signals for a set of reconfigurable I/O buffers surrounding the processor array. We propose a fully programmable design of these required hardware components and add the required compiler support to generate the configuration data from the same scheduled loop program as well. In summary, the major contributions of this dissertation enable and ease the fully automated mapping of nested loop programs onto tightly coupled processor arrays.
An embedded system is a computer system designed to perform one or a few dedicated functions, often with stringent constraints such as high reliability, high performance, low cost, and low power. Such systems are omnipresent in our everyday life, in industrial applications, and in the consumer market. The underlying computing engines of these embedded devices have evolved to become complex System-on-Chip (SoC) architectures that integrate several processors of different type as well as dedicated and programmable hardware accelerators. These heterogeneous Multi-Processor System-on-Chip (MPSoC) platforms are nowadays defacto standard for embedded system design due to their high energy efficiency offered by different special purpose processors and accelerators. However, the software which is getting deployed on these systems is getting really complex. The software can no longer be characterized as a single simple application but rather as a combination of multiple complex applications having their own constraints which have to be satisfied together with the other applications that are running on the system. In general, the control-intensive software which is sequential in nature is executed typically by general purpose processors whereas data-or compute-intensive applications may be offloaded to special purpose processors or accelerators.

In order to harness the full compute potential of today’s complicated heterogeneous MPSoCs and to reduce the software productivity gap [Bor99], resource-aware programming paradigms are inevitable. These paradigms support application or software programmers with an abstract view of the architecture such as available processors and how heterogeneous they are, communication as well as memory resources by various hardware/software middleware layers. The motto here is that knowing about the hardware will make us a way more efficient in software. In fact, according to the roadmap of ITRS 2011 [ITR11], heterogeneous parallel processing and improving the programming of parallel architectures are identified as the grand challenge in the years to come. In order to optimize one or more design objectives, efficient methodologies for design space exploration and simulation frameworks are absolutely required for programmers to explore and compare a wide range of mapping alternatives during the early design stage. Furthermore, integration and verification times account for a large portion of
development time of an MPSoC, techniques such as component reuse, synthesis of dedicated accelerators from a high-level language or generating codes for programmable hardware accelerators are proposed—to have the early success, correct-by-construction, and short development cycles. However, it is the task of the programmer to handcraft the application in a way that parts of it having data-intensive parts get mapped to programmable hardware accelerators to get high performance at low power.

In this realm, this dissertation deals with handling the design complexity and increasing productivity for mapping compute intensive loop programs onto programmable hardware accelerators, realized by highly parameterizable Tightly Coupled Processor Arrays (TCPAs) as proposed in [KHKT06b, KHKT06a]. The ubiquitous consumer markets such as mobile devices (smart phones, tablets) require most of the time the deployment of computationally intensive nested loop kernels to process signal, image, and video data at much higher energy efficiency—justifying the use of loop accelerators. In particular, programmable accelerators are of interest as these can be configured at run-time with different applications by just changing programs and other control configuration data. The correct-by-construction mapping approach should consider the available parallelism in the accelerator, exploit partitioning, efficient data reuse, transfer, storage and other transformations, which are often a daunting and intricate task [Dut11].

In this dissertation, we address all the needs for mapping a given nested loop application or algorithm, specified in a high-level language, onto tightly coupled processor arrays. In this context, we extend an existing high-level synthesis framework called PARO [HRDT08] to schedule and transform a given nested loop. This requires to automatically generate the instruction sequence for each Processing Element (PE) of a given array as well as generation of the interconnection topology configuration among them. We also focus on designing two hardware components for the array: The first one is essential to synchronize the execution of the PEs. The second one is responsible for delivering the right data to the PEs by generating address and enable signals for I/O buffers encompassing the array border. We also generate the required configuration data automatically within the compiler framework for these two components. These two components are essential to integrate the TCPA core into a more traditional accelerator-based MPSoC designs.

In this chapter, we first discuss the evolution, challenges and future trends of SoCs and state of the art heterogeneous MPSoCs in Section 1.1 and Section 1.2, respectively. The landscape of embedded computing and the next generation applications are discussed in Section 1.3 followed by the motivation for accelerator-based MPSoCs and different types of accelerators as well their characteristics in Section 1.4. Programmable hardware accelerators are discussed in Section 1.5. In Section 1.6, overview of the challenges in compil-
1.1 Evolution, Challenges, and Trends in System-on-Chip (SoC) Design

The Metal–Oxide–Semiconductor Field-Effect Transistor (MOSFET) has been the workhorse for decades, scaling in performance according to the famous Moore’s law and providing the foundation for today’s unprecedented compute performance. In the last two decades, transistor scaling in performance and shrinking in size enabled the steep progress of the semiconductor industry and tremendous growth in compute performance by providing the following three major benefits [Bor99]. First, the feature sizes of the transistors are scaled by 30\% (0.7\times), resulting in area shrink of 50\% (0.5\times), doubling the transistor density every technology generation—the fundamental reason behind Moore’s law. Second, as the transistor is shrinking in size, its performance increases by about 43\% (0.7 \times \text{delay reduction}, or 1.4 \times \text{frequency increase}), providing higher system performance. Third, to keep the electric fields constant everywhere in the transistor to maintain the reliability, the supply voltage is reduced by 30\%, or power by 50\% [BC11]. Supply voltage scaling has been rewarding since it reduces the active power, allowing us to use the transistor budget to employ complex microprocessor architectures to deliver performance. But the threshold voltage of the transistor has to scale to deliver circuit performance, and as the threshold voltage scales, the transistor sub-threshold leakage increased exponentially. Today, this sub-threshold leakage power amounts to almost 40\% of the total power of microprocessor, and the threshold voltage scaling has to stop or slow down [BC11]. Putting it all together, Moore’s law coupled with Dennard scaling (transistors get smaller their power density stays constant by scaling down both voltage and current) has resulted in consistent exponential performance increase till last decade.

On the contrary, green computing, which is formally defined as the environmentally responsible and eco-friendly use of computers and their resources, is compelling for higher energy efficiency. Therefore, many manufacturers and vendors are continuously investing in designing energy-efficient computing devices. In the era of current Deep Sub Micron (DSM) process technologies, with the end of Dennard scaling, technology scaling can sustain the doubling the tran-
1. Introduction

Transistor density count every generation, but with significantly less improvement in transistor switching speed and energy efficiency [BJS07]. In other words, it is difficult to increase the clock frequencies in the future as this results in huge power consumption and heat problems. This transistor scaling trend indicates a divergence between energy efficiency gains and transistor density increases. Therefore, the recent shift to multi-core designs, aimed to continue proportional performance scaling by utilizing the increasing transistor count to integrate more cores, which leverage application and/or task parallelism while meeting the power budgets [Dav05].

Despite the many advantages of that multi-core processor comes with, the amount of performance gained by the use of a multi-core processor is strongly dependent on the software algorithms and implementation [PDG06]. In particular, the possible gains are limited by the fraction of the software that can be "parallelized" to run on multiple cores simultaneously; this effect is described by Amdahl's law [Amd67]. In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores. Many typical applications written for single cores, however, do not realize such large speedup factors and thus, the parallelization of software is a significant ongoing topic of research. Furthermore, increasing the number of cores will result in so-called dark silicon—fraction of cores that must remain passive, or "dark" in order to stay within a chip's power budget [GHSZ+12, Tay13]. Dark silicon describe the growing gap between how many transistors that can be crammed into a chip with each advancement in lithography versus how many transistors can be actually used simultaneously given the power budget—the gap between area gains and power gains [GHSZ+12]. It is predicted that going down to the 8 nm process, the amount of dark silicon rises to 51% for the same area of the chip [HER+13, Tay12]. In the future, only the customization and heterogeneity among the cores can improve energy efficiency—the new fundamental limit for performance.

Based on the scaling issues of manycore processors, predicted developments and bottlenecks in process technologies, the following trends will be observed in the design of future MPSoCs. There will be a shift towards simple, low frequency, low complexity cores, coupled with an increase of the core count to the levels of several hundreds or even thousands. Figure 1.1 shows the trend in the predicted number of cores in an MPSoC. Heterogeneity in the core capabilities will continue and a vast amount of hardware accelerators including custom as well as programmable hardware accelerators will be integrated into a single SoC [CK08]. Aggressive power optimization mechanisms such as power gating and Dynamic Voltage and Frequency Scaling (DVFS) will influence the design methodologies. Unless there is a revolutionary new method of designing SoCs and deus ex machina, heterogeneous MPSoC systems will predominate in the future.
1.2 Heterogeneous Multi-Processor System-on-Chip (MPSoC)

In the future, SoC performance is mainly driven by multiple cores and customization. Multiple cores can increase the computational throughput, and customization can reduce the execution latency. Clearly, these both techniques—multiple cores and customization—can improve energy efficiency [BC11]. For instance, an application can have varying resource requirements during the different stages of its execution. One stage might have a large amount of Instruction-Level Parallelism (ILP), which can be exploited by a core that can issue multiple instructions per cycle. The same core, however, might be inefficient with little or no ILP and may consume significantly more power than a standard core.

The hardware customization includes fixed-function accelerators (such as cryptography engines, multimedia codecs), programmable accelerators, and even dynamic customizable logic. The block diagram in Figure 1.2 illustrates such a heterogeneous MPSoC architecture. In general, customization in the form of hardwired or customized functional units, customized wiring/interconnect for data movement, and reduced instruction overheads will improve the computational performance. In some cases, units hardwired to a particular data representation or computational algorithm can achieve a 50×-500× greater energy efficiency than a general-purpose register organization [HGW+10]. Future smartphone SoCs might include as many as 10 to 20 accelerators to achieve...
1. Introduction

![Diagram of MPSoC architecture](image)

**Figure 1.2:** A generic block diagram of a heterogeneous MPSoC architecture containing multiple processors (CPUs), a DSP, a GPU, an ASIP, custom as well as programmable hardware accelerators, connectivity IPs (e.g. USB, UART, …), memories.

A superior balance of energy efficiency and performance. The hardware customization may include vector processing capabilities of GPUs which enable them to perform parallel operations on very large sets of data at much lower power consumption relative to the serial processing of similar data sets on CPUs. Commercial examples of heterogeneous systems include IBM’s Cell Broadband Engine [PAB05, KDH05] and TI’s DaVinci [Tex]. In IBM’s Cell Broadband Engine, eight 128-bit Synergistic Processing Elements (SPEs) optimized for floating-point vector algebra and a 64-bit master Power processor to run the operating system, divide up tasks are connected over a bus. Similarly, special DSP engines are incorporated in TI’s DaVinci architecture which comes along with an ARM core. These architectures are especially suited for multimedia and embedded applications. AMD’s "LLANO" fusion Accelerated Processing Unit (APU) [BFS12] and Intel’s Sandy Bridge [RNR12] are examples of integrating a GPU and CPU on a single chip to provide the best of both worlds.

The most important problem, however, related to manycore processors, i.e. how to efficiently exploit the nominally abundant processing power of the avail-
able cores, is still unsolved \cite{HHB+12}. It is a widely tackled research topic in industry and academia how applications can efficiently use heterogeneous manycore processors. This involves developing resource-aware programming paradigms, compilers and supporting the application programmer with the available cores and how heterogeneous they are, communication as well as memory resources by various hardware/software middleware layers to write parallel and resource-aware programs.

1.3 Next Generation Embedded Applications

As aforementioned, embedded computing systems are omnipresent including industrial, automotive, consumer, medical and mobile applications. They not only require lots of computation but also must meet quantifiable goals such as real-time performance, power and cost. Depending on the application domain, these goals and constraints for embedded systems are derived. For instance, mobile and portable embedded systems have to be power-efficient since they are often constrained due to limited battery capacity. In safety-critical embedded systems, non-functional requirements such as safety and reliability are utmost important since these systems are often involved with people. Furthermore, applications or underlying algorithms from the domains of image and signal processing always show different levels of parallelism including task level, loop level and operation level parallelism. Therefore, the embedded systems in this application domain must exploit the inherent parallelism available in the applications. There are many other characteristics of embedded systems that are wishful such as cost effectiveness, fail-safe, small size, just enough performance and real-time requirements. These characteristics of the applications and quantifiable goals makes the design of embedded computing systems a very different experience than the design of general-purpose computing systems, in which we cannot predict the use of it in advance. It is really a challenging task for system architects to combine these characteristics and propose a good hardware architecture for a given application domain.

In the following, a brief explanation about a simple yet compute-intensive application from the medical imaging domain is given to emphasize the required computational power. Typical medical applications are usually subdivided into three sub tasks: preprocessing (low level of processing—preliminary operations used to filter noise, and to eliminate the unnecessary or unwanted details), data reduction (mid level processing—extraction of useful information, segmentation), feature analysis (high level processing—ensuring semantic meaning extraction) \cite{Dou09}. Let us consider a simple application which is used for enhancement of images received from a Magnetic Resonance Imaging (MRI) device. The advantage of an MRI scan is that magnetic resonance signal is
1. Introduction

![Figure 1.3:](image)

Figure 1.3: An example application from medical image processing showing preprocessing of images using an unsharp mask filter for better diagnosis. Image source: http://www.cdc.gov

able to penetrate radio opaque tissues (e.g., bones) with little attenuation or distortion. It also offers excellent soft tissue contrast and low noise. However, the level of noise depends on the acquisition time. This means that real-time images will contain a higher level of noise than non real-time images. Hence, a basic operation that can be done on MRI is noise removal and sharpening of the boundaries between regions. An unsharp mask filter can be used to reduce the noise and sharpen the edges, contrary to what its name might mean, as shown in Figure 1.3. It performs a 2D high-pass filter by subtracting a low-pass filtered image from the original and adding the result to the original image. It smoothens homogeneous regions and highlights edges. This improved sharpened image will help the doctors in diagnosis. It is to be noted that it is difficult to figure out how to enhance just the right structures to provide the optimum diagnostic presentation to the doctor. Therefore, medical imaging manufacturers continue to devise robust image processing algorithms that provide a net benefit in the applications in which they are used. These applications might need huge performance up to teraflops ($10^{12}$ operations per second) with stringent power constraints [Dut11]. Many of these applications are already shaping future architectures.

The recent switch to multi-core in embedded computing allows the execution of many compute intensive algorithms efficiently by boosting their performance manyfold. An application may contain multiple dwarfs\(^1\) [ABC+06] that each

---

\(^1\)A *dwarf* captures a pattern of computation and communication to a class of important applications.
consume a significant percentage of an application’s computation. On a multiprocessor platform, these dwarfs can be either distributed temporally or spatially and may require translation of data structures. Hence, there is a drastic need for new hardware architectures, memory technologies, communication protocols as well as interconnection topologies, programming models, and software (compilers and operating systems) to pave the way for new applications. These applications are also characterized by the presence of multiple computationally intensive loop algorithms. In the next section, we explore state-of-the-art architectures, which cater to the needs of these applications.

1.4 Accelerator-based MPSoCs

In light of diminishing performance gains from process scaling, a paradigm shift is necessary to sustain insatiable performance improvements in future technology generations without exceeding the power budgets. Much of the high power/performance cost associated with computing in general purpose cores is due to control flow and data movement overhead. In this realm, customization in the form of hardware accelerators could eliminate the majority of these costs and offer a viable solution. These are hardware components that work together with the processor and executes some specific functions much faster than the processor. An accelerator appears as a device on the bus in contrast to a coprocessor which connects to internals of the CPU and executes instructions. An exemplary accelerated system architecture is shown in Figure 1.4 including a simple accelerator architecture. Accelerators can be of different type including an Application-Specific Integrated Circuit (ASIC), a Field-Programmable Gate Array (FPGA), or standard components such as a GPU.
1. Introduction

Accelerators have become prevalent in industrial SoC designs for their low power, high performance potential. There is a growing acceptance of accelerator-based architectures in the complete spectrum of computing, starting from supercomputing, mainstream desktop computing, to embedded computing. According to the recently published Green500 list in June 2014, top 15 super computers are heterogeneous architectures for the first time which contain many accelerators [Com14]. TSUBAME-KFC is a supercomputing system developed at the Tokyo Institute of Technology in Japan, tops the list with an energy efficiency of 4.5 GFLOPS/watt. Each computational node within TSUBAME-KFC consists of two Intel Ivy Bridge processors and four nVidia Kepler GPUs which function as accelerators. The complete system consists of 40 nodes delivering a peak performance of 210.61 TFLOPS. Similarly, in mainstream desktop computing, CPU and GPU are integrated to deliver better performance for various application classes, increased battery life in case of notebooks, and lower cost. AMD Fusion APU LLANO [BFS12] contain four x86 cores and a GPU. Intel Haswell [HKO14] integrates up to eighteen cores, a GPU and an on-die power management control unit. The Fermi architecture from nVidia offers up to 512 CUDA cores and special features for gaming and high-performance computing [WKP11]. The Cell processors used in Sony PlayStation 3 gaming consoles are another major player in the mainstream acceleration arena. For embedded computing, particularly in mobile processors, accelerators bring significant additional performance to an SoC. TI OMAP, Samsung Exynos, Qualcomm Snapdragon, nVidia Tegra, and Apple A7 are some of the known SoC systems catering to the needs of the the so-called smartphone mobile industry. These smart mobile SoCs are the leading product driver for the semiconductor industry [Yea13]. Normally, applications running on these SoCs are made up of two parts: control and computation. Usually, the computation kernels are mapped onto accelerators while control parts, which can not be parallelized, are executed on a CPU. The computational efficiency or power efficiency of these accelerators comes from by exploiting some of the following features: (a) exploiting properties of the applications running on them such as parallelism at different levels, (b) simple cores (c) run-time reconfiguration, (d) customization, (e) different levels of memory hierarchy, and (f) fine grain power control mechanisms.

Parallelism can be uncovered at different levels: Instruction-Level Parallelism (ILP), Loop-Level Parallelism (LLP), Thread Level Parallelism (TLP), and Data Level Parallelism (DLP). For each type of parallelism there is a corresponding accelerator architecture. ILP expresses the possibility of executing more instructions at the same time. There exists several architectures that exploit ILP; however, the most common ones are: pipelined, super scalar, and Very Long Instruction Word (VLIW) architectures. LLP is present when consecutive loop iterations can be executed in parallel. One way of exploiting LLP is to converting it to ILP by employing techniques such as loop unrolling and
1.4 Accelerator-based MPSoCs

Software pipelining. Task level parallelism can be exploited by an operating system or hardwired scheduler, for instance, in a multiprocessor accelerator. The idea here is that multiple threads executed in parallel on multiple processors. Single Instruction Multiple Data (SIMD), systolic, and vector machines take the advantage of DLP [FK97, Kun03]. In SIMD computation, the same instruction is performed in parallel on a set of operands. SIMD computation is very popular in the domains of multimedia processing. Recent studies show that a deep execution pipeline, branch prediction, out-of-order execution have diminishing returns on performance, which was outweighed by power and energy usage [GH96]. Therefore, the processor cores used in accelerators have to be simple which can be designed and verified easily [ABC+06]. Accelerator architectures rely on a simplified data path without functions like data cache or prefetching, and have multiple small processing elements with partly general and specialized or customized data path. The processing elements have less control overhead, as each processing element synchronously executes a copy of the same program [Dut11]. In certain accelerator architectures, an application-specific instruction set and data path at synthesis time can be configured or at runtime, instruction set can be reconfigured based on the application properties that may vary during runtime [HBHG11]. Reconfigurability is an excellent means of combining the performance of a hardware circuits with the flexibility of programmable architectures. To deal with huge access latencies to the main memory, accelerators often have some kind of scratch pad memory or local memory. In some accelerators, to hide the data latency, double buffering schemes are also employed to avoid any performance hindrance due to data starvation. Fine-grained power gating mechanisms [LMB+12a] and DVFS mechanisms [MBF+12] are employed in accelerators to reduce the power consumption.

In embedded computing, accelerators come in different forms and can differ a lot from each other: differences can relate to the purpose for which they are designed (accelerators can be specifically designed to implement a single application—non-programmable or can support a broad range of different applications—programmable), their implementation technology (ASIC, FPGA), the way they interface to the rest of the system, and of course their architecture [FKDM09]. Figure 1.5 shows the power efficiency versus flexibility for different accelerators used in embedded computing. Non-programmable hardware accelerators have the highest power efficiency at the cost of flexibility, as they neither support hardware nor software programmability. DSPs and ASIPs are having fixed architectures, thus performance improvements can only come from software optimization. FPGA solutions support only the hardware programming model, i.e., configuration is done at the hardware level. Programmable accelerators provide both a software programming model as well as a hardware programming model; therefore they achieve performance of more than 100 times that of an off-the-shelf, general-purpose embedded processor. In this disserta-
1. **Introduction**

In this section, we focus on programmable accelerators which are briefly described in the next section.

![Energy Efficiency versus Flexibility Trade-Offs](image)

**Figure 1.5**: Energy efficiency versus flexibility trade-offs of different accelerator alternatives in embedded computing. Adapted from: Jan Rabaey, UCBerkeley [Rab00].

### 1.5 Programmable Hardware Accelerators

In the semiconductor industry, dealing with change has become more important than ever before. Often there are emerging protocol standards, changing product requirements, specifications, and bug fixes. Therefore, a product has to support these requirements without a costly re-spin. This creates conflicting requirements for the product. On one hand, the product needs to be customized to achieve performance close to that of an ASIC. On the other hand, there is a need for the flexibility of a programmable solution, where modifications and enhancements can be done by software changes rather than by costly re-spins of the hardware. In this realm, programmable hardware accelerators are ideal candidates which provide this flexibility.

These programmable hardware accelerators are often implemented as a Massively Parallel Processor Array (MPPA) as shown in Figure 1.6. An MPPA consists of a two-dimensional mesh of PEs having Arithmetic Logic Units (ALUs), an instruction memory, local data storage that feed the array in terms of data. A PE is equipped with one or more register files to store data and instructions, and compute the result and send the output to its neighbors to compute the next step. PEs are interconnected by a reconfigurable interconnect. In
order to change the functionality of the MPPA, instruction memory is populated with a new program as well as interconnections among PEs are reconfigured. Coarse-grained reconfigurable architectures share a similar concept. However, the basic building block of these architectures is an ALU, while that of MPPAs is a whole PE. Coarse-grained reconfigurable architectures, see Section 2.4 for a detailed review of existing architectures, typically consist of ALUs connected by a reconfigurable interconnection infrastructure which can be reconfigured according to the functionality. Since coarse-grained reconfigurable architectures have ALUs—not generic processors—as their primary primitive, they are only amenable to the implementation of data-path-dominated algorithms, not control-oriented algorithms [SML09]. In this dissertation, we always refer to coarse grained reconfigurable architecture as Coarse-Grained Reconfigurable Array (CGRA), MPPA and use the term often interchangeably. With minimal instruction overhead, explicit software pipelining, and static scheduling, CGRAs [Rab00, KSHT09] have been shown to achieve power efficiencies of up to 10-100 MOPS/mW. This is about two orders of magnitude higher than the Intel core i7 (quad core) processor, which has a peak performance of 45 GOPS, but consumes 130 W of power, providing a power efficiency of 0.347 MOPS/mW [FKDM09]. Owing to their excellent power-efficiency, MPPAs have been used in embedded systems for streaming applications. In this dissertation, we will focus on so-called Tightly Coupled Processor Array (TCPA) [KHKT06b]—a customizable and programmable hardware accelerator architecture which target the parallel execution of nested loop programs. A TCPA (see Chapter 2) is an MPPA that exploits the ILP of a loop

**Figure 1.6:** A massively parallel processor array showing PEs interconnected in a mesh form, internal structure of a PE, main memory, local memory, and configuration memory. (Image source: adapted from [KLSP10])
1. Introduction

body by having a VLIW architecture inside each PE and LLP by having multiple PEs executing more than one iteration using software pipelining technique. However, one of the biggest challenges for MPPAs is application mapping, or compilation. An MPPA compiler is much more complex than a regular compiler, since in addition to the regular task of “converting high level programs to machine instructions,” an MPPA compiler must: (a) exploit the parallelism available in the array (ILP and LLP), (b) map operations to PEs in space as well as time, (c) route data among the PEs without violating any data dependencies, and (d) generate configuration data for all the required components in the array. When we program a general-purpose processor, the code just contains the “application” expressed in terms of the instruction set, and all these tasks are automatically managed by the processor hardware. In contrast, this has to be explicitly done in the application code for MPPAs, and therefore compilation for MPPAs is quite tough [KLSP10]. Hence, we need design methodologies and compiler techniques to automatically map applications and generate all required codes and configuration data for an MPPA. These methodologies and frameworks will also increase the productivity of the application programmer. In the next section, a brief explanation of existing programming models, design methodologies, and compiler frameworks for both MPSoCs and programmable hardware accelerators is given.

1.6 Programming Flow

The discussions in previous sections made clear that programming MPSoCs is a challenging task in embedded system design. There is an increased research focus both in academia and in industry to develop design methodologies, compilers and new programming paradigms to map applications onto these MPSoCs automatically. To fully unleash the power of MPSoC architectures, new tools are needed which will take care of the increasing complexity of the software running on these architectures. A generic MPSoC programming flow is shown in Figure 1.7. It shares similarities with the well-known Y-chart approach used in the system synthesis community [KDvdW97]. In this idealized flow, a mapper takes as input a set of applications together with their constraints and produces the software, i.e., code that could run on a given target platform. The architecture model is an explicit input to the mapper which ensure that a particular combination of applications are running simultaneously adhering to their constraints.

The programmer can tweak the applications if the design objectives (such as performance, power, and cost) are not met after executing them. Therefore, a feedback path from design objectives to applications as shown in Figure 1.7. However, as mentioned before, knowing the hardware or architecture while pro-
1.6 Programming Flow

programming itself makes us a far better in generating the software. In a recently proposed resource-aware parallel programming paradigm called invasive computing [THH+11], an abstract architecture view with sufficient details is made available to the programmer so that while programming the application, he/she can write different variants of the application. This helps the mapper which is essentially an MPSoC compiler, formally defined as—a tool-chain to tackle the problems of expressing parallelism in applications' modeling/programming, mapping/scheduling and generating the software for a given MPSoC architecture [LSC10]. This definition of an MPSoC compiler is subtly different from the term software synthesis as it is understood in the hardware/software co-design community [Tei12, TH07]. In this context, software synthesis emphasizes that starting from a single high-level system specification, the tools perform hardware/software partitioning and automatically synthesize the software part so as to meet the system performance requirements of the specifications [LSC10]. The flow is also called an application-driven “top-down” design flow. The MPSoC compiler is used mostly in platform-based designs where the programmers have to develop the software for a given platform. An MPSoC compiler not only translates applications written a in high-level programming language into the machine binary code but also performs complex tasks such as partitioning/mapping. Though software synthesis and MPSoC compilers share some similarities (both generate code), the major difference is that they exist in different methodologies’ contexts, thus focusing on different objectives [CSL11].

![Figure 1.7: MPSoC programming flow, similar to well known Y-chart approach [KDVvdW97]](image)
1. Introduction

On one hand, an MPSoC compiler enforces the programmer to write applications using parallel programming models so as to best utilize the hardware resources but thinking in parallel is often a difficult task for programmers. On the other hand, the heterogeneity of MPSoC architectures requires the compilation process to be ad-hoc. The programming models for different architectural components of the MPSoC platform can be different as these components may differ in the granularity of the parallelism that they offer. The compiler tool-chains may come from distinct vendors for each distinct architectural components and Network on Chips (NoCs). Putting it altogether, MPSoC compilation is an extremely sophisticated process which justifies the increased research focus in this area.

In this dissertation, we focus on the compilation of compute-intensive loop programs onto TCPAs that will be introduced in Chapter 2. Figure 1.8 shows the mapping flow or compiler framework followed in this dissertation. Nested loop algorithms described in a high-level language are provided as input to the compiler. After parsing the input, algorithms are internally represented as Reduced Dependence Graph (RDG) in which the data dependencies between the statements of the algorithm are captured. The architecture description of the processor array is also provided as an input to the compiler which include information such as number of PEs in the array and functional units within each PE and their type.

The compiler front end contains a high-level source-to-source transformation tool box. Transformations like loop unrolling and loop tiling are applied here first for parallelization. A major task of the compiler is to exploit architectural features like multiple levels of parallelism supported in the architecture (ILP through multiple functional units per PE and LLP through concurrent execution of multiple loop iterations on the processing elements). Loop transformations in combination with scheduling map a nested loop program onto a given processor array (allocation). During scheduling, functional unit binding also takes place where each statement of the nested loop program is bound to a functional unit inside the PE [Han09]. In the back end of the compiler, assembly code sequence for each PE as well as array interconnect configuration code are generated automatically. Furthermore, the entire static control flow of the loop algorithm is captured by a global controller (see Figure 1.8), which generates necessary control signals to orchestrate sequencing and synchronization between the PEs, has to be configured for each application. The processor array also contains small local reconfigurable I/O buffers surrounding it, which feed the border PEs with data. However, these buffers require addresses and enable signals that determine from where and when to read from a connected buffer. To deal with this issue, separate Address Generators (AGs) (see Figure 1.8) are instantiated, which also have to be configured depending on the application. In the compiler back end, configuration data for the global
Figure 1.8: Overview of compiler flow for mapping nested loop programs onto a Tightly Coupled Processor Array (TCPA).
controller and the address generators are also generated automatically. The programming model and TCPA architecture are vital for our compiler framework. Therefore, an elaborate groundwork on background, fundamentals, and related work on programming models for accelerators and TCPA architecture are presented in the next chapters. In this dissertation, we tackle different problems while mapping applications onto TCPAs—more specifically in (a) automatically generating code and interconnect configuration for each PE, and (b) designing and generating the configuration data for global controller as well as address generator. These problems are briefly explained in the next section.

1.7 Problem Definition

The problems as tackled in this thesis can be summarized as follows:

1. In TCPAs, the required number of PEs can be instantiated as per the needs of the application by selecting appropriate number of rows and columns of the array at synthesis time. Here, an important problem is: How can we generate code or program for each of the PE in the array in a scalable manner and irrespective of the array size? Generating code for each PE in the array could be tedious and a time-consuming task. Even assuming the array size of $10 \times 10$, there are 100 PEs for which one should generate the code.

2. All PEs in a TCPA are only having an instruction memory of small size in the order of only few kilobytes to reduce the area and power consumption. Therefore, the generated programs should be compact and should be also independent of the problem size. Since changing the problem size involves only modifying the loop bounds in the given nested loop program, this should not influence the size of the generated code. Moreover, a loop program is already scheduled considering the resource constraints in the front end of the compiler framework. In the back end, the code generation approach should preserve a pre-computed loop schedule of computations, i.e., it should not introduce any delay or modify the schedule by inserting some extra instructions. Here, an unsolved problem is: How can we generate a code/program that is compact and independent of the problem size while preserving a given schedule of a given loop program?

3. For a given TCPA architecture, the interconnection structure is essential and responsible for the propagation of data and control signals within the processor array. In this context, a credible question is: How can we generate an interconnect configuration for all the PEs in the array in a scalable manner and automatically from a given scheduled loop program?
4. In order to execute the generated programs correctly, a Global Controller (GC) is needed which captures the entire static control flow of the loop program. It must generate control signals at the right times while the generated code is executing on a PE. Control signals emanating from this global controller must be forwarded to all the PEs in a linearly staggered fashion with appropriate delays as determined by a given schedule. Furthermore, when a TCPA is integrated as a programmable accelerator in a more traditional MPSoC design—catering to the needs of multiple applications—GC has to be programmable depending on the application. In this context, an important problem is: How can we design such a programmable hardware controller and generate its configuration data automatically from a given input loop program so that it generates the control signals at the right times to steer the correct program execution of all PEs?

5. In a TCPA, processor elements are tightly coupled and they do not have direct access to a global memory. Instead, small reconfigurable buffers are provided surrounding the processor array to hide the communication latency as well as to improve data locality. Therefore, apart from program code and interconnect structure configuration, also a schedule of buffer read/write address sequence must be determined. In this context, Address Generators (AGs) are proposed as a solution which can be configured depending on the application. Here, an important problem is: How can we design such programmable address generators that generate address sequences as well as enable signals for read and write access to reconfigurable buffers at the array borders, which depend on the input loop program. In other words, can we efficiently also generate the configuration data for such programmable address generators?

1.8 Contributions of the Thesis

Solutions to the above mentioned problems and the major steps of our mapping flow are shown in Figure 1.8. The main contributions of the thesis are (a) automatic code and interconnect generation approach for nested loop programs mapped onto TCPAs, (b) concepts and design of a programmable Global Controller (GC) as well as Address Generator (AG) with proper compiler support to generate the configuration data. The following is a brief summary of these novel contributions.

1. In this dissertation, we propose an automatic code generation approach that will address all the “How can we?” questions raised above. Once a given nested loop is mapped onto a TCPA, it divides the entire array into so-called processor classes, where all processors that belong to the
1. **Introduction**

same class obtain the same program (assembly code). Instead for each individual PE, we propose to generate code only for one candidate element of each processor class and load it to all the processors belonging to the same processor class [BHT14*, HLB+14*]. In this way, we end up only in generating code for a problem-size independent number of processor classes.

2. The iteration space of a given loop nest that is partitioned to execute on a processor class candidate element is further subdivided into mutually exclusive sub iteration spaces, also called as program blocks. The execution of an iteration space can then be represented as a control flow graph of these program blocks [TBHL15*] from which compact codes are generated [BHT13*, BHT14*] as follows: During code generation, repetitive patterns of instructions are identified and looping instructions are introduced to avoid code explosion and to compact the code. Our code compaction approach introduces zero-overhead looping—with the help of a GC—these looping instructions are executed multiple times in accordance with the given schedule. Moreover, changing the problem size involves only changing the configuration of the GC—making the generated assembly code size independent of the problem size.

3. We also propose a scalable interconnect generation approach by dividing the entire processor array into so-called interconnect classes, where all processors in a class will have the same type of interconnections. The interconnect architecture for data signals consists of dedicated links with delay shift registers [BHT14*], whose lengths are also determined automatically from the given schedule.

4. We propose the design of a programmable GC [BHT14*, TBHL15*] with required interfaces to integrate it into an MPSoC. We present the functionality of the GC including necessary algorithms, systematically convert the algorithm into a circuit/hardware, and generate its configuration data within the compiler framework. In addition, the required delay register lengths for propagating control signals to all PEs through PEs are also computed from the given schedule.

5. We propose the design of programmable address generators and discuss how the configuration data for these address generators may be extracted automatically from a given scheduled and partitioned loop program. Moreover, due to linear wave-form like scheduling, address and enable signals of an input/output variables generated in an AG can be reused by propagating to each buffer assigned to the same variable with appropriate delay
1.9 Thesis Organization

In Chapter 2, we present the concept and structure of TCPAs from the micro architectural point of view including the PE architecture, Instruction Set Architecture (ISA), branch unit and the interconnect wrapper. We also present a tool called TCPA editor by which all the parameters for generating a required TCPA including number of PEs, Functional Units (FUs), etc. We also discuss how a TCPA can be integrated into a MPSoC design and different hardware components such as GC and AGs. In Chapter 3, we introduce our compiler framework for mapping nested loop programs onto TCPAs and present the front end of it.

In Chapter 4, all the steps in the code generation approach are discussed in detail focusing on: a) the register allocation in Section 4.2, b) a brief overview of the code generation approach in Section 4.3, c) the reasons for the existence of processor classes as well as program blocks, and extracting processor classes as well as program blocks automatically from a given input loop specification in Section 4.4, d) generating control flow graph of program blocks in Section 4.5, and e) finally, compact code generation using this control flow graph for a processor class in Section 4.6. We also present the systematic approach of generating the interconnect structure, and dividing the processor array into different interconnect classes in Section 4.8.

In Chapter 5, we focus on the concept, design and implementation details of a programmable Global Controller (GC). At first, we explain the procedure of deducing control signals automatically from a program block control flow graph. In Section 5.1, functionality as well as requirements of the GC are specified. In Sections 5.2 and 5.3, the design of the GC is explained using different sub blocks and their functionality. Generating the configuration data for the GC is explained in Section 5.5, and followed by implementation results in Section 5.8.

Chapter 6 deals with the design and implementation details of programmable AGs. In Section 6.1, importance of the Address Equation (AE) in designing the AG hardware and different types of AGs are discussed briefly. An overview of the address generation in the context of TCPA is given with a motivating example in Section 6.2. Different address generator architectures found in the literature are discussed briefly with few hardware details in Section 6.3. Few selected architectures are implemented and compared in terms of area, performance, and power dissipation in Section 6.4. We also discuss how the configuration data for the AGs can be generated automatically within the compiler framework in Section 6.5.

Finally, conclusions and future work are presented in Chapter 7.
2

Tightly Coupled Processor Arrays

In the previous chapter, we emphasized the importance of programmable accelerator-based heterogeneous systems as a solution to meet the insatiable high performance needs, higher energy efficiency while providing the flexibility in next generation embedded systems. In this chapter, we give an overview of the class of Tightly Coupled Processor Array (TCPA) architectures, fundamentals, background and related work. The main contributions of this chapter can be summarized as follows:

- Fundamentals and background of the highly parameterizable class of TCPA architectures.

- Interfacing and integration of a TCPA into a Multi-Processor System-on-Chip (MPSoC) design as a programmable accelerator and the need for additional hardware components. More specifically, we highlight the importance of a Global Controller (GC) and Address Generators (AGs)—on which this dissertation focuses on.

- Related work summarizing the state of the art coarse-grained architectures including Massively Parallel Processor Arrays (MPPAs).

2.1 Tightly Coupled Processor Array (TCPA)

This section presents the generic architecture of Tightly Coupled Processor Arrays (TCPAs) from the micro-architectural point of view. As aforementioned, TCPAs are built to accelerate computationally intensive loop programs by exploiting both Instruction-Level Parallelism (ILP) as well as Loop-Level Parallelism (LLP) while achieving a better energy efficiency compared with general purpose Commercial Off-The-Shelf (COTS) embedded processors [KSHT09]. A TCPA is a highly parameterizable architecture template, and thus offers a high degree of flexibility. Some of its parameters have to be defined at synthesis time, whereas other parameters can be reconfigured at runtime. The heart of TCPA comprises of tightly coupled Processing Elements (PEs) whose number is
2. Tightly Coupled Processor Arrays

defined at synthesis time by specifying the number of rows and columns. Each PE is encapsulated in an interconnect wrapper by which PEs are connected in a regular two-dimensional grid topology to form a particular instance of a TCPA. The building blocks of a TCPA are shown in Figure 2.1, and are briefly described in the following subsections.

2.1.1 Processing Element (PE)

A TCPA consists of an array of Very Long Instruction Word (VLIW) Processing Elements (PEs) arranged in a 1-D or 2-D grid with local interconnections as proposed in [KHKT06a], see also Figure 2.1. Each PE has a VLIW structure as shown in Figure 2.2. Many parameters of the PE architecture are configurable at synthesis time. For example, different types and number of functional units, (e.g., adders, multipliers, logical units, shifters, and data movers), different types and number of registers and number of words in instruction memory. Each PE is weakly programmable [KHKT06a] in the sense that each PE has only a limited amount of instruction memory and the control overhead is kept as small as possible for the given application domain of algorithms. For example, there is typically no support for interrupts and exceptions. The PEs are able to operate on two types of signals, i.e., data signals (shown in blue color in Figure 2.1)
2.1 Tightly Coupled Processor Array (TCPA)

whose width can be defined at synthesis time and control signals (shown in red color in Figure 2.1), which are normally one-bit signals used to control the flow of execution in the PEs. Therefore, also two types of registers do co-exist inside a PE: data registers and control registers\(^2\). These registers consist of four kinds of storage elements: general purpose registers \(RDX_i\), input registers \(IDX_i\), output registers \(ODX_i\), and feedback registers \(FDX_i\). \(IDX_i, ODX_i\) are ports that are used mainly to communicate with neighboring PEs for data input/output, respectively, and \(ICX_i, OCX_i\) are for control input/output. Input registers are implemented as shift registers of length \(l\), whose input delay can be configured at runtime from 1 to \(l\). Feedback data registers \(FDX_i\) are special so-called rotating registers that are provided to support loop-carried data dependencies and cyclic data reuse. Figure 2.3 shows the semantics of the read and write for these feedback data registers as well as the organization of different registers in a register file. Each functional unit can read and write each general-purpose register and multiple yet disjoint writes are supported in parallel. This scheme is easy to implement and very flexible, but it can result in a complex and expensive interconnect between the registers and the functional units [Kis12].

All Functional Units (FU) in a PE support single cycle operations, i.e., within a single cycle an instruction is read, decoded, appropriate operands are read, the instruction is executed, and finally results are written back to specified internal or output registers. Furthermore, an appropriate branch may also be taken, i.e., the Program Counter (PC) is updated with the right value at the end of the same cycle as explained in Section 2.1.1.

Functional Units and Instruction Set Architecture (ISA)

There are several types of functional units available for customization of PEs, each having its own instruction set. The functional units in a PE may be chosen according to domain-specific computational needs and thus can be optimized accordingly. Since our architecture primarily addresses compute-intensive algorithms as typically specified by nested-loop programs, the instruction set is tuned to the distributed parallel execution of loop nests. The different flags of a functional unit are set according to the result of an operation. For instance, the zero flag is set when the result is zero, the negative flag is set when the result is negative, the overflow flag is set when an overflow occurs, and the carry flag is set, when the carry bit is set. In the PE instruction set architecture, also the widths of the different instruction fields, like operation code, and regis-

\(^2\)For the sake of better visibility, control registers and control I/O ports are not shown in Figure 2.2. However, it should be noted that similar to data-path specific resources, such as registers \(RDx_i, IDx_i\), and so on, there also exist equivalent control path resources, e.g., registers \(RCx_i, ICx_i\), etc.
Figure 2.2: Processing Element (PE) internal architecture showing multiple functional units, a VLIW instruction memory, an instruction decoder, and a set of registers for local data processing (RD\(_x\)), input registers (ID\(_x\)), and output registers (OD\(_x\)).

ter addresses are parameterizable at synthesis time to suit different application needs.

In general, five instruction types with corresponding semantics are distinguished as explained below:

- **Register instructions**: Both source operands, denoted by ‘op1’ and ‘op2’, are specified by registers and the result of the operation is written to a destination register, denoted by ‘dest’, see e.g., Table 2.1.

- **Immediate instructions**: Similar semantics as register instructions except for the second operand is being coded in the instruction itself.

- **Constant instructions**: Used to load a specific constant into a destination register.

- **Move instructions**: Used for transferring data among different storage elements.
Figure 2.3: Register file organization showing general purpose, output, input FIFO, and feedback-FIFO registers. It should be noted that a read from a feedback-FIFO results in writing the same data to the FIFO whereas a write will push new data into it.

- **Branch instructions**: Used for conditional and unconditional branching.

Table 2.1 shows few example instructions. For further details, we refer to [KHKT06b, Kis12].

**Branch Unit**

In case of VLIW architectures, very often multi-way branch units are used [MC95]. The branch unit decides which instruction is to be executed next, i.e., determines the next PC value. The different branch target addresses are part of the branch instruction itself. In order to choose a target address, several so-called *branch flags* are evaluated where the number of flags\(^3\), denoted by \(n\), is parameterizable. As a branch flag, any control signal from the control register file, input control signals, or any status flag from any functional unit can be chosen. The mapping of the aforementioned signals to the branch flags can be

\(^3\)In practice, \(n\) is usually 2 or 3.
Table 2.1: Example instructions and supported addressing modes.

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder/Subtractor</td>
<td>ADD dest op1 op2</td>
<td>dest ← op1 + op2</td>
</tr>
<tr>
<td></td>
<td>ADDI dest op1 (\text{im}^1)</td>
<td>dest ← op1 + (\text{im})</td>
</tr>
<tr>
<td></td>
<td>ADDC dest (\text{const}^2)</td>
<td>dest ← dest + (\text{const})</td>
</tr>
<tr>
<td></td>
<td>SUB dest op1 op2</td>
<td>dest ← op1 - op2</td>
</tr>
<tr>
<td></td>
<td>SUBI dest op1 (\text{im})</td>
<td>dest ← op1 - (\text{im})</td>
</tr>
<tr>
<td></td>
<td>SUBC dest (\text{const})</td>
<td>dest ← dest - (\text{const})</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
<td>no operation</td>
</tr>
<tr>
<td>Multiplier</td>
<td>MUL dest op1 op2</td>
<td>dest ← op1 * op2</td>
</tr>
<tr>
<td></td>
<td>MULI dest op1 (\text{im})</td>
<td>dest ← op1 * (\text{im})</td>
</tr>
<tr>
<td></td>
<td>MULC dest (\text{const})</td>
<td>dest ← dest * (\text{const})</td>
</tr>
<tr>
<td></td>
<td>NOP</td>
<td>no operation</td>
</tr>
<tr>
<td>Logic Unit</td>
<td>ANDI dest op1 (\text{im})</td>
<td>dest ← op1 (\land) (\text{im})</td>
</tr>
<tr>
<td>Shifter</td>
<td>SHLI dest op1 (\text{im})</td>
<td>dest ← op1 (&lt;\text{im})</td>
</tr>
<tr>
<td>Data Path Unit (DPU)</td>
<td>CONST dest (\text{const})</td>
<td>dest ← (\text{const})</td>
</tr>
</tbody>
</table>

\(^1\) \(\text{im}\) denotes an immediate operand. Immediate instructions are binary operations.
\(^2\) \(\text{const}\) denotes a constant value. Constant instructions are unary operations.

Table 2.2: Branch instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRANCH((f_{n-1}\ldots f_1, f_0))</td>
<td>(pc ← \text{target}(\text{addr}_{2^n-1}, \ldots, \text{addr}_1, \text{addr}_0))</td>
</tr>
<tr>
<td>NEXT</td>
<td>(pc ← pc + 1)</td>
</tr>
</tbody>
</table>

selected dynamically via multiplexers, with the only restriction that for every functional unit only one flag of the available four flags can be selected to be a branch flag [KHKT06a]. With the help of this concept, \(n\) branch flags may be analyzed in parallel and, thus, a total of up to \(2^n\) different branch targets can be addressed in one branch instruction in each cycle. The whole branch process, including the selection of the branch flags, their evaluation, selection of the target jump address and writing it to the program counter, is executed in a single cycle. Table 2.2 shows the branch instructions. For further details, we refer to [KHKT06a].

2.1.2 Reconfigurable Inter-Processor Network

In order to provide support for many different interconnection topologies, a structure of multiplexers inside an interconnect wrapper module [KHKT06a]
2.1 Tightly Coupled Processor Array (TCPA)

![Diagram of a tightly coupled processor array](image)

(a) Internal structure of a interconnect wrapper module  
(b) Exemplary adjacency matrix

**Figure 2.4:** A $2 \times 2$ PE array interconnected with interconnect wrapper modules in a regular fashion where an adjacency matrix defines the connection structure between a PE and an interconnect wrapper.

Around each PE is provided, which allows to reconfigure inter-PE connections flexibly, see Figure 2.4(a). These interconnect wrappers themselves are connected in a mesh topology. Thanks to such a circuit-switched interconnect, a fast and reconfigurable communication infrastructure is established among PEs, allowing data produced in a PE to be used by a neighboring PE in the next cycle. To configure any particular interconnect topology, an adjacency matrix is specified for each interconnect wrapper in the array at synthesis time. Each adjacency matrix defines how the input ports of its corresponding wrapper and the output ports of the encapsulated PE are connected to the wrapper output ports and the PE input ports, see Figure 2.4(b). Here, two input signals in each (North, East, South, and West) direction of the interconnect wrapper module are assumed, and two input and output ports for the encapsulated PE are assumed. If an arbitrary input $i$ and output signal $j$ have to be connected, the variable $C_{ij}$ is set to 1 else to 0. If multiple source ports are allowed to drive a single destination port, then a multiplexer with an appropriate number of input signals is generated [KHKT06a]. The select signals for such generated multiplexers are stored in configuration registers and can be changed even dynamically, i.e., different interconnect topologies—also irregular ones—can be established and changed at runtime. To summarize, by means of the first static level of interconnect signals with regular layout, the problems of placement and routing of PEs in an array are considerably reduced [Har01]. With the help of
the second level of interconnect, topologies different from a mesh can be specified and even changed dynamically.

Figure 2.5 shows some well known network topology configurations such as 4D hypercube, 2-D torus and connections of a systolic array implemented on a 4×4 processor array. In each case, the logical connection topology is depicted on the top and the corresponding array interconnect configuration is shown on the bottom of Figure 2.5(a)–(c). Here, each interconnect wrapper has three connection channels to each neighboring PE. Depending on the desired topology, some of them are activated (shown in different colors) and some are inactive (shown in light gray color). It is worth to mention that for some connections, wrappers may also be used to bypass a neighbor PE (connections with a length of more than one PE distance). This capability is highly useful, e.g., to implement a 2-D torus. As mentioned above, data may be transmitted on each port connecting two PEs within a single cycle. However, bypass capabilities are usually restricted to pass a maximum of 2 to 4 hops. Figure 2.5(c) shows how diagonal connections may be realized on TCPA. This capability is used especially when optimally mapping (embedding) nested loop programs with uniform data dependencies onto TCPA which is the main focus of our work.

**Figure 2.5:** Different network topologies, such as, (a) 4D hypercube, (b) 2-D torus, and (c) a systolic array, are implemented on a 4×4 processor array. In each case, the top figure depicts the logical connections between different nodes in the network, and the bottom figure illustrates the physical connections on the processor array established by configuration of interconnect wrappers [TBHL15*].


2.1.3 Configuration Management

To configure VLIW programs and the interconnect structure of each PE according to the currently processed algorithm, a configuration loader present in each PE is used, see Figure 2.6(a). This loader is controlled by a finite state machine called global configuration controller. There is also a global configuration memory for storage of different programs and interconnect schemes on the array level. The global configuration controller is connected to a configuration manager which reacts on external configuration requests, locates the corresponding program in the configuration memory with the help of a small address look-up table, and configures the configuration controller. The external reconfiguration requests are supposed to be issued, e.g., from an embedded processor core in an System-on-chip (SoC) design.

In a (re)configuration phase, the global configuration controller reads the current configuration from the global configuration memory and puts the data on a global configuration bus. It should be noted that the width of the global configuration memory and the configuration data bus are parameterizable at synthesis time whose value might differ from either VLIW instruction width of a PE or the interconnect configuration registers width present in the interconnect wrapper module. The local configuration controllers present in each PE

![Diagram](image)

(a) Interaction among the different architectural components involved in TCPA configuration.

![Diagram](image)

(b) Multicast programming scheme showing vertical and horizontal mask registers and interconnect wrapper connections. Different colors in the background of the PEs denote their selection using distinct multicast signatures as shown.

Figure 2.6: Configuration hardware and multicast programming scheme in TCPA [KHKT06a, Kis12]
Tightly Coupled Processor Arrays

must run synchronously with the global configuration controller and write the received configuration to either instruction memory or interconnect configuration registers [KHKT06a].

To achieve scalability of the configuration scheme in TCP A, a multicast scheme has been provided [TSA05]. As a global bus is used for (re)programming of PEs, it is important to be able also to program and reprogram parts of a large processor array, like for example an array of size 10\times 10 with 100 processing elements or more. With the help of multicast bits stored in special registers, see Figure 2.6(b), it is possible to program single processing elements as well as groups of columns and rows of processing elements in an array concurrently. There is a corresponding multicast bit for each row and each column in a two-dimensional processor array. Each interconnect wrapper in the processor array is connected to one horizontal and one vertical multicast bit line, see Figure 2.6(b). If a multicast bit is set, the corresponding row or column in the array is selected. Exactly those PEs will be programmed simultaneously which have both the vertical and horizontal multicast bit lines set. The single multicast bits are grouped into two multicast registers: A vertical mask register for the columns of the array and a horizontal mask register for the rows of the array. Both mask registers are located in the global configuration controller and are accessible from the configuration manager. A tuple containing the two mask register values, also called multicast signature, uniquely identifies a group of PEs by which partial reprogramming of the array also becomes possible [Kis12].

Configuration data in the global configuration memory contains different configuration types, i.e., VLIW programs and interconnect schemes. A combined configuration type also exist where a VLIW program and interconnect scheme are grouped to build one configuration type. Since the configuration memory also consumes logic resources, it has to be as small as possible. Therefore, in a TCP A, a possibility is given to address the VLIW program configuration and the interconnect scheme separately. A higher flexibility may be achieved if the VLIW instruction memory and configuration registers in the interconnect wrapper are configured separately. If for a new application, for instance, only the current interconnect configuration has to be changed but the programs remain the same, then only a small interconnect configuration data has to be changed, resulting in less configuration time.

2.2 TCPA Editor

A graphical tool called TCPA Editor has been designed for TCPA architecture and software development, see [Kup08]. A screen-shot is shown in Figure 2.7. A TCPA architecture can either be created/edited graphically, or loaded from a MAML (machine mark-up language) textual description using
2.3 TCPA: Programmable Accelerator Perspective

TCPAs can be integrated into heterogeneous MPSoC designs to use it as a programmable hardware accelerator. However, it requires few other additional hardware components (details to follow next) to interact with the rest of the

Figure 2.7: Different graphical framework views of TCPA Editor, for further details refer to [Kup08].

this tool [KHKT08]. Different software and hardware configuration panels as well as distinct interconnect views are provided. This tool also generates all the required files for a cycle-accurate simulation of the array. Furthermore, simulation results can be loaded back into the editor where cycle by cycle execution of the programs and register values in each PE can be observed graphically. This particular feature helps to debug the programs or software executing on the array. TCPA editor also dumps out all the required parameters of the array in VHDL syntax for the generation of processor array description in the Register-Transfer Level (RTL) form. This RTL can be synthesized for either Application-specific Integrated Circuit (ASIC) or Field-Programmable Gate Array (FPGA) technology.
components in an MPSoC. In this dissertation, we introduce the term TCPA tile [HHB+12] where the TCPA and all the required additional components are integrated as a tile in a heterogeneous tiled architecture as shown in Figure 2.8. Few additional components, such as Reconfigurable Buffers (RBs), AGs, GC, and a Configuration and Communication Processor (CCP) are required. These additional components and the TCPA are interconnected within a tile using a bus-based protocol, for instance, an Advanced High-performance Bus (AHB). The purpose and functionality of each these components are explained briefly in the following. However, GC and AGs are designed as part of this dissertation; thus, the subsequent chapters give an elaborated information on these two components.

Global Controller (GC)

During the execution of loop programs, numerous control flow decisions such as incrementation of iteration variables, loop bound checking, and other static control flow operations may cause in general a huge computational overhead within each PE. In case of programmable accelerators such as TCPAs, the program control is realized using a programmable instruction memory, program counter, and instruction decoder. Furthermore, thanks to the regular mapping and scheduling of iterations among PEs, much branching information is just executed in a delayed fashion and can therefore be moved as much as possible away from each PE into a common control flow. This global control flow, can therefore be outsourced for being computed by one Global Controller (GC) per loop program. The GC generates branch control signals, which are then propagated in a delayed fashion over the control network to the PEs where it is combined with the local control flow (program execution). This orchestration and distribution of work enables also the execution of nested loop programs with zero-overhead loop, not only for innermost loops but also for all static conditions in arbitrary multidimensional data flow [HLB+14]. For further details, we refer to Section 4.7, which address the assembly code generation and explain how these control signals are combined with program execution. For hardware details as well as configuration generation, we refer to Chapter 5 on global controller.

Reconfigurable Buffers and Address Generators

Data locality is a key factor for the acceleration of loop programs on processor arrays. As the processor elements are tightly coupled, they do not have direct access to a global memory. Data transfers to and from the array are performed through the border PEs, which are connected to a set of surrounding reconfigurable buffers. These buffers can be configured, to either work as simple FIFOs
Figure 2.8: Architectural block diagram of a TCPA tile showing: 2-D processor array, mesh-like interconnection scheme among Processing Elements (PEs), reconfigurable buffers surrounding the processor array, address generators, global controller, communication & configuration processor, interrupt controller, and Network Adapter (NA) interface. Figure also shows the integration of the TCPA tile as part of a heterogeneous tiled architecture [HBB+12].
or as RAM-based addressable memory banks [HSL+13*]. For the latter, appropriate AGs are configured to generate the correct sequence of read/write accesses according to a given loop schedule. For further details, we refer to Chapter 6 on addressers generators.

**Configuration and Communication Processor**

This processor is one of the essential components in the TCPA tile that configures/reconfigures all the components in the array. Furthermore, it communicates with the rest of the SoC through a Network Adapter (NA) interface. For instance, it responds to interrupts raised within the array such as buffer empty and buffer full by initiating appropriate DMA transfers through the NA to fill and flush the reconfigurable buffers, respectively. Thus, this component is named *configuration and communication processor* and a LEON3 RISC processor is proposed and used here for this purpose. Furthermore, this processor can keep some kind of bookkeeping data such as the number of processors already in use and average temperature of the array to propagate it to the upper layers in resource-aware programming paradigm such as invasive computing [Tei08].

**2.4 Related Work**

In this section, a summary of state of the art coarse grain reconfigurable architectures as well as massively parallel processor arrays are discussed with focus on their architecture details such as processor/core architecture, inter-processor communication methods, memory hierarchy and control mechanisms.

**Silicon Hive’s Streaming Array** [BJLV06]: The basic component of the architecture of Silicon Hive’s accelerator is the Processing and Storage Element (PSE), see Figure 2.9. A PSE is a VLIW-like data-path consisting of several interconnect networks (IN), one or more operation-issue slots (IS) with associated function units (FU), distributed register files (RF), and optionally local memory storage (MEM). PSEs are designed according to a template that ensures all PSEs are easy and clean data-paths for a compiler to handle, guaranteeing high-level of programmability by correct construction. A matrix of one or more PSEs, together with a VLIW-like controller (CTRL) and configuration memory (CONFIG MEM), make up a cell. A cell is a fully-operational processor capable of computing complete algorithms. PSEs within a cell can communicate with each other via data communication lines (CL). Typically, an application function at a time is mapped onto the array of PSEs. However, if PSEs are abundant, then more application functions can be mapped in space, in a data-flow manner. An array of one or more cells, connected together via a data-driven communication mechanism, forms a streaming array, see e.g., Figure 2.9. The
communication across cells takes place through blocking FIFOs accessed from load/store (LD/ST) units within the cells. Multiple functions can be concurrently mapped onto the streaming array, each one occupying a non-overlapping sub set of the cells. The power efficiency lies around 70 MOPS/mW in 130 nm technology, see [Hal03, Hiv08], which is a typical value for coarse-grained architectures.

ADRES [MVV+03]: The ADRES architecture has a matrix structure with two basic functional views: a VLIW processor view and a reconfigurable matrix view. There are three types of basic components which can be parameterized: functional units, storage elements such as register files and memory blocks, and routing resources that include wires, multiplexers and buses. In fact, ADRES is a template more than a concrete processor architecture. It tightly couples these two views, where both share functional units and register file because their executions will never overlap with each other thanks to the processor/co-processor model, see Figure 2.10. Basically, this architecture may speed up an application by executing the unaccelerated part on the VLIW processor and pipelined loops on the reconfigurable matrix. For the VLIW processor, several FUs are allocated and connected together through one multi-port register file, which is typical for VLIW architectures. Compared with the counterparts of the reconfigurable matrix, these FUs are more powerful in terms of functionality and speed. They can execute more operations such as branch operations. Some of these FUs are connected to the memory hierarchy, depending on available ports. Thus, the data access to the memory is done through the load/store operation available on those FUs. Typical power efficiency values are around
2. **Tightly Coupled Processor Arrays**

![Architecture of the ADRES coarse-grain reconfigurable array](image)

**Figure 2.10:** Architecture of the ADRES coarse-grain reconfigurable array, source [BKMS09].

50 MOPS/mW, see [BKMS09].

**STHORM:** The STHORM [MBF+12, BFFM12, MCU14] platform is a high performance accelerator from STMicroelectronics. It is organized around multiple clusters, connected via a high performance NoC. Each cluster is implemented with independent and power and clock domains. Each cluster can contain up to 16 tightly coupled processors sharing a L1 memory. Each cluster has its own Core Controller (CC) to control the cluster itself, to communicate with other clusters and host, and to perform DMA transfers between the different memories. All clusters are controlled by a fabric controller and all clusters share a L2 memory. The whole cluster fabric is connected to a host using a system bus.
2.4 Related Work

This platform comes with different simulators to develop and test parallel programs with matured and efficient SystemC/TLM and co-simulation approaches.

**Intel SCC [HDH+10]:** The Intel Single chip Cloud Computer (SCC) is originated from Intel tera-scale research project. It deploys 48 Intel Pentium P54C-x87FP processor cores and incorporates technologies intended to scale multi-core processors to 100 cores and beyond, such as an on-chip network, advanced power management technologies and support for message-passing. The SCC architecture can be seen in Figure 2.11. It consists of a $6 \times 4$ two-dimensional network on chip which connects 24 Pentium P54C-x87FP dual-core tiles. Each tile contains two processor cores together with 256 KB level two cache per core and a packet router. 16 KB level one caches are available on the core itself. The chip has 8 voltage and 28 frequency islands for cores and network-on-chip which can be set independently. In a sense, the SCC is a microcosm of cloud data center. Each core can run a separate OS and software stack and act like an individual compute node that communicates with other compute nodes over a packet-based network. Fine-grained power management is a focus of the chip as well. Software applications are given control to turn cores on and off or to change their performance levels, continuously adapting to use the minimum energy needed at a given moment. The SCC can run all 48 cores at one time over a range of 25 W to 125 W and selectively vary the voltage and frequency of the mesh network as well as sets of cores. Each tile (2 cores) can have its own frequency, and groupings of four tiles (8 cores) can each run at their own voltage.

**xSTream [MAB+11]:** The STMicroelectronics xSTream architecture is based on the convergence of communication and computing as a way to solve scalability and programmability of high-performance embedded functionalities, such as
graphics, multimedia and radio subsystems. Figure 2.12 shows a high-level view of a complete system embedding an instance of the xSTream processor fabric. The system is composed of the traditional host processing part on the left side, while the entity on the top-right end of the Figure 2.12 is the streaming engine of the xSTream architecture. It is intended to address the needs of data-flow dominated, highly computational intensive semi-regular tasks, typical of many embedded products. The streaming nature of the kernels mapped onto it makes it possible to design a semi-regular fabric of programmable engines interconnected via a relatively simple network of point to point channels. The fabric support many simultaneous software pipelines running on the processing elements as to accommodate complex applications and also provide load balancing and latency hiding capability. A property of the streaming fabric is to support very high internal data bandwidth, throughput and computationally intensive tasks. The processing elements of the streaming fabric (xPEs) are relatively simple programmable processors or engines with a general purpose but simple basic ISA that can be extended with SIMD or vector mode instructions. In addition, the xPE is design time parametric and can be instantiated in a number of variants each supporting a different degree of features and capabilities. They execute instructions fetched from local memories instead of caches, a great
simplification at the pipeline forefront.

**TILE64:**
The Tile architecture [BEA+08] has its origins in the RAW research processor developed at MIT and later commercialized by Tilera, a startup company founded by the same MIT research group. TILE64 SoC consists of an $8 \times 8$ tiled array with router nodes and local cache mechanisms in a two dimensional mesh as shown in Figure 2.13. The key differentiating technology of the tile architecture is the on-chip interconnect and the cache architecture. Each core (also called tile) has a short three issue in-order pipeline with a VLIW instruction set, 8 KB L1 cache, 64 KB L2 cache and an interconnect switch that can connect the neighboring tiles. This way, the interconnect is essentially a switched network with very short wires connecting neighboring tiles linked through the tile-local switch. The router nodes provide the communication functionality to exchange data among the cores in the network. Tilera claims that a fully coherent caching mechanism based on a local L1 cache and a distributed L2 cache is providing higher efficiency and concurrent access from multiple cores. Compared to a large centralize L2 or L3 cache access bottlenecks are avoided and power efficiency is improved. The chip was fabricated in 90 nm CMOS technology with multi-threshold devices. The operating frequencies of the chip range from 500–866 MHz and its power consumption ranges from 15 to 22 W. Its main target applications are advanced networking, digital video, and telecom.

**picoArray:** The PicoArray is a highly regular structure with a large number of replicated basic building blocks [DPT03]. The picoArray is a multi-core processor system for signal processing applications developed by PicoChip. The two building blocks of the picoArray are an ARM926EJ-S microprocessor and the array itself. The picoArray consists of 248 VLIW DSP processors connected together in a 2-D array as shown in Figure 2.14.
All processors consist of a basic building block with a 16-bit wide Harvard architecture supporting three-way VLIW. Each processor has dedicated instruction and data memory as well as access to on-chip and external memory. Further, the core can be enhanced with local memories, functional accelerator units, and hardwired execution units for specific applications. The ARM926EJ-S used for control functions is a 32-bit RISC processor. Application specific functional accelerators for correlation and path metric calculations, for turbo code (CTC), FFT, Viterbi, Reed Solomon, and cryptographic calculations are available and can be integrated in the picoArray.

The picoArray processor nodes are interconnected by a two-dimensional grid of 32-bit buses and programmable bus switches. Additional I/O and communications facilities are connected to the grid and are used to interface to external input/output devices or an external host processor system. Some of the picoArray applications are in high-speed wireless data communication standards for metropolitan area networks (WiMAX) and cellular networks [high-speed downlink packet access(HSDPA) and wideband code division multiple access (WCDMA)], as well as in the implementation of advanced wireless protocols.

Montium [HSM03]: The Montium tile processor is a programmable architecture for fixed-point digital signal processing algorithms developed by Recore Systems. It is tailored for a specific algorithm domain and algorithms such as FIR filters, FFT, correlation, turbo decoding, DCT, matrix multiplication and linear interpolation are used to refine the data path. The processor data path includes many local memories and functional units interconnected by a reconfigurable communication network. The small local memories are also motivated by the locality of reference principle to obtain energy efficiency. The data-path width and memory capacity is customizable at design time. The datapath has a width of 16-bits and the ALUs support both signed integer and signed fixed-point arithmetic operations with local registers to store intermediate data. Each local memory is 16-bit wide and has a depth of 512 positions, which adds up to a storage capacity of 8 Kbit per local memory. A memory has only a single address port that is used for either reading or writing. A reconfigurable Ad-
dress Generation Unit (AGU) accompanies each memory. A central program sequencer controls the functional units and interconnection network with a small configuration binary of about 1 KB which can be updated in less than 5 µs. The communication and control unit provides the interfaces to external devices.

AsAP \[\text{YMA}^+08\]: The Asynchronous Array of simple Processors (AsAP) architecture comprises a 2-D array of reduced complexity programmable processors with small memories (128 word program and 128 word data) interconnected by a reconfigurable mesh network. AsAP was developed by researchers from the University of California, Davis and achieves high performance and high energy efficiency, while requiring a relatively small circuit area. This architecture relies on the data flow nature of DSP algorithms to obtain power and performance efficiency. AsAP processors are well suited for implementation in future fabrication technologies, and are clocked in a Globally Asynchronous Locally Synchronous (GALS) fashion. The processors communicate asynchronously by doubly clocked FIFO buffers, and each core has its own clock generator so that the device is essentially clockless. When a FIFO is either empty or full, the associated cores will go into a low power state until they have more data to process. These and other power saving techniques are used in a design that is heavily focused on low power computation. Individual oscillators fully halt in 9 cycles when there is no work to do, and restart at full speed in less than one cycle after work is available. The chip requires no crystal oscillators, PLLs, DLLs, or any global frequency or phase-related signals whatsoever. The multi-processor architecture efficiently makes use of task-level parallelism in many complex DSP applications, and also efficiently computes many large tasks utilizing fine-grain parallelism. A test chip was manufactured that contains 167 programmable processors with dynamic supply voltage and dynamic clock frequency circuits, three algorithm-specific processors, and three 16 KB shared memories, all clocked by independent oscillators and connected by configurable long-distance-capable links \[\text{TCM}^+09\].

MPPA-256 \[\text{IDAB}+13\]: The Kalray MPPA-256 is a single-chip many core processor manufactured in 28 nm CMOS technology that targets applications, where low energy per operation and time predictability are the primary requirements. It integrates 256 user cores and 32 system cores which are distributed across 16 compute clusters of 16+1 cores, and 4 quad-core I/O subsystems. Each compute cluster and I/O subsystem owns a private address space, while communication and synchronization between them is ensured by data and control Network on Chip (NoC). The MPPA-256 processor is also fitted with a variety of I/O controllers, in particular DDR, PCI, Ethernet, and GPIO. It also comes with a software development kit that provides standard GNU C/C++ and GDB development tools for compilation and debugging.
2. **Tightly Coupled Processor Arrays**

To summarize, all of the architectures discussed above are having processors or cores in a very regular structure in the form of array and interconnected mostly by scalable topologies such as NoC, interconnect switches, and dedicated mesh-like point-to-point connections. It is also very common practice that all the processors in the array are either VLIW processors or some other simple cores to execute computationally intensive tasks whereas control intensive tasks are in general performed by a control processor different from a array processor. Different memory cache hierarchies as well local scratch pad memories are implemented to improve the data locality and hide the data communication latencies. In fact, all of these mechanisms are supplied also by a TCPA tile which makes it attractive for executing computationally intensive loop programs. In a TCPA, a parameterizable PE having VLIW architecture is replicated in the form of an array where all PEs are connected by a scalable mesh-like point-to-point interconnections. This circuit-switched interconnect allows data produced in a PE to be used by a neighboring PE in the next cycle. PEs are tightly coupled with each other similar to systolic processor arrays where each processor perform some computation and pass the results to the next processor to do some additional task using these results. To improve the data locality, small local buffers are provided surrounding the array similar to scratch pad memories. However, TCPAs do not have any cache memories and control-intensive tasks as well as data transfer to and from buffers may be initiated by a control processor as aforementioned.

### 2.5 Summary

In this chapter, initially the micro architecture of Tightly Coupled Processor Arrays (TCPAs) was presented focusing on the PE architecture, functional units and ISA, interconnect configuration, and configuration management. Later on, we presented TCPA editor, which is a tool used to dump out all the needed parameters for the hardware synthesis of a desired TCPA with intended number of PEs and FUs. We also discussed briefly integration of a TCPA as a tile of a heterogeneous MPSoC. In this context, we also enunciated the need for additional hardware components as well as their functionality in the TCPA tile. Finally, related work on the state of the art Coarse-grained Reconfigurable Arrays (CGRAs) as well as Massively Parallel Processor Arrays (MPPAs) found both in academia and industry was discussed.
In the previous chapters, we enunciated how the heterogeneous architectures are emerging as a requirement for energy-efficient system design. In this context, we also introduced the architecture of Tightly Coupled Processor Arrays (TCPAs) and how these can be integrated into a heterogeneous Multi-Processor System-on-Chip (MPSoC) design as a programmable accelerator. Yet, one of the most important challenges in using these programmable accelerators today is the lack of mapping tools or compiler frameworks that can increase the productivity of the programmer as well as shorten the time to market. Therefore, in this chapter, a brief overview of a compiler framework for mapping nested loop programs onto TCPAs is presented. We focus more on the fundamentals and background of the underlying models for algorithm and application specification. The main contributions of this chapter can be summarized as follows:

- We review the fundamentals of loop parallelization in the polytope model. In this context, we introduce an existing High-Level Synthesis (HLS) synthesis compiler framework PARO [HRDT08] on the basis of which code generation techniques for mapping nested loop programs onto TCPAs are elaborated in Chapter 4.

- We also enunciate few important open problems to be addressed in the back end of a compiler framework while mapping loop programs onto TCPAs. More specifically: Generating the code and interconnection structure for all Processing Elements (PEs) in a TCPA, and configuration code for the hardware components such as Global Controller (GC) and Address Generator (AG).

- Finally, we summarize the related work on HLS tools and mapping approaches for coarse-grained and embedded multi-processor arrays.

### 3.1 Loop Parallelization in the Polyhedron Model

The ubiquitous consumer markets such as mobile devices (smart phones, tablets) are the driving forces in the fast growing semiconductor industry. These devices
require most of the time the deployment of computationally intensive nested loop kernels where 80% of the execution time is typically spent in 20% of the program code to process signal, image, and video data [SMN+03]. These computationally intensive loop kernels contain massive inherent parallelism. With the recent availability of massively parallel compute architectures as introduced in Chapter 2, the potentials of loop parallelization have gained a lot of importance. Hence, a lot of research in the field of parallel computing has been spent on modelling, parallelization, scheduling, allocation, code generation, and memory management of loop nests [Dut11]. The polyhedron model has been recognized as a useful mathematical model to parallelizing loop programs for massively parallel compute architectures [FL11].

3.1.1 Preliminaries: Algorithm Specification

The polytope model is universally accepted as a basis for mathematical optimizations of loop nests in compiler theory [Fea96, Len93]. In this model, the iteration space of a perfect $n$-nested loop can be represented by a polyhedron in $\mathbb{Z}^n$ i.e., a convex subspace as defined below.

**Definition 3.1 (Z-polyhedron)** [Anc91] is mathematically defined as integer solutions of a system of affine inequalities

$$\mathcal{I} = \{ I \in \mathbb{Z}^n | A I + b \geq 0 \}$$

where $b \in \mathbb{Z}^m$ and $A \in \mathbb{Z}^{m \times n}$ denote a integer vector and integer matrix, respectively.

A polytope is a bounded polyhedron. Each integral point, denoted by $I$, in the polytope represents an iteration of the loop nest and all iterations form the iteration space, denoted by $\mathcal{I}$. The loop bounds of nested loop programs may be described by the intersection of set of half-spaces described each by an inequality, see Equation (3.1). For example, the loop nest of a matrix-matrix multiplication algorithm is given in Listing 3.1. Assuming the matrices dimensions as $8 \times 8$, the iteration space of the matrix-matrix multiplication algorithm can be given as the following $\mathbb{Z}$-polyhedron, visualized in Figure 3.1(b).

$$\mathcal{I} = \left\{ \begin{pmatrix} i \\ j \\ k \end{pmatrix} \in \mathbb{Z}^3 \mid \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 8 \\ 8 \end{pmatrix} \geq 0 \right\}$$ (3.2)
3.1 Loop Parallelization in the Polyhedron Model

Listing 3.1: Matrix-matrix multiplication.

```c
int A[8][8], B[8][8];
int C[8][8];
int i, j, k;
int sum;

//multiplication kernel
for (i = 0; i < 8; i++) {
    for (j = 0; j < 8; j++) {
        sum = 0;
        for (k = 0; k < 8; k++) {
            sum = sum + A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

(a)

(b)

Figure 3.1: A loop nest description of matrix-matrix multiplication (MMM) algorithm in 'C' in (a), and its corresponding iteration space are shown in (b).

In [Tei93], the concept of linearly bounded lattices (LBL) was introduced, which generalize \( \mathbb{Z} \) polyhedra to be able to represent any arbitrary \( n \)-dimensional integer point set.

**Definition 3.2** (Linearly Bounded Lattice (LBL) [Tei93]) *A linearly bounded lattice denotes an index space of the form*

\[
\mathcal{I} = \{ I \in \mathbb{Z}^n \mid I = M\kappa + c \land A\kappa \geq b \}
\]

*where \( \kappa \in \mathbb{Z}^l \), \( M \in \mathbb{Z}^{n \times l} \), \( c \in \mathbb{Z}^n \), \( A \in \mathbb{Z}^{m \times l} \) and \( b \in \mathbb{Z}^m \). \( \{ \kappa \in \mathbb{Z}^l \mid A\kappa \geq b \} \) denotes the set of integral points within a convex polyhedron or in case of boundedness within a polytope in \( \mathbb{Z}^l \). This set is affinely mapped onto iteration vectors \( I \) using an affine transformation \( I = M\kappa + c \).*

Throughout this dissertation, we assume that the matrix \( M \) is square matrix and of full rank. Then, each vector \( \kappa \) is uniquely mapped to an index or iteration point \( I \). Often, when the lattice matrix \( M \) is the identity matrix and the vector \( c \) is zero, the iteration space coincides with the definition of a polyhedron given in Equation (3.1).

All the needed properties of nested loops translate nicely to properties of polyhedra, for which many algorithms have been designed for the needs of optimization. Moreover, the polytope can be sliced or tiled for parallel execution.
and it can be transformed into another polytope without violating the data dependencies. This polytope contains the same number of points, but in a new coordinate system in which some directions are strictly temporal—i.e., scanning along them enumerates time—and the others are strictly spatial—i.e., scanning along them enumerates space. Such partitioning techniques have been studied in detail for compilers and are used for program acceleration through better cache reuse on sequential processors (i.e., loop tiling or blocking) [ND88, Tei93, Wol95] as well as algorithm’s implementation on parallel architectures, ranging from multi-core systems to supercomputers. One such partitioning transformation is introduced briefly in Section 3.2.3, where there are two spatial dimensions that correspond to a 2-D processor array and the remaining dimensions are executed temporally. The transformed polytope can then be used for code generation for the processors and dependencies between iterations translates into interconnections among the processors.

Furthermore, there exists already few open source ‘C’ libraries for the polyhedral models where most of the basic operations on polyhedra are implemented. In our framework in this dissertation, we use the open source library ‘polylib’ [Loe07]. An extension of Z-polyhedra is the parametric Z-polyhedron, which can model loops with parametric bounds [Fea88], which is also implemented as part of polylib. There is a rich mathematical framework for operations based on these polytope models. For example, the Fourier-Motzkin elimination algorithm [Sch86] can be used to find the existence of a solution to a system of affine inequalities given in Equation (3.1). Parametric integer programming (PIP) is an extension to handle parametric bounds and integer solutions [Fea88]. Ehrhart polynomials [Eug62] can be used for counting the number of integer points inside a parameterized Z-polytope as a function of its parameters [VSB+04]. All the needed functions to calculate these Ehrhart polynomials are also implemented in polylib [CLW97]. To summarize, loop nests can be best represented in the polyhedral model, which offers a rich mathematical framework for loop transformations and are available as open source libraries.

### Piecewise Regular Algorithms

Mapping of loop algorithms onto massively parallel architectures would require a data dependency analysis in order to make the inherent parallelism explicit. However, this process is very complex in sequential languages, since variables that are once defined can be often arbitrarily overwritten. A lot of work has been done in the area of high performance compilers for extracting parallelism from loop nests [Wol95]. One option to overcome this limitation is to start from recurrence equations in the polytope model that have been used to organize computations in a Single Static Assignment (SSA) manner with implicit order of execution. SSA is closely related to set of recurrence equations called...
Dynamic Piecewise Linear Algorithm (DPLA) [HT04, Han09]. DPLAs are a consistent advancement (e.g., handling of run-time dependent data flow to a certain extent) of the classes of Piecewise Regular Algorithm (PRA) as defined by Thiele in [Thi88] and Piecewise Linear Algorithm (PLA) as defined in [TR91], respectively. In fact, origins of these algorithm classes lies in the System of Uniform Recurrence Equations (SURE) introduced by Karp et al. in [KMW67], which was further extended to class of Regular Iterative Algorithms (RIA) by Rao in [Rao85]. However, in this dissertation, for the sake of brevity, we focus mainly on Piecewise Regular Algorithms (PRAs) that are defined as follows:

**Definition 3.3** (Piecewise Regular Algorithm (PRA)) A Piecewise Regular Algorithm (PRA) (paraphrased from Thiele’s work in [Thi88]) consists of a set of $K$ quantified recurrence equations, $S_1[I], \ldots, S_i[I], \ldots, S_K[I]$. Each equation $S_i[I]$ is of the form:

$$\forall I \in \mathcal{I} : x_i[I + f_i] = F_i(\ldots, x_j[I - d_{ji}], \ldots) \text{ if } I \in C_i$$

where $x_i$, $x_j$ are indexed variables, $F_i$ denotes an arbitrary function, and $f_i$, $d_{ji}$ are constant integral vectors of corresponding dimension. The dots … denote similar arguments. $\mathcal{I} \subseteq \mathbb{Z}^n$ is called global iteration space. The set of all vectors $I + f_i$, $I \in \mathcal{I}$, is called the index space of variable $x_i$. Furthermore, in order to account for irregularities in programs, we allow a quantified equation $S_i[I]$ to have iteration dependent conditionals, specified by an iteration space called condition space $C_i$. The condition space imposes further restrictions to the iteration space $\mathcal{I}$, that is, equation $S_i[I]$ is defined only for $I \in \mathcal{I} \cap C_i$. Both the global iteration space and all condition spaces $C_i$ are defined by polyhedral sets of the form $\{I \in \mathbb{Z}^n \mid A \cdot I \geq b\}$. If in addition $f_i$ is zero, an algorithm is said to be in output normal form. Then, $d_{ji}$ is called dependency vector, which denotes a data dependency from variable $x_j$ to variable $x_i$. In the rest of the dissertation, we assume that the global iteration space is rectangular (an orthotope).

**PAULA Language**

In our compiler framework, loop program is specified using the notations of Equation (3.3) in a language called PAULA [HRT08, Han09] instead of a sequential language such as ‘C’. Describing the algorithm in imperative form such as in ‘C’ has the disadvantage that their semantics force a lot of restrictions on the execution order of the program. Most of the parallelism contained in the original mathematical model of an algorithm is lost during the transformation to sequential code. For illustration, consider a simple summation $s = \sum_{i=0}^7 a[i]$, which is written in ‘C’ often as a for loop in the following manner:
### 3. Compiler Framework

**Figure 3.2:** Sequential and parallel specification of a sum consisting of eight two-input summands.

**Listing 3.2: A simple summation equation in C.**

```c
int s=0;
for(i=0; i<=7; i++) {
    s += a[i];
}
```

Here, a sequential order of summation is predefined and a later parallelization can become a crucial task—obviously not for this simple code fragment but, for instance, in case of pointer arithmetic, recursions, indirect array access, or arbitrarily mixed control and data flow. In Figure 3.2, the difference between a sequential and a parallel specification is shown. The latency may be reduced from linear to logarithmic time. One option to overcome this limitation is to directly start with a functional programming language such as PAULA [HRT08, Han09], where parallelism is given explicitly. PAULA follows the semantics of SSA, where each instance of a variable is defined only once. SSA is closely related to set of recurrence equations called Piecewise Regular Algorithm (PRA) as defined above.

In PAULA, the order of appearance of statements (equations) in a program does not matter; thus, can be arbitrarily interchanged. Similarly, the definition of iteration spaces does not imply any execution order of the loop body, that is, if there are no restrictions imposed by data dependencies, all iterations might be executed in parallel. Furthermore, we assume that a given PRA is computable which basically means that there exists an ordering of variable instances $x_i[I]$ appearing on the left hand side of an equation such that all variable instances $x_j[I-d_{ji}]$ occurring on the right hand side appear earlier in this ordering (schedulability). In Definition 3.3, we finally assume that each instance $x_i[I]$ of a variable appears only once on the left hand side in an equation; thus, guaranteeing the static single assignment property. SSA allows to apply manifold compiler optimizations and transformations in a very efficient way. Thus, SSA is also the method of choice as intermediate representation in state-of-the-art compilers (e.g., gcc [GCC] as of version 4 or LLVM [LA04])—however, only at basic block level, whereas our compiler framework can even handle this property at a higher abstraction level for multidimensional data flow. Other features of
3.1 Loop Parallelization in the Polyhedron Model

PAULA include an intuitive and compact syntax for describing iteration spaces, the deployment of reductions (big operators such as $\sum$, $\Pi$, min, etc.), and the advantage to describe also architectural parts of a target architecture like number and type of Functional Units (FUs), register file size, I/O ports, latencies and pipeline rates of FUs. All these parameters are considered automatically during the allocation and scheduling phases of our compilation flow.

At a first glance, the separation into calculations and iteration spaces seems to be similar to General Purpose Computation On Graphics Processing Unit (GPGPU) programming models such as OpenCL [Mun12] where a kernel only describes how to compute a single iteration of a loop, and later the kernel is invoked for an iteration space. In comparison to our approach, the main two differences of these General Purpose Computation On Graphics Processing Unit (GPGPU) programming models are: (a) they are limited to at most 3-dimensional arrays and (b) kernels do not allow to have loop-carried data dependencies. In the following, a FIR filter example is given in PAULA notation.

**Example 3.1** An FIR (finite impulse response) filter can be described by the simple difference equation $y_{\text{out}}(i) = \sum_{j=0}^{N-1} a_{\text{in}}(j) \cdot u_{\text{in}}(i-j)$ with $0 \leq i < T$, $N$ denoting the number of filter taps, $T$ denoting the number of samples over time, $a_{\text{in}}$ the filter coefficients, $u_{\text{in}}$ the filter inputs, and $y_{\text{out}}$ the filter results. The difference equation after parallelization, localization of data dependencies [TR91], and embedding of all variables in a common iteration space can be written as the following PAULA program with the global iteration space or domain $\mathcal{I} = \{ I = (i,j)^T \in \mathbb{Z}^2 \mid 0 \leq i \leq T - 1 \land 0 \leq j \leq N - 1 \}$.

**Listing 3.3**: FIR filter program in PAULA notation.

```plaintext
1 // resourcetype specifies the type of available
2 // functional units & allocation specifies how many
3 // such functional units of each type are available
4 resourcetype adder_16
5 { ops 10; // number of different instructions
6  input a integer<16>;
7  input b integer<16>;
8  output c integer<16>;
9 }
10 allocation adder_16 1;

13 resourcetype multiplier_16x16_16
14 { ops 2;
15  input a integer<16>;
16  input b integer<16>;
17  output c integer<16>;
18 }
19 allocation multiplier_16x16_16 1;
```

53
3. **Compiler Framework**

```c
resourcetype dpu_16
{ ops 1;
  input a integer<16>;
  output b integer<16>;
}
allocation dpu_16 5;

// binding possibility determines the possible functions
// which can be performed by a resource type
bindingpossibility function add(integer<16>,integer<16>)
{ op 0; // instruction id
  input a, b;
  output c;
  cycles 1;
}

bindingpossibility function sub(integer<16>,integer<16>)
{ op 1;
  input a, b;
  output c;
  cycles 1;
}

bindingpossibility function mul(integer<16>,integer<16>)
{ op 0;
  input a, b;
  output c;
  cycles 1;
}

...  

bindingpossibility function mov(integer<16>)
{ op 0;
  input a;
  output b;
  cycles 1;
}

program FIR
{

  variable a_in 2 in integer<16>;
  variable u_in 2 in integer<16>;
  variable a 2 integer<16>;
```

54
variable u 2 integer<16>;
variable x 2 integer<32>;
variable y 2 integer<40>;
variable y_out 2 out integer<40>;

parameter N = 32; // number of filter taps
parameter T = 32; // number of input samples

par (i ≥ 0 and i ≤ T-1 and j ≥ 0 and j ≤ N-1) {
    S₁: a[i, j] = a_in[i, j] if (i == 0);
    S₂: a[i, j] = a[i-1, j] if (i > 0);
    S₃: u[i, j] = u_in[i, j] if (j == 0);
    S₄: u[i, j] = 0 if (i == 0 and j > 0);
    S₅: u[i, j] = u[i-1, j-1] if (i > 0 and j > 0);
    S₆: x[i, j] = a[i, j]*u[i, j];
    S₇: y[i, j] = x[i, j] if (j == 0);
    S₈: y[i, j] = y[i, j-1]+x[i, j] if (j > 0 and j < N-1);
    S₉: y_out[i, j] = y[i, j-1]+x[i, j] if (j == N-1);
}

In Listing 3.3, by using keywords resourcetype and allocation, different functional units and number of such functional units are specified, respectively. These characterize basically the instruction level parallelism of the PEs and will be used for scheduling. The keyword bindingpossibility specifies on which resource type a particular function may be bound to executed on. The block in between program keyword denotes the actual algorithm and statements within keyword par block specifies the actual computation kernel of the algorithm. Intermediate variables in the program are specified using the keyword variable and number beside the identifier name denote the dimension of the variable. Input and output variables are differentiated by in and out keywords, respectively. Within the par statement, iteration space of the algorithm is specified. Each statement may contain an additional condition space that is specified using if statement as shown in Listing 3.3. Dependences among the different statements are specified using the dependence vectors. For FIR filter algorithm, the set \( D = \{(1 0)^T, (0 1)^T, (1 1)^T\} \) corresponds to the set of all dependence vectors. Moreover, the global iteration space of this program is given by \( I = \{(i j)^T \in \mathbb{Z}^2 \mid 0 \leq i \leq T - 1 \wedge 0 \leq j \leq N - 1\} \). For further details on the syntax and semantics of writing PAULA programs, we refer to [HRT08].

**Reduced Dependence Graph (RDG)**

A Dependence graph leverage the geometrical representation of the polyhedral model for modeling single assignment algorithms as well as PRAs. The topology of the unfolded dependence graph (also known as iteration space graph)
Compiler Framework

is specified by the iteration space, i.e., each node of the graph is an iteration of the loop nest. An edge contains the dependency information consisting of a variable name and the relative displacement of the source and target iteration—specifying the partial ordering among the computations in the algorithm. For instance, if there is a directed path in the dependence graph from node $x[J]$ to node $y[K]$ with $J, K \in \mathbb{Z}$, then computation represented by node $y[K]$ must be executed after node $x[J]$. The representation of the unfolded graph can be very large if not bounded, as the number of nodes is equal to the number of iterations. In case of algorithms with uniform dependencies, dependencies are regular throughout the algorithm ($x[I]$ is directly dependent on $y(I - d)$ irrespective of any $I$ value). The dependence graph is regular and can be folded to a Reduced Dependence Graph (RDG) [RK88, Thi88], defined as follows.

**Definition 3.4 (Reduced Dependence Graph (RDG))** Let a PRA is given in the output normal form with $\mathbb{I} \subseteq \mathbb{Z}^n$. Then a corresponding Reduced Dependence Graph (RDG) $G = (V, E, D)$ of dimension $n$ has the following properties. There exists a set $V$ of nodes and a set of edges $E \subseteq V \times V$. The number of nodes is equal to the number of statements (equations) plus the number of input variables. In addition, there exists a mapping $E \rightarrow D$ that assigns to each edge $e = (v_i, v_j) \in E$ an $n$-dimensional dependency vector $d_{ij} \in D$ if a variable $x_j$ of the algorithm depends on a variable $x_i$ of the algorithm.

In our compiler framework, once a PAULA program is parsed, it is represented by an extended version of RDG internally, as this is a compact graph representation of a corresponding loop program formulated by recurrence equations of the PRA. The extended version of the RDG used in our compiler design flow, contains different types of nodes and edges, each with attributes denoting the variable, functionality, the iteration space, and the indexing function [Han09]. For illustration, an example algorithm for matrix-matrix multiplication is given in PAULA notation in Listing 3.4, and its corresponding RDG is shown in Figure 3.3. Each node is associated with an equation and corresponds to an operation. The edges are annotated with dependence vectors which denote either intra-iteration dependences if dependence vector $d_{ij} = 0$ or inter-iteration dependencies if dependence vector $d_{ij} \neq 0$. This kind of representation of the PRA using RDG is needed for further steps in our mapping approach such as local allocation and scheduling as described in the next sections.

Listing 3.4: Matrix-matrix algorithm in PAULA notation.

```plaintext
program matmul
{
  variable A 3 in integer<16>;
  variable B 3 in integer<16>;
  variable C 3 out integer<40>;
  variable a 3 integer<16>;
```
3.2 Compilation Flow

In this section, we give an overview of our compiler framework PARO [HRDT08] for mapping nested loop programs onto tightly coupled processor arrays. This tool has been developed to synthesize custom hardware accelerators from a given input PAULA program. However, in this dissertation, we focus mainly on compiling nested loop programs onto programmable hardware accelerators such as TCPAs where the array architecture is already available. Our work extends PARO by providing a compilation rather than synthesis flow. It is mainly divided into two parts: front end and back end as shown in Figure 3.4. In the following, we first explain the front end of the compilation flow.

3.2.1 Front End

In Step (1) of the front end, a nested loop algorithm is specified in PAULA notation. In Step (2) of our compilation flow, based on the given algorithm, various source-to-source compiler transformations and optimizations are applied for better usage of the registers and compute resources such as FUs. De facto compiler optimizations like common sub-expression elimination (CSE), constant and variable propagation, dead-code elimination, operator strength reduction, expression splitting, and others [Muc97] are applied. The CSE transformation identifies identical expressions within a program and analyzes if it is worth to replace an expression with an intermediate variable to store the computed value.
Dead-code elimination removes redundant code which is either unreachable or which is executed but whose results are never used in any other computation.
3.2 Compilation Flow

Figure 3.4: Extending PARO [HRDT08]: The compilation flow for mapping a loop program onto a TCPA. The blocks highlighted in blue belong to the back end of the compiler and build the major pieces of the dissertation.

Operator strength reduction replaces costly operations like integer divisions and multiplications with a combination of shifts, adds, and subtractions. Of course, this transformation results in increased registers usage and an increase in latency of the algorithm. In the following, few more source-to-source transformations are explained in brief.

3.2.2 High-Level Transformations

Replace Operators

Replace denotes a simple transformation that replaces operators such as multiplications by functions as shown below for a single equation.

\[ S6 : x[i, j] = a[i, j] \cdot u[i, j] \]

after transformation
This transformation is helpful during the resource binding stage and the scheduler searches for the appropriate functional unit to implement the function of this equation. As this transformation just replaces operators by functions, it will neither increase the number of equations nor the code size.

Expression Splitting

This transformation splits lengthy expressions with more than one operation/-function into a number of simpler expressions. For example, consider the following equation adding three variables. If we have an adder FU in a PE that can add only two inputs at a time, we need to split the equation into two equations as shown below by introducing an intermediate variable $z[i,j]$.

\[
EQ_0 : y[i,j] = p[i,j] + q[i,j] + r[i,j]
\]

after transformation

\[
EQ_{0.1} : z[i,j] = p[i,j] + q[i,j]
\]

\[
EQ_{0.2} : y[i,j] = z[i,j] + r[i,j]
\]

This transformation will increase the iteration latency and the code size.

Loop Unrolling

Loop unrolling is also known as loop unwinding. It is a classical compiler optimization technique that originally attempts expose the parallelism in a loop program. Loop unrolling expands the loop kernel by a factor of $n$ by copying $n-1$ consecutive iterations, which leads to larger data flow graphs at the benefit of possibly more Instruction-Level Parallelism (ILP).

Localization

Localization [TR91] is a transformation which converts affine data dependencies into uniform data dependencies by the propagation of variables from one iteration point to neighboring iteration points [Tei93, MMRR01]. Non-uniform data-dependencies are not suitable for mapping onto regular processor arrays as they result in expensive global communication in terms of memory access. A localization transformation as proposed in [TR91], replaces affine dependencies by regular dependencies and enables maximum data reuse within the processor array; thus minimizes the amount of external I/O communication (with peripheral memory) by replacing broadcasts with short propagations.

After PAULA programs read and parsed, transformations are applied in the front end of our compiler framework. Subsequently, using the function declarations in the architecture model (see Listing 3.3) and based on the statements
or expressions, each node is assigned a return data type. This information on the return data type and function name is a node attribute in the reduced dependency graph (see Figure 3.3). After replacing each operations by a function, propagation of data type, and expression splitting, a RDG can be generated. The nodes of the RDG are annotated with binding possibilities, return data type, equation number, node type, variable name, and iteration space, see Figure 3.3 for instance for the matrix-matrix multiplication example. The most important transformation in our design flow is partitioning which is discussed in the next subsection.

3.2.3 Partitioning

Loop partitioning is a well known transformation which covers the iteration space of computation using congruent hyperplanes, hyperquaders, or parallelepipeds called tiles. The major purpose of partitioning is to map a loop nest of given size onto an array of processors of fixed size and therefore map tiles to processors and schedule these accordingly. However, the transformation has been studied also in detail for compilers with the purpose of acceleration through better cache reuse on sequential processors and known as loop tiling or blocking [Wol95] targeting implementations on parallel architectures ranging from supercomputers to multi-DSPs and FPGAs. Yet, tiling transformation have been proposed first in the area of mapping iterative applications to systolic arrays [ND88, Tei93]. Well known partitioning techniques are multi-projection, Locally Sequential Globally Parallel (LSGP) and Locally Parallel Globally Sequential (LPGS) [DHT06b]. These strategies allocate the iterations to Processing Element (PE) in a processor array.

Formally, partitioning divides the iteration space \( \mathcal{I} \) using congruent tiles such that it is decomposed into spaces \( \mathcal{I}_1 \) and \( \mathcal{I}_2 \) as follows

\[
\mathcal{I} \subseteq \mathcal{I}_1 \oplus \mathcal{I}_2 = \mathcal{I}', \text{ where} \\
\mathcal{I}' = \{ I = I_1 + P \cdot I_2 \mid I_1 \in \mathcal{I}_1 \land I_2 \in \mathcal{I}_2 \land P \in \mathbb{Z}^{n \times n} \} 
\]

\( \mathcal{I}_1 \subseteq \mathbb{Z}^n \) represents the points within the tile and \( \mathcal{I}_2 \subseteq \mathbb{Z}^n \) accounts for the regular repetition of the tiles, i.e., the origin of each tile. We assume that a tile is an orthotope that can be defined by a diagonal tiling matrix \( P \).

**Example 3.2** For the FIR filter example given in Listing 3.3, the iteration space is shown in Figure 3.5, which is partitioned for execution on a 4x4 processor array using the LSGP method. Figure 3.6 shows the partitioned iteration space. Note that for the sake of illustration, the number of samples \( T \) is limited to 32 in the Figure 3.5. The tiling matrix used for partitioning the filter onto a 4 x 4
Figure 3.5: Iteration space of the FIR filter program.

The processor array is \( P = \begin{pmatrix} 8 & 0 \\ 0 & 8 \end{pmatrix} \). It should be noted that tiling doubles the number of iteration variables in each dimension. Hence, iteration variable \( i \) is represented using \( i_1, i_2 \) and iteration variable \( j \) is represented using \( j_1, j_2 \) in the tiled iteration space (see Figure 3.6). For instance, it can be verified that the index point \( I = (10 \ 10)^T \) where \( I = (i \ j)^T \) is uniquely mapped to \( I_1 = (2 \ 2)^T \), \( I_2 = (1 \ 1)^T \), where \( I_1 = (i_1 \ j_1)^T \in \mathcal{I}_1 \) and \( I_2 = (i_2 \ j_2)^T \in \mathcal{I}_2 \) with \( \mathcal{I}_1 = \{ I_1 \in \mathbb{Z}^2 \mid 0 \leq i_1 < 8 \land 0 \leq j_1 < 8 \} \), and \( \mathcal{I}_2 = \{ I_2 \in \mathbb{Z}^2 \mid 0 \leq i_2 < 4 \land 0 \leq j_2 < 4 \} \).

Note that partitioning may also introduce new equations and respective iteration dependent conditions in the algorithmic description to keep the data dependencies intact. Partitioning also embeds the algorithm in the new iteration space with new iteration variables. For illustration, we refer to Listing 3.5, which shows the new equations along with their condition spaces.
3.2 Compilation Flow

Figure 3.6: Tiled iteration space of the FIR filter program.

Listing 3.5: Partitioned FIR filter program in PAULA notation.

```plaintext
program FIR_partitioned
{
variable a_in 4 in integer<16>;
variable u_in 4 in integer<16>;
variable a 4 integer<16>;
variable u 4 integer<16>;
variable x 4 integer<32>;
variable y 4 integer<40>;
variable y_out 4 out integer<40>;

parameter T_1 = 8;
parameter T_2 = 4;
```
3. Compiler Framework

In [TT93], a transformation for tiling based on a partitioning matrix that automatically generates the tiled iteration space, embeds the given statements in the new tiled iteration space, preserves the data dependences, and may introduce extra equations with appropriate condition spaces to keep the data dependences intact is proposed. In [TTZ97], this transformation has been extended to apply even for PLAs with affine data dependencies.

3.2.4 Space-Time Mapping

In Step (3) of our compilation flow according to Figure 3.4, we need to tackle the problem of scheduling which assigns loop iterations and loop body operations to time steps for execution on the Processing Elements (PEs) in the TCPA. In addition, the loop computations must be allocated to the PEs with optimal granularity in terms of computational intensity, data parallelism, and locality.

In our compiler framework, for scheduling and allocation, the following affine transformation is used for PE allocation and scheduling

\[
\begin{pmatrix}
  p \\
  t
\end{pmatrix} = \begin{pmatrix}
  Q \\
  \lambda
\end{pmatrix} \cdot I + \begin{pmatrix}
  q \\
  \gamma
\end{pmatrix}
\]

(3.5)

which assigns to each iteration point \( I \in \mathcal{I} \) a processor index \( p \in \mathcal{P} \) (allocation) and a time index \( t \in \mathcal{T} \) (scheduling). Here, \( Q \in \mathbb{Z}^{s \times n} \) and \( \lambda \in \mathbb{Z}^{1 \times n} \) denote the allocation matrix and scheduling vector, respectively with \( s < n \). The offset
3.2 Compilation Flow

Vectors $q \in \mathbb{Z}^*$, and $\gamma \in \mathbb{Z}$ are optional. The set $\mathcal{T} = \{t \mid t = \lambda I + \gamma \land I \in \mathcal{I}\} \subset \mathbb{Z}$ is called space of start times, that is the set of all time steps where an execution takes place. The set $\mathcal{P} = \{p \mid p = QI + q \land I \in \mathcal{I}\} \subset \mathbb{Z}^*$ is called processor space, that is the set of all processor indices on which loop body execution takes place. Its cardinality $|\mathcal{P}|$ denotes the number of processors. In the literature, the affine transformation Equation (3.5)$^4$, is known as space-time mapping[Rao85, Kun87, Len93], or scheduling function and processor allocation [FM01].

The processor allocation requires the determination of $Q$ which depends on the tiling strategy. The tiling (partitioning) transformation initially decomposes the iteration space into inner and outer loops, $\mathcal{I} \subseteq \mathcal{I}_1 \oplus \mathcal{I}_2$, see Equation (3.4). Depending on the tiling scheme, the iterations within a tile have to be executed in parallel LPGS or sequentially LSGP, and the tile itself will be executed either sequentially LPGS or in parallel LSGP. The advantage of the LPGS method is in the minimal requirement of local memory for each PE. However, this benefit is offset by high communication costs and external memory requirements. In LSGP, the communication cost is minimal but either the amount of local memory or the number of required PEs is controlled by the selection of the tile size. Therefore, the programmer plays a major role in determining the degree of parallelism through tile size and tiling strategy selection.

Mathematically, the processor allocation is given by

$$ p = \begin{pmatrix} E \end{pmatrix} \cdot \begin{pmatrix} I_1 \cr I_2 \end{pmatrix} \quad (LPGS) \quad \begin{pmatrix} 0 \cr 0 \cr 0 \cr 1 \end{pmatrix} \cdot \begin{pmatrix} I_1 \\
 I_2 
\end{pmatrix} \quad (LSGP) \tag{3.6} \tag{3.7} 
$$

where $0$ and $E$ denote the zero and identity matrix, respectively. $I_1$ and $I_2$ each denote $n$-dimensional iteration vectors for indexing of iterations within a tile and origins of tiles, respectively. The allocation matrix, $Q$ contains information on which loop variable is executed sequentially or in parallel. In other words, $Q$ is a linear allocation matrix which is determined by the tiling strategy (i.e., LPGS, and LSGP).

**Example 3.3 (LSGP partitioning)** For LSGP partitioning, and a two dimensional iteration space and as well processor space, a space time mapping transformation without any offset vectors could be defined by the following linear transformation:

$$ \begin{pmatrix} p_1 \\
p_2 \\
t \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\lambda_1 & \lambda_2 & \lambda_3 & \lambda_4 \end{pmatrix} \cdot \begin{pmatrix} I_1 \\
 I_2 \end{pmatrix} \tag{3.8} \end{pmatrix} $$

where $I_1 = (i_1 \ j_1)^T \in \mathcal{I}_1$ and $I_2 = (i_2 \ j_2)^T \in \mathcal{I}_2$. Here, the schedule vector $\lambda = (\lambda_1 \ \lambda_2 \ \lambda_3 \ \lambda_4)$ provides the start time iteration $I = (I_1 \ I_2)^T$. The vector $(p_1 \ p_2)^T \in \mathcal{P}$ is also called the processor index.

$^4$Moldován [Mol83] uses already a similar linear transformation but he requires the transformation to be bijective.
The scheduling function is represented by the affine transformation
\[ t_i(I) = \lambda \cdot I + \tau(v_i) \]  
(3.9)

Here, \( \lambda \) is called the global schedule vector. \( \tau(v_i) \) denotes the start time of each individual statement \( S_i[I] \) in the loop body. Thus, the overall start time of statement (node) \( v_i \) of a Reduced Dependence Graph (RDG) of iteration point \( I \) is given by Equation (3.9). In other words, the schedule vector \( \lambda \) determines the global schedule, and \( \tau(v_i) \) determines the local schedule of starting the execution of an equation \( S_i \) at iteration \( I \), respectively. In case of a tiled iteration space with LSGP or LPGS tiling, the scheduling function may be defined as follows:

\[ t_i(I) = (\lambda_{\text{par}} \lambda_{\text{seq}}) \begin{pmatrix} I_1 \\ I_2 \end{pmatrix} + \tau(v_i) \quad (LPGS) \]  
(3.10)

\[ t_i(I) = (\lambda_{\text{seq}} \lambda_{\text{par}}) \begin{pmatrix} I_1 \\ I_2 \end{pmatrix} + \tau(v_i) \quad (LSGP) \]  
(3.11)

where \( \lambda_{\text{par}} \) and \( \lambda_{\text{seq}} \) are the parts of the schedule vector, which correspond to iteration variables being executed in parallel and sequentially, respectively. For the iterations or tiles which shall be executed sequentially, a sequential order of execution, also called scanning order, has to be defined. This information may be represented by a loop matrix, which is defined as below.

Definition 3.5 (loop matrix [Tei93, Han09]) A non-singular\(^5\) loop matrix \( R = (r_1 \ r_2 \ \ldots \ r_n) \in \mathbb{Z}^{n \times n} \) consists of \( n \) loop vectors that determines the ordering of iteration points within a parallelepiped-shaped tile. The iteration points in direction of \( r_1 \) are mapped side by side. Iteration points in direction of \( r_2 \) are separated by blocks of points in direction \( r_1 \) and so on. The ordering is similar to a sequential nested loop program where the loop index \( i_k \) corresponds to iterations in direction of \( r_k \). The inner loop index is \( i_1 \), and the outermost loop index is \( i_n \).

Example 3.4 Figure 3.6 shows the tiled iteration space of a FIR filter algorithm.

After selecting the loop matrix \( R = \begin{pmatrix} 0 & 8 \\ 8 & 0 \end{pmatrix} \) to scan the iterations within a tile and choosing the LSGP partitioning strategy, a space time mapping is obtained as shown Figure 3.7. The processor allocation implies that the iteration variables \( i_2 \) and \( j_2 \) are executed in parallel and denote the processor dimension, whereas \( i_1 \) and \( j_1 \) are executed sequentially. The resulting processor array implementation is shown Figure 3.7 where the number of processors is equal to the number of tiles.

\(^5\)A square matrix is not singular i.e., one that has a matrix inverse.
3.2 Compilation Flow

Local Scheduling and Resource Binding

The purpose of scheduling is to determine a schedule vector $\lambda$ that determines the global schedule denoting the start time of iterations as well as a local schedule $\tau(v_i)$ that denotes the start times of execution of statement $S_i[I]$ different operations corresponding to nodes in the RDG. An affine-by-statement schedule according to Equation (3.9) may be obtained by solving a latency minimization problem formulated as a mixed integer linear program similar as presented in [Thi95, Han09]. In the following, the objective function is the minimization of Global Latency (GL), defined as the total execution time of the loop nest,
i.e., the time difference between first and last loop iteration. In our compiler framework, software pipelining is employed to minimize the global latency. The execution time of a single iteration is called Local Latency (LL). It denotes the number of time steps that elapse between the first and the last computation of all \( K \) equations belonging to a single iteration \( I \), is also determined during scheduling.

The purpose of the resource binding step is to assign each node in the RDG a functional unit (resource) so that it can execute the node functionality or operation. In the architecture specification, (see for e.g., Listing 3.3), allocation denotes for each resource type \( r_k \), the number of available instances per PE \( \alpha(r_k) \). The binding possibility \( \beta(v_i) \) defines the resources (such as adders, multipliers, and other functional units) on which the functionality of the RDG node \( v_i \) can be executed. The execution time \( w(r_k) \) and the pipeline rate \( p(r_k) \) of a resource \( r_k \) is also given in the architecture model. After binding is done, each node in the RDG is annotated with a resource type, see e.g., Figure 3.3.

Here, the problem of resource-constrained scheduling and binding is solved using a mixed integer linear programming (MILP) formulation similar as in [TTZ97, HT04, Han09]. Note that this approach may determine an optimal schedule in terms of global latency (GL) for an entire loop nest w.r.t. to a set of given resource constraints (see Listing 3.3), instead of traditional modulo scheduling techniques [Rau94], which schedule only the innermost loop. Additionally, beside the number of considered PEs and FUs within them, our approach encompasses also register and interconnect constraints. By solving the Mixed Integer Linear Programming (MILP) using a state-of-the-art LP solver such as CPLEX [ILO11], both a schedule vector \( \lambda \) and start offsets (\( \tau \)) for each node are obtained. A simplified MILP formulation is given below. Note that for the sake of brevity, the constraints for the computation of \( \tau \) for all nodes \( v_i \) are omitted.

### Listing 3.6: A simplified MILP formulation for affine scheduling.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Input:</strong></td>
</tr>
<tr>
<td>2</td>
<td>Reduced dependency graph ( G = (V, E, D) )</td>
</tr>
<tr>
<td>3</td>
<td>Execution time ( w_i ) of each node ( v_i \in V )</td>
</tr>
<tr>
<td>4</td>
<td>Iteration space ( I ) is defined by a polyhedron ( I = { I \in \mathbb{Z}^n \mid AI \geq b } ) where ( A \in \mathbb{Z}^{m \times n} )</td>
</tr>
<tr>
<td>6</td>
<td><strong>Output:</strong></td>
</tr>
<tr>
<td>7</td>
<td>Schedule vector ( \lambda \in \mathbb{Q}^{1 \times n} )</td>
</tr>
<tr>
<td>8</td>
<td>Start times ( \tau(v_i) \in \mathbb{Q} ) of all nodes ( v_i \in V )</td>
</tr>
<tr>
<td>9</td>
<td><strong>Program:</strong></td>
</tr>
<tr>
<td>10</td>
<td>minimize (- (y_1 + y_2) b )</td>
</tr>
<tr>
<td>11</td>
<td>subject to ( y_1 A = \lambda ) ( y_1 \in \mathbb{Q}^{1 \times n} )</td>
</tr>
<tr>
<td>12</td>
<td>( y_2 A = -\lambda ) ( y_2 \in \mathbb{Q}^{1 \times n} )</td>
</tr>
<tr>
<td>13</td>
<td>( y_1 \geq 0 )</td>
</tr>
<tr>
<td>14</td>
<td>( y_2 \geq 0 )</td>
</tr>
<tr>
<td>15</td>
<td>( \lambda d_e + \tau(v_i) - \tau(v_j) \geq w_j \quad \forall e = (v_j, v_i) \in E )</td>
</tr>
</tbody>
</table>
The above simplified MILP does not take any resource constraints into account. For the full description, we refer to [Han09]. To determine the start time $\tau(v_i)$ of node $v_i$, binary variables $x_{i,k,t} \in \{0,1\}$ are introduced (similar as in [Thi95]). They encode not only the start times of the nodes but also may realize the selection from different binding possibilities—for instance, a `mov` instruction can be issued either on an adder or a multiplier unit. Thus, module selection and binding can be incorporated into the MILP. As each operation can be executed on one resource type $r_k$ only, we need to add the following binding constraint, where $(v_i,r_k) \in E_R$ specifies a valid binding possibility from one node $v_i$ to a functional unit of type $r_k$ (see e.g., Listing 3.3).

$$\sum_{\forall k : (v_i,r_k) \in E_R} \sum_{t=l_i}^{h_i} x_{i,k,t} = 1 \quad \forall v_i \in \mathcal{V} \quad (3.12)$$

The first sum in the constraint in Equation (3.12) denotes that a node $v_i$ should be executed every time on the same resource type, that is, its resource type binding is static. The second sum makes sure that a node is started exactly once per iteration. The bounds $l_i$ and $h_i$ denote the earliest and latest possible start time of a node $v_i$, respectively.

The relationship between the relative start times $\tau(v_i)$ and binary variables $x_{i,k,t}$ is obtained from the binary variables as follows:

$$\tau(v_i) = \sum_{\forall k : (v_i,r_k) \in E_R} \sum_{t=l_i}^{h_i} t \cdot x_{i,k,t} \quad \forall v_i \in \mathcal{V} \quad (3.13)$$

The above MILP formulation is extended further with resource constraints such as global number of processors, register file size within each processor, number of interconnections between processors, and conditional scheduling. However, these techniques are out of the scope of this dissertation and we refer to [Han09] for further details.

**Example 3.5** For the FIR filter of Listing 3.3 and for $N = T = 32$, a feasible space-time mapping of the LSGP partitioned iteration space to an array of $4 \times 4$ processors is given by Equation (3.14).

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 8 & 1 & 57 & 8 \end{pmatrix} \cdot \begin{pmatrix} I_1 \\ I_2 \end{pmatrix} \quad \text{where} \quad I_1 = \begin{pmatrix} i_1 \\ j_1 \end{pmatrix}, \quad I_2 = \begin{pmatrix} i_2 \\ j_2 \end{pmatrix} \quad (3.14)$$

After scheduling, for the implementation shown in Figure 3.7, an Iteration Interval (II)$^6$ of 1 cycle, a LL of 3 cycles, and a GL of 261 cycles is obtained.

$^6$The Iteration Interval (II) denotes the number of clock cycles between the evaluation of two successive iterations.
Once both schedule vector $\lambda$ and $\tau$ are known, the exact execution time of each operation $v_i$ (statement $S_i[I]$ in the program) for each iteration $I$ is also known. For instance, variable ‘$x$’ for iteration $I_1 = (4, 2)^T$ is mapped to PE(0,0) (i.e., $I_2 = (0, 0)^T$) and is executed at time $\lambda_1 \cdot I_1 + \lambda_2 \cdot I_2 + \tau(x)$; i.e., at cycle $(8, 1) \cdot (4, 2)^T + (57, 8) \cdot (0, 0)^T + 1 = 35$. In Figure 3.7, numbers on the top-left corner of each tile denote the start time of the processor for this schedule vector.

Putting it altogether, given a RDG, iteration space, processor allocation function, loop matrix, and resource constraints, one can determine a linear global schedule $\lambda$ and local schedule $\tau$ by minimizing the overall latency using a MILP formulation. In LSGP, $\lambda$ has to be determined such that all iterations within a tile are executed sequentially (by the same processor) whereas the iterations of different tiles may be executed in parallel by different processors while satisfying all data dependencies. Figure 3.8(a) shows the RDG after scheduling where the result of resource binding i.e., functional unit allocation is annotated back to the RDG. Here, the allocation of resources within a PE is assumed as follows: 1 adder unit, 1 multiplier unit, and 5 DPUs. It also shows the start times $\tau$ of the nodes. In Figure 3.8(b), functional unit allocation for first four iterations is shown. However, this allocation remains the same for all other iterations as well. It should be noted that in Figure 3.8(b) both statement $y$ and $y_{out}$ are assigned the same functional unit since both of these statements are mutually exclusive i.e., at any time only one of them is executed. Therefore, only one adder functional unit is assigned for both of these statements. This type of scheduling is called conditional scheduling which is an important feature of the scheduler used in our compilation flow.

3.2.5 Back End

Finally, all the required information such as schedule and functional unit allocation which are necessary for the tasks of program code generation (back end) are annotated to the RDG itself. In the following, these important tasks of the back end, see Figure 3.4, on which this dissertation focuses on are explained briefly to give an overview of the rest of the chapters as well as the major contribution of this dissertation.

1. **Code and Interconnect Generation**: The most important step in the back end is the code and interconnect generation for all the processors present in the array. In fact, these codes and interconnects have to be generated from the same loop program which was scheduled and partitioned in front end of the compiler. In Chapter 4, we present an automatic code generation approach that is scalable, preserve the given schedule and generates the most compact code possible. In addition, we also present a scalable interconnect generation approach.
2. **Global Controller**: In the overall TCPA tile architecture, the global controller plays an important role in synchronizing the execution of the generated codes correctly on the processors. For example, it generates the control signals to steer start times of program execution, see Figure 3.7. From the hardware point of view, the global controller has to be a programmable component so that its configuration data can also be changed depending on the application and computed schedule. In Chapter 5, we discuss not only the design of such a programmable global controller but also generating its configuration data automatically in the back end of the compiler from a given scheduled loop program.

3. **Address Generators**: As aforementioned in Chapter 2, Address generators are responsible for generating the address and enable signals for the reconfigurable buffers surrounding the processor array so that appropriate data is read/written from/to buffers according to the pre-computed schedule. These address generators have to be also programmable hardware components so that their configuration data can be changed depending on the application. In Chapter 6, we discuss design of such programmable address generators and generating their configuration data automatically in the back end of the compiler.

Putting it altogether, in the back end of our compiler framework, code for Processing Elements (PEs), interconnections among them are generated auto-
matically based on a given scheduled loop program. Configuration data for the
global controller as well as address generators is automatically extracted from
the same input program. Later on, a configuration and communication processor (LEON3 RISC core) in the TCPA tile is used to load the generated programs
into the PEs, configure interconnections among PEs, global controller, and ad-
dress generators. Once all the components are configured, the parallel execution
of the loop program starts according the schedule by reading from the input local
buffers and writing the results back to the output local buffers. Meanwhile, the
global controller generates the controls signals to steer the programs running on
the PEs. Address generators generate addresses as well as enable signals for the
buffers at appropriate times so that the data is read/written from/to buffers.

3.3 Related Work

This section about related work is separated into two parts, each having its
own perspective. In the first part, synthesis of processor arrays and high level
synthesis tools both from industry and academia are discussed. The second part
elucidate mapping approaches for coarse-grain reconfigurable architectures and
embedded multi-processor arrays.

3.3.1 High-Level Synthesis Tools

In this subsection, a brief overview of few HLS tools is given emphasizing their
model of computation, transformations used, input language.

**MMAlpha [GQR08]:** MMAlpha is a HLS tool which also relies on the founda-
tions of the polyhedral model and recurrence equations. In this framework, sys-
tems of recurrence equations are described by means of the Alpha programming
language [LMQ90] which follows the semantics of single assignment language.
Later on, Alpha and its associated synthesis environment MMAlpha provide
a framework for the synthesis of regular architectures. In particular, how to
go from an affine system of recurrence equation to a uniform system of recur-
rence equation has been studied and implemented in the MMAlpha environ-
ment [MMRR01]. In other words, MMAlpha consists of all the required math-
ematical packages enabling a design trajectory down to synthesizable VHDL
code. In [DDR05], a methodology for the generation of a hardware/software
interface for system integration is proposed. Scheduling techniques for both one
and multi-dimensional time, are available. However, MMAlpha does not contain
any architectural modeling possibilities as for instance, multi-cycle operations,
modulo selection, or resource-constrained scheduling.
3.3 Related Work

**Figure 3.9:** PICO’s [KAS+02] system-level architecture template showing three subsystems: a custom EPIC/VLIW processor, a custom two-level hierarchy, and an NPA subsystem that includes one or more custom NPAs.

**PICO [KAS+02]:** The PICO (program in, chip out) project at HP Labs automates the design of optimized application-specific computer systems. PICO uses an application written in ‘C’ to architect a set of high-quality system designs that trade cost for performance. PICO system design contains one EPIC/VLIW (explicitly parallel instruction computing/very long instruction word) processor and an optional non-programmable accelerator (NPA) subsystem consisting of one or more NPAs, both connected to a two-level cache subsystem that, in turn, connects to the system bus. Each NPA is customized to execute a compute-intensive loop nest that would otherwise have been executed on the VLIW. The overview of architecture template is shown in Figure 3.9. PICO emits structural Verilog/VHDL for the hardware components, modifies application code to include software interfaces to the generated hardware, and retargets the compiler, assembler, and simulator to the custom VLIW processor. PICO includes transformations for the minimization of global memory accesses (instantiation of local registers and exploitation of uniform data dependencies) and rectangular tiling. In case of tiling, the loop nest is extended by one outer loop that is implemented in software on the host processor, and the inner loops are synthesized as an NPA. Scheduling is performed in a hierarchical fashion. First, a schedule between the iterations is determined, before the offsets for each operation within the loop body are computed subsequently. PICO was one of the
first design flows integrating design space exploration. The research project has been commercialized by Synfora [Syn09], and now part of Symphony C compiler from Synopsys [Syn12].

**ROCCC** [GNB08]: ROCCC (Riverside Optimizing Configurable Computing Compiler) is a compilation tool, formerly known as SA-C [NBD+03], which compiles code written in C/C++ or Fortran to VHDL code for mapping onto the FPGA fabric of a configurable System-on-Chip (SoC). In the execution model underlying ROCCC, sequential computations are carried out on the microprocessor, while the compute intensive code segments such as loop nests, often operating on large arrays or streams of data, are mapped onto the FPGA. The front end of ROCCC performs a very extensive set of loop analysis and transformations aimed at maximizing parallelism and minimizing the area. The transformations include loop unrolling and strip mining, loop fusion, and common sub-expression elimination across multiple loop iterations. Most of the information needed to design high-level components, such as controllers and address generators, is extracted automatically from the intermediate representations. ROCCC uses Machine-SUIF virtual machine (SUIFvm) [HS02] intermediate representation in the back end. ROCCC has a strong emphasis on filter and image processing algorithms with window operations. Figure 3.10 shows the ROCCC machine model for a window operator execution. Outstanding in the design flow is the ability to generate *smart buffers* for storage to decouple memory access from computations. The smart buffers also facilitate data reuse in loop algorithms through an architecture consisting of dedicated address generators and multiple connected registers.

**DEFACTO** [DHP+05]: Design Environment For Adaptive Computing Technology, is a compiler technology to synthesize RTL descriptions from a high-level imperative languages, such as C. Programmers retain the advantages of a simple computational model via a high-level language but rely on powerful compiler analysis to identify parallelism, provide high-level loop transformations, and automate most of the tedious and error-prone mapping tasks. In DEFACTO, the interaction between compiler transformations, most notably loop transformations, and their hardware realizations on FPGAs using commercially available synthesis tools are explored. DEFACTO performs several code transformations that are useful for tailoring code to space constraints, such as loop tiling, and loop unrolling. Further, optimization in the DEFACTO system is to decrease latency and increase the effective bandwidth to external memory. While many locality optimizations used in conventional systems can be applied (e.g., loop permutation, tiling, unrolling, scalar replacement), new optimizations are possible such as parallelization of memory accesses and customization of memory access protocols [DHP+03]. It uses a hardware structure called *tapped delay line*.
and **scalar replacement** for data reuse. Furthermore, a parameterizable memory controller can be generated which is optimized for external memory access. In addition, a custom data layout is undertaken by the compiler where the array data is distributed across multiple memory banks according to the chosen loop unrolling. In DEFACTO, compiler and synthesis tools are integrated via design space exploration where an interface to Synopsys Behavioral Compiler was developed to automatically calculate balance (where data is fetched from memory at approximately the same rate as it is consumed by the computation) for a particular optimized loop nest design [Dut11].

**Streamroller** [KFM06]: Streamroller is an automated system for designing stylized accelerator pipelines from streaming applications. Streamroller synthesizes a highly customized pipeline that minimizes hardware cost while meeting a user-prescribed performance level. Figure 3.11 shows the overview of streamroller synthesis system. The system takes as input the application written in ‘C’, expressed as a set of communicating kernels. Performance and design constraints, such as overall throughput of the pipeline, clock period and memory bandwidth, are also specified. The front end of the system performs data de-
3. Compiler Framework

Figure 3.11: Overview of streamroller synthesis system, source [KFM06].

Dependency analysis on the application to derive the loop graph, which is a representation of the communication structure between the kernels. The system synthesizes an accelerator pipeline with minimum cost to meet the performance constraints. The pipeline consists of a number of loop accelerators (LAs) to execute the kernels in the application and ping-pong memory buffers for communicating values. A unique aspect of the system is the utilization of multifunction loop accelerators to enable multiple pipeline stages to time multiplex the hardware for a single pipeline stage. This approach sacrifices performance as kernel execution is sequentialized, but greatly increases the ability to share hardware in the design and thus drive down the overall cost. In each LA, the resulting data path is similar to a VLIW template, which is generated by a compiler based on the SUIF compiler [HS02] front end and Trimaran compiler [The05] as back end. The accelerators are connected over SRAM buffers. The basic building blocks of the applications are filters, and queues implemented on SRAM buffers.

SPARK [GDGN03]: The SPARK synthesis framework is a modular and extensible high-level synthesis system that takes a behavioral description in C as input and produces synthesizable register-transfer level VHDL code. SPARK uses parallelizing compiler technology consisting of a transformation toolbox to enhance instruction-level parallelism and re-instruments it for high-level synthesis by incorporating ideas of mutual exclusivity of operations, resource sharing and hardware cost models. The transformations toolbox contains a data dependency extraction pass, parallelizing code motion techniques, dynamic renaming of variables, the basic operations of software pipelining and some supporting compiler passes such as copy and constant propagation and dead code elimination. After scheduling, SPARK does control synthesis and optimization. Control synthesis generates a finite state machine controller and also does resource binding. The back end of the SPARK system then generates synthesizable RTL and hence, the SPARK system integrates into the standard synthesis design flow. This framework was particularly targeted to mapping control intensive func-
3.3 Related Work

Bluespec [Blu14]: Bluespec performs for hardware synthesis from an operation centric description, where the behavior of a system is described as a collection of atomic operations in the form of rules. Such a rule is typically defined by a predicate condition and an effect on the state of the system. In an execution, a rule reads the state of the system in one step, and if enabled, the effect of the rule updates the state in the same step. If several rules are enabled at the same time, any one of the rules can be non-deterministically selected to update the state in one step, and afterwards, a new step begins with the updated state. The atomicity requirement simplifies the task of hardware description by permitting the designer to formulate each rule as if the rest of the system is static [HA00]. Bluespec uses a subset of system verilog, called Bluespec System Verilog. BSV is explicitly parallel and based on atomic transactions, the best-known tool for specifying complex concurrent behavior, which is so prevalent in SoCs. BSV has a precise notion of mapping atomic transactions to synchronous logic, and can do so in a refinable way. BSV is based on synthesizable high-level types and higher-order programming facilities more often found in advanced programming languages, delivering succinctness, parameterization, reuse and control adaptivity [CM08].

There exists a number of commercial high-level synthesis tools from industry such as CatapultC from Mentor Graphics [Men08], Forte Synthesizer [For09], C-to-silicon from Cadence [Cad09], Nios-II C-to-hardware acceleration compiler C2H from Altera [Alt08], Autopilot [ZFJ+08], Impulse Codeveloper [Imp09]. An excellent and exhaustive overview of the state-of-the-art HLS tools is given in [CM08]. In [Dut11], comparison of few HLS tools based on the language, model of computation, loop transformations, hardware acceleration generation, support for data reuse as well as communicating loops, interface synthesis for SoC integration, and design space exploration is given. Our PARO [HRDT08, SHTT14] high-level synthesis framework for accelerator synthesis and exploration is able to support all the above aspects in its entirety. A survey focusing on efforts that map computations written in imperative programming languages to reconfigurable architectures which identifies the main compilation and synthesis techniques is presented in [CDW10]. Finally, in [MS09] history and evolution of HLS from research to industry adoption is presented.

3.3.2 Mapping Approaches for Coarse-Grained and Embedded Multi-Processor Arrays

Often, there is only a fine line between on-chip processor arrays and coarse-grained reconfigurable architectures (CGRA) since the provided functionality...
is similar. Even though there exists only few research work that deals with the compilation to CGRAs, we want to distinguish ours from it. CGRAs has the advantage of less configuration time and configuration data compared to the FPGA-based accelerators [KHKT06b]. The existing mapping approaches to CGRAs have their origin in methods for high-level synthesis (HLS). An approach for efficient circuit implementations is parallelizing high-level synthesis [GGDN04]. This approach takes into account the effect of code transformations on control, interconnect, and area costs rather than only latency. However, this method is not able to consider array architectures. In contrast, our HLS framework PARO [HRDT08, SHTT14] performs aggressive parallelization and generates deeply pipelined array architectures. The authors in [VNK+03] describe a compiler framework to analyze SA-C programs, perform optimizations, and automatically map the application onto the MorphoSys architecture [SLL+00], a row-parallel or column-parallel SIMD (Single Instruction-stream Multiple Data-stream) architecture. This approach is limited since the order of the synthesis is predefined by the loop order and no data dependencies between iterations are allowed. Another approach for mapping loop programs onto CGRAs is presented by Dutt and others in [LCD03]. Outstanding in their compilation flow is the target architecture, the Dynamically Reconfigurable ALU Array (DRAA), a generic reconfigurable architecture template, which can represent a wide range of CGRAs. The mapping technique itself is based on loop pipelining and partitioning of a data flow graph into clusters, which can be placed on a line of the array. However, all the aforementioned mapping approaches follow a ‘traditional’ compilation approach where first transformations such as loop unrolling are performed, and subsequently the intermediate representation in form of a control/data flow graph is mapped by using placement and routing algorithms. Another well known CGRA is ADRES [MVVL04], an architecture that tightly couples a coarse-grained reconfigurable matrix and a VLIW processor. ADRES has a compiler framework [MVV+02], which tries to identify pipelinnable loops, which can be accelerated by the reconfigurable matrix. Then, the compilation and scheduling process is two fold one for the VLIW processor and another for reconfigurable matrix. Afterwards, the two parts of scheduled code are put together, ready for being executed by the ADRES architecture. Both schedulers take advantage of shared resources to maximize the performance. The compiler employs modulo scheduling, a placement and routing similar as in FPGAs, and a technique to remove prologue and epilogue code.

Unique to our approach is that we use an exact resource-constraint scheduler [Han09], and—thanks to using loop tiling [IT88, TTZ97, DHT06b] in the polyhedron model [FL11]—the placement and routing [HDT06] is implicitly given—i.e., for free and much more regular. In the context of processor arrays and the polyhedron model, only some early and very limited work (no tiling,
3.3 Related Work

no software pipelining [Rau94] within the processing elements, etc.) for program-
mable processor networks such as transputers [LBI91] or coarse-grained
reconfigurable arrays [HDT06] exists.

The STHORM [MBF12] platform is a high performance accelerator from
STMicroelectronics which is organized in multiple clusters, connected via a high
performance NoC. It can be programmed by either using a proprietary pro-
gramming model or OpenCL [Mun12]. However, OpenCL has some limitations,
which are already discussed in Section 3.1.1. The PACT-XPP architecture can
be programmed at low-level using the Native Mapping Language (NML) or at
high-level in C language [Tec06]. When using the NML language, the resource
allocation and configuration of the PEs are described by the programmer in
the code. However, using the C language has an advantage that a compiler is
available to semi-automatically generate the configuration bitstream from the C
code. Another interesting architecture is Reconfigurable Instruction Cell Array
(RICA) an heterogeneous array of instruction cells with an Harvard architec-
ture with program data memory and program memory, reconfigurable rate con-
troller to control delay in data paths [KNM08]. Its framework consists of an
image-processing C library along with tools for compilation (C/C++ frontend),
implementation and emulation. For testing, profiling and debugging, both a
functional emulator and cycle accurate simulator are available. In general these
architectures came in two types: arrays of arithmetic logic units (ALUs), with
a central controller, and arrays of small CPUs, tightly connected and generally
intended to communicate in a very synchronized manner. Serious program-
m challenges remain with this kind of architecture because it requires two
distinct modes of programming, one for the CPUs themselves and one for the
interconnect between the CPUs. A single programming language would have
to be able to not only partition the workload but also comprehend the mem-
ory locality, which is severe in a mesh-based architecture. The multi-core and
many core architectures from the companies like Chameleon, PACT, Picochip,
Morpho, Morphics, and Quicksilver arose in the late 1990s and mostly died in
the fallout of the tech bubble burst [KAG09]. They suffered from a lack of
production quality tooling and no clear programming model. Therefore, pro-
gramming models and compilation frameworks are utmost important for these
kind of architectures.

The processors used in our TCPA are VLIW processors [Fis83]. The general
perception of a VLIW processor is that it may cause a huge increase in code
size by not always employing all functional units (NOP operations). There-
fore, numerous software pipelining techniques have been proposed, which allow
a processor to execute more than one iteration at a time, and thus, optimize
the resource utilization while respecting (loop-carried) data dependencies. In
addition to these software pipelining concepts, we augment our accelerator ar-
chitecture with an external controller (see Figure 2.8), which generates branch
control signals, based on the loop bounds, to orchestrate the correct execution of a nested loop program in order to support zero-overhead loop (ZOL). Thanks to the controller, we are able to map arbitrarily nested loops onto such VLIW architectures with ZOL—in contrast to the work in [UWW+99] and [KRJ+09]. Our approach for mapping loops onto programmable processor arrays is different from [KRJ+09] in the following ways: We first strictly separate the control flow from the data flow. Secondly, not only the control of the innermost loop but also that of all outer loops is captured by a controller. Hence, we do not perform any static control flow related operations such as loop bound initialization, loop incrementing/decrementing or loop bound checking in the processor itself as in [KRJ+09] but outside by one single (global) controller. As a result, only branching is performed based on the control signals that are generated and come from the controller. This separation is unique to our approach. Due to software pipelining, instructions from different iterations may overlap, resulting in increased code size but our controller and the proposed approach can significantly reduce the code size.

3.4 Summary

In this chapter, we discussed the fundamentals of our compiler framework for mapping loop nests to processor arrays, which was based on the polyhedron model, for mapping nested loop programs onto TCPAs. Subsequently, front end of the compiler was briefly presented where a nested loop program was described at first using the PAULA language, which follows the semantics of SSA. Few high-level transformations which were reviewed to match hardware constraints as well as partitioning strategies, and the problem of resource constrained scheduling of loop programs. Later, several important problems in the back end that surface during the code and interconnect generation for processors in the array, designing and generating the configuration data for both the global controller as well as address generators were highlighted. Finally, related work on HLS tools and mapping approaches for coarse-grain reconfigurable architectures as well as embedded multi-processor arrays was presented.
4

Automatic Code and Interconnect Generation

In the previous chapters, we highlighted the importance of heterogeneous Multi-
Processor System-on-Chips (MPSoCs) based on programmable accelerators for
providing high energy efficiency while meeting the orthogonal objectives of high-
performance, and flexibility. In this context, we introduced the architecture of
Tightly Coupled Processor Array (TCPA) and our compiler framework for mapping
computationally intensive nested loop programs onto it. In this chapter,
we present our compiler framework with emphasis on automatically generating
efficiently the assembly level programs for all processors in the array and
generating the configuration of interconnections among the processors.

4.1 Motivation

In a TCPA design, required number of Processing Elements (PEs) can be instan-
tiated as per the needs of set of applications by selecting an appropriate number
of rows and columns of the array at synthesis time. Alternatively, it might be
dimensioned to fit into a single Field-Programmable Gate Array (FPGA). Here,
a plausible question is: How can we generate correct and compact code for each
of the PE of an allocated PE array? Moreover, generating individual code for
each PE in an array would be a tedious and time consuming task. Assuming an
array size of $10 \times 10$, there are 100 PEs for which one should generate code. Here,
again a plausible question is: How can we generate these codes in a scalable and
efficient manner, i.e., irrespective of the array size? Finally, the PEs in an ar-
ray have an instruction memory of small size in the order of only few kilobytes
to reduce the area and power consumption. Therefore, to allow the mapping
of even very complex and lengthy loop kernels, the generated programs must
be compact and should also be independent of the problem size. In particular,
changing the problem size involves only modifying the loop bounds in the given
nested loop program. This should not influence the size of the generated code.
Moreover, we assume loop program is already scheduled considering the resource
constraints (e.g., PEs and functional units within a PE) and the Global Latency (GL) is already calculated in the front end of the compiler framework. In the back end, the code generation approach should preserve the computed schedule, i.e., it should not introduce any additional delay or modify the schedule when, e.g., inserting some extra instructions. Here, a valid question is: How can we generate a code that is compact and independent of problem size while preserving a given schedule of data flow operations? For a given TCPA architecture, each PE is encapsulated in an interconnect wrapper, and these wrappers are already connected. However, the connections between each PE and its interconnect wrapper have to be generated or configured depending on the application. This whole interconnection structure is responsible for the propagation of data and control signals within the processor array. For instance, consider a program running on a PE, it might read/write data from/to neighboring processors via the input/output ports connected to its interconnect wrapper. In this context, a credible question is: How can we generate such an interconnect structure for all the PEs in the array in a scalable manner automatically from a given scheduled loop program?

In this chapter, we propose an automatic code and interconnect generation approach that will address all the “How can we?” questions raised above. The main contributions of this chapter can be summarized as follows:

- A code generation approach for a scheduled loop program that
  - Preserves the pre-computed schedule of data flow operations.
  - Generates the compact code that is as fast as the fully equivalent unrolled (across all loop levels) code.
  - Scalable, i.e., independent of both the array as well as the problem size.

- An interconnect generation approach that is scalable and configures the interconnect among the processors with appropriate delay registers from a given scheduled loop program.

The rest of the Chapter is organized as follows: In order to generate machine codes from a scheduled loop program, first of all, variables in the loop program need to be assigned to registers present in the processor architecture. Therefore, in Section 4.2, an important step called register allocation also often referred to as register binding is discussed in detail whereby variables in the loop program are assigned different register types is discussed in detail. In Section 4.3, an overview of the code generation approach is given, where the whole processor array is divided into so-called processor classes. All processors in a class will receive the same program but with a delayed control flow. Therefore, it is sufficient to generate a program for a candidate element from a processor class and
load it to all the processors in the same processor class. Moreover, the iteration space assigned to a candidate element is further divided into so-called *program blocks*. For this purpose, the program execution is represented by a control flow graph of these program blocks. Later on, programs for each processor class are generated using this control flow graph. In Section 4.4, a procedure for automatically extracting processor classes as well as extracting program blocks for each processor class from a given scheduled loop program is discussed. In Section 4.5, the required algorithms for creating a program block control flow graph are presented. In Section 4.6, a three-step approach for compact code generation is proposed where repeating pattern of instructions are identified and branching or looping instructions are introduced without introducing any delay, i.e., the given schedule is kept intact. The functionality of the global controller which is responsible for orchestrating the programs running on the processor is described in Section 4.7. Furthermore, how this global controller approach makes the generated code independent of the iteration space size is explained in this section. The interconnect architecture for data signals may consist of dedicated links with delay shift registers for each inter-processor dependency where the length of the shift registers depends on the computed schedule and can be determined easily. The control signals hailing from a global controller are responsible for orchestrating the programs running on the PEs. These control signals are propagated from the global controller to all PEs through PEs in a linear staggered fashion with appropriate delays. These delays also depend on the computed schedule and can be determined automatically. Furthermore, for the interconnect generation, we divide the processor array into so-called *interconnect classes* where all processors in a class will have the same type of interconnections. These classes will also help to reduce the configuration time since all processors in a class can be selected by an appropriate multicast signature and programmed at once. In Section 4.8, we present this process of automatically synthesizing interconnections among processors with appropriate delays. Results of the code generation for selected case studies are discussed along with comparison of our code generation approach with other compiler frameworks in Section 4.9. Finally, the summary of this chapter is presented in Section 4.10.

### 4.2 Register Allocation

Register allocation is the process of assigning registers available in the architecture, such as general purpose (RD) registers, input (ID)/output (OD) registers, and feedback data (FD) registers, see Figure 2.2, to the variables existent in the algorithm. Here, our assumption is that enough registers are available to support a desired user-specified throughput (Iteration Interval (II)). Otherwise, scheduling will fail since register spilling, which is available in the case of stan-
4. Automatic Code and Interconnect Generation

dard processors, is not supported. In order to do the register allocation, the life time of a variable is of interest which is defined below. Once a schedule is available, the Reduced Dependence Graph (RDG) $G = (V, E, D)$ is decorated with the Life Time (LT) of each variable.

**Definition 4.1 (Life time interval & Life time)** The life time interval of a variable $v_i \in V$ is defined as the time interval beginning at the production of the variable and ending at the time where no more operations use (consume) this variable. The life time interval is denoted by $LT_i = [LT_{lb}^i, LT_{ub}^i]$ where $LT_{lb}^i$ and $LT_{ub}^i$ denote the production time and the latest consumption time of the variable, respectively. The life time of a variable $v_i$ is the time difference between the $LT_{ub}^i$ and $LT_{lb}^i$, denoted by $LT^i = LT_{ub}^i - LT_{lb}^i$.

Given a RDG $G = (V, E, D)$ with $e \in E$ be an edge and $d_e$ be the associated dependence vector. In [Han09], the author computes three different types of edges and life times in dependence on the space-time mapping $Q$ and the value of $d_e$. The set of outgoing edges $F(v_i)$ of a node $v_i$ can be decomposed into three disjoint sets, $F(v_i) = F_0(v_i) \cup F_d(v_i) \cup F_a(v_i)$. That is, edge $e$ can be classified to exactly one of the three sets $F(v_i) = F_0(v_i)$, $F_d(v_i)$, and $F_a(v_i)$ as follows.

- **Type1** [Han09]: If the data dependency vector $d_e$ satisfies $Q \cdot d_e = 0$ and $d_e = 0$, $e$ belongs to set $F_0(v_i)$. In terms of hardware, this set represents local memory within a processor to store intermediate results within one iteration interval. This set translates to $RD$ type registers in the hardware. The corresponding life time is denoted by $LT_0(v_i)$ and for further details such as how to compute it, we refer to [Han09].

- **Type2** [Han09]: If the data dependency vector $d_e$ satisfies $Q \cdot d_e = 0$ and $d_e \neq 0$, $e$ belongs to set $F_d(v_i)$. This set also denotes the internal storage within a PE but since $d_e \neq 0$, the corresponding data need to be stored for more than one iteration interval. In terms of hardware, this set represents local memory within a processor to store intermediate results for more than one iteration interval. This set translates to $FD$ type registers in the hardware. The corresponding life time is denoted by $LT_d(v_i)$ and it denotes the depth of the $FD$ register [Han09].

- **Type3** [Han09]: If the data dependency vector $d_e$ satisfies $Q \cdot d_e \neq 0$, $e$ belongs to set $F_a(v_i)$. From the hardware perspective, this set denotes delay registers between PEs. Each such dependency translates to an extra $OD$ register for one PE and $ID$ register for the other PE in the hardware. The corresponding life time is denoted by $LT_a(v_i)$ and it denotes the delay register length between processors.
Once the dependency type or the life time type is identified as mentioned above, the pseudo code in Algorithm 4.1 gives a systematic approach of register allocation in our compiler framework. It is to be noted that we further assume that each node $v_i$ can have only one type of life time, i.e., its outgoing edges are fall into exactly one of the three disjoint sets mentioned above. At first in Algorithm 4.1, primary input and output variables in the given input PAULA program are assigned to $ID_x$ and $OD_x$ registers, respectively. In a PAULA program, input/output variables are specified with the keyword `in/out`. Therefore, they can be identified easily and assigned input/output registers. In the second step, depending on the type of the life time of the variables, corresponding registers in the hardware are assigned as mentioned in Algorithm 4.1.

In the following paragraphs, we often refer to Type1 life time as simply life time which is subsequently used in the general purpose registers ($RD_x$) allocation. The main idea of the general purpose register allocation and its requirements are explained with two example PAULA programs given in Listing 4.1 and Listing 4.2. It is assumed that these two programs are mapped onto a single processor only and scheduled with enough number of resources, which result in a schedule vector of $\lambda = 1$ and $II = 1$ cycle for both programs.

**Listing 4.1: Register Allocation - example PAULA program.**

```plaintext
1: par ( i >= 0 and i <= 5 )
2: {
3:   a[i] = A[i] if (i == 0);
4:   a[i] = a[i-1] if (i > 0);
5:   b[i] = a[i] * 7;
6:   C[i] = a[i] + b[i];
7: }
```

Figure 4.1(a) shows the scheduled RDG for the example of Listing 4.1 where the life times of the variables are annotated back to the RDG after scheduling. For variable ‘b’, a life time of 1 cycle is obtained since it is used immediately in the next cycle to calculate the value of output variable ‘C’. Please note that all the operations are single cycle operations and the $II = 1$ cycle for this example. Therefore, for each iteration, variable ‘b’ lives only for one cycle. It is also implied that for variable ‘b’, only one general purpose register is required. Whereas, if we consider variable ‘a’, it is used in computing the value of variable ‘b’ in the next cycle and variable ‘C’ in the cycle after the next. Therefore, the life time $LT_a = 2$ is obtained for each computed instance of variable ‘a’. As $II = 1$, $\left\lceil \frac{LT_a}{II} \right\rceil = 2$. This also implies, two general purpose registers are required, which can be used repeatedly (one at a time) in two consecutive iterations. The general purpose register allocation for this example is shown in Figure 4.1(b).
Algorithm 4.1: Overview of register allocation for a given scheduled loop
program given in form of its RDG.

\textbf{Input}: Scheduled loop program RDG.

\textbf{Output}: Register allocation for all variables of the loop program, back
annotated RDG with register allocation.

// considering a partitioned, scheduled, resource allocated loop program given by
its RDG

\begin{verbatim}
1 begin
  2   C_ID = 0, C_OD = 0, C_RD = 0, C_FD = 0
  3   foreach v_i \in G do
  4     if (type(v_i) == PI) // PI: primary input
  5       Assign IDx register for v_i, C_ID+=1.
  6     end
  7     if (type(v_i) == PO) // PO: primary output
  8       Assign ODx register for v_i, C_OD+=1.
  9     end
 10     if (out_edges(v_i) \in F_c(v_i)) then
 11       Assign unique \( \left\lceil \frac{LT_i}{II} \right\rceil \) RDx registers for v_i, C_RD+=\( \left\lceil \frac{LT_i}{II} \right\rceil \).
 12     end
 13     if (out_edges(v_i) \in F_a(v_i)) then
 14       Assign a FDx register for v_i, C_FD+=1.
 15     end
 16     if (out_edges(v_i) \in F_d(v_i)) then
 17       Assign both IDx/ODx for v_i, C_ID+=1, C_OD+=1.
 18     end
 19 end
\end{verbatim}

For the same example, if the \( II = 2 \) cycles, variable ‘a’ needs only one general
purpose register.

Listing 4.2: Register Allocation - Example PAULA program.

\begin{verbatim}
1 : 
2 par(i >= 0 and i <= 5)
3 {
  4   a[i] = A[i] if(i == 0);
  5   a[i] = a[i-1] if(i > 0);
  6   b[i] = a[i] * 7;
\end{verbatim}
4.2 Register Allocation

(a) Scheduled RDG showing the LT and LTI of each variable.
(b) Register allocation where variable ‘a’ requires two and variable ‘b’ requires one general purpose register.

Figure 4.1: A scheduled RDG and computed register allocation for Listing 4.1

7 \[ c[i] = b[i] \times 8; \]
8 \[ D[i] = c[i] + a[i]; \]
9 \[ E[i] = c[i] + b[i]; \]
10 }

Similarly, for the example in Listing 4.2, Figure 4.2(a) shows the scheduled RDG where the lifetimes of each variable have been annotated back to the RDG after scheduling. For the same reasons as mentioned above, variable ‘a’ requires three general purpose registers since \( \lceil \frac{LT}{II} \rceil = 3 \). These three general purpose registers one at a time will be used repeatedly in three consecutive iterations. Variable ‘b’ require only two registers which can be used repeatedly (one at a time) in two consecutive iterations since the LT of the variable is 2 cycles. In contrast, variable ‘c’ requires only one register which is reused in all the iterations since the LT of the variable is just 1 cycle. The computed general purpose register allocation for this example is shown in Figure 4.2(b). An observation of the register allocation is that after 6 cycles, the same registers are used repeatedly again for the next 6 cycles. In other words, general purpose register allocation will repeat for each set of 6 cycles (see Figure 4.2(b)). This period after which the general purpose register allocation repeats is called Super Iteration Interval (SII) and defined as follows.

**Theorem 4.1 (Super iteration interval)** is defined as the minimum number of iteration intervals required for the register usage allocation for all variables of a given loop to repeat.
4. Automatic Code and Interconnect Generation

(a) A scheduled RDG for Listing 4.2 showing local schedule and LTs of the variables.

(b) Computed register allocation for Listing 4.2 showing that variables might need multiple general purpose registers.

Figure 4.2: A scheduled RDG and computed register allocation for Listing 4.2.

Lemma 4.1 Let \( N = \{ n_i \mid 1 \leq i \leq |V| \} \), where \( n_i = \left\lceil \frac{LT_i}{II} \right\rceil \) denotes the multiple of iteration intervals required for a variable \( v_i \in V \) to repeat its register assignment schedule. Then, \( SII \) is the least common multiple (LCM) of all \( n_i \in N \).

**Proof.** In a periodic schedule, \( SII \) must be obviously a multiple of each \( n_i \). Hence, the LCM returns the smallest number of iteration intervals until the register assignment may repeat. \( \square \)

In Figure 4.1(b), \( SII = LCM(2,1) = 2 \), since for variable ‘a’ the register usage repeats after two iteration intervals and for variable ‘b’ after one iteration interval. Whereas for the example in Figure 4.2(b), \( SII = LCM(3,2,1) = 6 \). For the general purpose register allocation, \( LEFT\_EDGE \) (see Appendix A), a well-known algorithm for coloring interval graphs, can be used which was proposed by Hashimoto and Stevens for channel routing [HS71, She95]. The same algorithm can be used with a little modification as given in Algorithm 4.2. In this algorithm, if the life time of a variable is greater than the iteration interval, extra intervals are created so that the standard left edge algorithm colors these intervals with distinct colors, denoting that multiple registers are needed for a variable. For the two examples shown in Listing 4.1 and Listing 4.2, applying Algorithm 4.2 results exactly in the register allocation as shown in Figure 4.1(b) and Figure 4.2(b), respectively.

For our FIR filter specification introduced in Listing 3.5, complete register allocation with emphasis on the allocation of ID, OD, and FD registers is explained below.
Algorithm 4.2: General purpose register allocation.

**Input**: Reduced dependence graph \( G = (V, E, D) \), with \( V \) denoting the set of nodes and \(|V|\) denoting the number of nodes in the RDG, 
\( \text{LTI} \) is the list of lifetime intervals where each set is initialized with \( \text{LTI}[i] = \{[LTI_i^{lb}, LTI_i^{ub}]\} \), \( \text{LT} \) is the list of lifetimes with \( LT[i] = LTI_i^{ub} - LTI_i^{lb} \ \forall 1 \leq i \leq |V| \), and \( \Pi \) is the iteration interval.

**Output**: Register assignment to nodes in RDG.

```plaintext
1 Register Allocation (LTI, LT, II)
2 begin
3   foreach \( v_i \in V \) do
4     \( M_i = \left\lceil \frac{LT_i}{\Pi} \right\rceil \)
5     if \( (M_i > 1) \) then
6       for \( (j = 1; j < M_i; j++) \) do
7         \( \text{LTI}[i] = \text{LTI}[i] \cup \{ [LTI_i^{lb} + j \times \Pi, LTI_i^{ub} + j \times \Pi] \} \)
8     end
9   end
10  end

Apply use the standard left edge algorithm \( \text{LEFT}_\text{-EDGE}(\Pi) \) (Appendix A) with \( I = \bigcup_{i=1,\ldots,|V|} \text{LTI}[i] \).
```

**Example 4.1 (FIR-filter continued) [TBHL15*]** The input variables ‘a_in’, ‘u_in’ and output variable ‘y_out’ in the FIR filter program are assigned to input registers ID0, ID1 and OD0, respectively. In the following, for each variable, register allocation is discussed with proper reasoning.

**Variable ‘a’**: is used to propagate the input ‘a_in’ (filter coefficients) for reuse. However, it must be assigned not only to the general purpose register RD0, but additionally to a feedback register FD0, OD1, and ID4 for the following reasons:

- **RD0**: For variable ‘a’ only one register RD0 is required since its LT is 1 cycle same as the \( \Pi \). \( \Pi = 1 \) implies that a new instance of all RDG operations will start executing at each cycle. In the first 8 iterations and thus cycles, a new coefficient is read from the input ID0 (variable ‘a_in’, see also Figure 4.3). This is copied (assigned) to variable ‘a’. Here, register RD0 is used to propagate variable ‘a’ for PE-internal use within an iteration. To move the data from the input, a Functional Unit (FU) \( \text{DPU0}^7 \) is used.

---

\(^7\text{A DPU is a functional unit that just moves data from a source to a destination register.}\)
Figure 4.3: Partitioned iteration space of the FIR filter showing: distinct processors, intra-processor, and inter-processor dependencies. Input/Output registers corresponding to inter-processor dependencies are also shown.
4.2 Register Allocation

- FD0: Feedback data registers are used to store variables that are cyclically reused. They are used to implement loop-carried dependencies (dependencies spanning multiple iterations). For example, in case of processor PE(0,0) when computing iteration \((i_1 j_1)^T = (1 0)^T\), the value of 'a' is the same as 'a' at iteration \((i_1 j_1)^T = (0 0)^T\) (due to localization for efficient data reuse, see also Figure 4.3); hence, when computing iteration \((i_1 j_1)^T = (0 0)^T\), it has to be stored in a feedback data register (using DPU1 FU) so that it may be used again at the cycle where \((i_1 j_1)^T = (1 0)^T, (2 0)^T, \ldots (7 0)^T\) within the same processor. The length (adjustable delay in clock cycles) of the feedback data register may be determined easily by multiplying the intra-tile schedule vector \(\lambda_1\) and the dependence vector \(d\) associated with variable 'a'. In our example, \(\lambda_1 \cdot d_a = (8 1) \cdot (1 0) = 8\) (cf. Figure 4.3). That is, eight coefficients of the FIR filter are read successively from an input register ID0, and are then reused cyclically by reading from feedback register FD0.

- OD1/ID4: Due to partitioning, dependencies are split among processors. Variable 'a' dependency \(d_a = (1 0)\) is divided during partitioning into an intra-processor and an inter-processor dependency to propagate the value within processors and among processors, see Figure 4.3. Here, the inter-processor dependency translates into an output and an input register. For instance, PE(1,0) must receive the filter coefficients 'a' from PE(0,0). Hence, in order to support this communication, variable 'a' is also assigned to an output register OD1 that is physically connected to input register ID4 of the PE(1,0) processor, see also Figure 4.3.

Variable ‘u’: is used to propagate the input ‘u_in’ value. It is assigned to the registers RD1, FD1, OD2, ID2, ID5, and ID6 for similar reasons as explained for variable ‘a’. To move the data into RD1 functional unit DPU2 and to move data into FD1 functional unit DPU3 are used. It is worth noticing that during partitioning, the variable ‘u’ dependency \(d_u = (1 1)\) is divided into intra-processor and inter-processor dependencies to propagate the ‘u’ value within processors and among processors, see Figure 4.3. There are three inter-processor dependencies but only one output port and three distinct input ports are created as shown in Figure 4.3. In the real hardware also, from an output port multiple wires can be taken but for an input port there should be only one input driver. During the output register allocation this behavior is considered to optimize the output registers.

Variable ‘x’: is assigned to register RD2 only since its LT is only 1 cycle. It is used internally within a processor so no copies to extra input/output ports are necessary.

Variable ‘y’: is used to store the partially accumulated sum of the filter output. It is assigned to the registers FD2, OD3, and ID3 for similar reasons as
explained for both variable a and u. Variable y dependency \( d_y = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \) is divided into intra-processor and inter-processor dependencies to propagate the y value within processors and among processors, see Figure 4.3. Here, the inter-processor dependency translates into an output (OD3) and input (ID3) register.

Once the register allocation is done, all the register allocation results are back annotated to the RDG so that they can be used later in the code generation. An exemplary RDG with all such information is shown in Figure 4.4 for the FIR filter example. From this tagged RDG, one could select finally the instructions for each node by looking at the incoming source nodes, i.e., assigned registers, and the functional as well as register binding for that node. For instance, the instruction for node x in Figure 4.4 would be MUL0 MUL RD2 RD1 RD0 in which MUL0 is the used functional unit, MUL is the mnemonic. RD2 is the destination operand, RD1 is source operand 1, and RD0 is source operand 2. In the next section, we present an overview of the code generation approach and steps involved in it. In the subsequent sections, each of this step is elaborated.

4.3 Code Generation: Overview

In this section, we briefly outline our approach of code generation for all PEs as shown in Figure 4.5 [BHT14*]. We start with a loop program—after scheduling
4.3 Code Generation: Overview

and FU as well as register assignment—to be executed on a TCPA. The processor space $\mathcal{P}$ of this loop program is divided into so-called processor classes such that all processors that belong to the same class obtain the same program but may differ only by a delayed start of execution. Therefore, first of all, we need to determine the set of processor classes, and then, generate compact code for a candidate element of each processor class while satisfying a given optimized overall schedule $\lambda$. In Section 4.4.1, we propose an algorithm to derive automatically these processor classes from a given loop algorithmic description. For the example shown in Figure 3.7, a total of nine processor classes is extracted as shown in Figure 4.6 by different color shades under the PE names and they are also listed in Table 4.1.

Due to the condition spaces belonging to different statements of the given loop program, the program assigned to a single processor of a processor class may have to execute different code at each iteration assigned to it. Such code blocks are called program blocks ($PB$) in the following. After determining the set of processor classes, we therefore need to identify program blocks (see Figure 4.5) for each processor class. Nine such program blocks $PB_0$, $PB_1$, $PB_2$ ... $PB_8$
Table 4.1: Different processor classes and their corresponding processor elements for the example of Figure 4.6.

<table>
<thead>
<tr>
<th>Processor Class (PC)</th>
<th>Processing Elements (indices)</th>
<th>Candidate Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{ (0 0)^T }</td>
<td>(0 0)^T</td>
</tr>
<tr>
<td>1</td>
<td>{ (0 1)^T, (0 2)^T }</td>
<td>(0 1)^T</td>
</tr>
<tr>
<td>2</td>
<td>{ (0 3)^T }</td>
<td>(0 3)^T</td>
</tr>
<tr>
<td>3</td>
<td>{ (1 0)^T, (2 0)^T }</td>
<td>(1 0)^T</td>
</tr>
<tr>
<td>4</td>
<td>{ (1 1)^T, (1 2)^T, (2 1)^T, (2 2)^T }</td>
<td>(1 1)^T</td>
</tr>
<tr>
<td>5</td>
<td>{ (1 3)^T, (2 3)^T }</td>
<td>(1 3)^T</td>
</tr>
<tr>
<td>6</td>
<td>{ (3 0)^T }</td>
<td>(3 0)^T</td>
</tr>
<tr>
<td>7</td>
<td>{ (3 1)^T, (3 2)^T }</td>
<td>(3 1)^T</td>
</tr>
<tr>
<td>8</td>
<td>{ (3 3)^T }</td>
<td>(3 3)^T</td>
</tr>
</tbody>
</table>

have been identified in Figure 4.7(a) for processor class ‘0’ of our running FIR filter example. For the same reason, the RDG can now also be split into as many as identified program blocks, as shown in Figure 4.8, one for each program block. In Section 4.4.2, we propose an algorithm to derive these program blocks automatically from a given loop algorithmic description. After identifying the different program blocks and corresponding RDGs for each program block of a given processor class, the assembly instructions can be generated for all nodes in the program block RDG. For instance, for the FIR filter example, Figure 4.8 shows the assembly instruction sequences for each program block.

Finally, for a candidate element belonging to a processor class, flat yet lengthy code could be generated from such code sequences of each iteration by packing contiguously different instructions according to the schedule and software pipelining. This code would require $LL + |I| \cdot II - 1$ Very Long Instruction Word (VLIW) words—making the generated code not only dependent on the problem size (iteration space size $I$) but also might not fit into the tiny available instruction memories of the PEs. Therefore, we are interested in generating
code that is compact by introducing as much zero-overhead looping as possible for repetitively executed assembly code sequences.

In order to generate compact code for each processor class (see Figure 4.5), the program blocks are represented in a much more compact form called Control Flow Graph ($G_{CF}$), see Figure 4.7(b). Each node in a $G_{CF}$ denotes a PB. In the $G_{CF}$, the numbers annotated to the edges indicate how many times the execution is transferred from one node (PB) to the other node in the overall traversal of
4. Automatic Code and Interconnect Generation

(a) Processor class ‘0’ and its candidate element $PE(0,0)$ showing different program blocks. Numbers annotated to the iterations denote their start times.

(b) Control flow graph $G_{CF}$ of FIR filter showing the execution order of the program blocks, i.e., scanning order of the iteration space assigned to $PE(0,0)$.

Figure 4.7: Processor class iteration space of the running FIR filter example showing program blocks and its control flow graph.
4.3 Code Generation: Overview

the iteration space. For a node with a self-edge, the weight associated to the self-edge denotes the number of iterations that get executed consecutively whenever the execution enters that node. Consider, for instance, \( PB_4 \) in Figure 4.7(a) where the weight of the self-edge is 30 and of the other incoming edge is 6, resulting in \( 30 + 6 = 36 \) iterations which is exactly equals to the total number of iterations in \( PB_4 \). Such a graph \( G_{CF} \) can be generated from the given iteration space \( I \) and the schedule \( \lambda \) computed in Section 3.2.4. In Section 4.5, we propose an algorithm to generate a program block control flow graph \( G_{CF} \). We propose a three step approach for the generation of compact code using the program block control flow graph \( G_{CF} \) and program block assembly codes in Section 4.6.

To summarize so far, Algorithm 4.3 [BHT14*] lists the steps involved in generating compact code for a partitioned and scheduled loop program.

(a) RDG of \( PB_0 \), program block iteration space \( I_{PB_0} = \{i_1 = 0 \land j_1 = 0\} \), and its assembly code. Since the Local Latency (LL) = 3, a program block consists of 3 VLIW instructions denoted as 0, 1, and 2, which may contain NOPs as well. Different unique VLIW instructions are labeled as A, B, and C.

(b) RDG of \( PB_1 \), program block iteration space \( I_{PB_1} = \{i_1 = 0 \land 1 \leq j_1 \leq 6\} \), and assembly code. Different unique VLIW instructions are labeled as D, B, and E.

(c) RDG of \( PB_2 \), program block iteration space \( I_{PB_2} = \{i_1 = 0 \land j_1 = 7\} \), and assembly code. Different unique VLIW instructions are labeled as F, B, and G.
Figure 4.8: Program blocks of processor class ‘0’ of the FIR filter of Figure 4.7(b).
Algorithm 4.3: Summary of the code generation approach for a given scheduled loop program.

**Input**: Scheduled loop program  
**Output**: Compact code for each processor class (PC)

/* considering a partitioned, scheduled loop program after FU and register assignment */

1 begin
2 (1) Identify the set of processor classes PC.
3 (2) foreach PC ∈ PC do
4 (a) Identify the set of program blocks PB.
5 (b) Create a program block control flow graph G_{CF}.
6 (c) foreach PB ∈ PB do
7 Create RDG of program block PB.
8 Generate assembly code for each program block.
9 end
10 (d) Compact code using the program block control flow graph G_{CF} and program block’s assembly codes.
11 end
12 end

4.4 Processor Classes and Program Blocks

Given a Locally Sequential Globally Parallel (LSGP)-partitioned Piecewise Regular Algorithm (PRA), we explain the notion of processor classes and program blocks in this section and subsequently propose algorithms to derive the set of processor classes PC and program blocks PB automatically.

Let us consider the generic space-time mapping for a LSGP-partitioned loop program given by the following equation:

\[
\begin{pmatrix}
p \\
t
\end{pmatrix} = \begin{pmatrix}
0 & E \\
\lambda_1 & \lambda_2
\end{pmatrix} \cdot \begin{pmatrix}
I_1 \\
I_2
\end{pmatrix}, \text{ where } I_1 \in \mathcal{I}_1, I_2 \in \mathcal{I}_2
\] (4.1)

As mentioned earlier (see Listing 4.3), the condition space \( \mathcal{C}_i \) of each equation \( S_i \) of a given \( n \)-dimensional PRA gets transformed into a \( 2n \)-dimensional condition space during partitioning. Let \( \{ \mathcal{C}_i' \subseteq \mathbb{Z}^{2n} \mid 1 \leq i \leq K' \} \) be the set of condition spaces after partitioning (\( K' \) denotes the new total number of equations after partitioning). In the rest of the dissertation, we assume that the following conditions hold for each \( \mathcal{C}_i' \).

Orthogonality and shape of condition space constraints:

\[
\mathcal{C}_i' = \left\{ \begin{pmatrix}
I_1 \\
I_2
\end{pmatrix} \in \mathbb{Z}^{2n} \mid \begin{pmatrix}
A_{i,1} & 0 \\
0 & A_{i,2}
\end{pmatrix} \begin{pmatrix}
I_1 \\
I_2
\end{pmatrix} \geq \begin{pmatrix}
b_{i,1} \\
b_{i,2}
\end{pmatrix} \right\}
\] (4.2)
4. Automatic Code and Interconnect Generation

where

\[ I_1 = \{ (i_{1,1}, i_{1,2}, \ldots, i_{1,m_1}) \in \mathbb{Z}^{m_1} | \begin{align*}
&lb_{1,1} \leq i_{1,1} \leq ub_{1,1} \\
&lb_{1,2} \leq i_{1,2} \leq ub_{1,2} \\
&\vdots \\
&lb_{1,m_1} \leq i_{1,m_1} \leq ub_{1,m_1} \end{align*} \}, \]

\[ I_2 = \{ (i_{2,1}, i_{2,2}, \ldots, i_{2,m_2}) \in \mathbb{Z}^{m_2} | \begin{align*}
&lb_{2,1} \leq i_{2,1} \leq ub_{2,1} \\
&lb_{2,2} \leq i_{2,2} \leq ub_{2,2} \\
&\vdots \\
&lb_{2,m_2} \leq i_{2,m_2} \leq ub_{2,m_2} \end{align*} \}, \]

and \( m_1 + m_2 = 2n \)

That means, each condition space \( C'_i \) can be decomposed into two independent spaces. The first spaces \( C'_i(I_1) = \{ I_1 \in \mathbb{Z}^{m_1} | A_{i,1} \cdot I_1 \geq b_{i,1} \}, 1 \leq i \leq K' \) represent only conditions within a tile and the second spaces \( C'_i(I_2) = \{ I_2 \in \mathbb{Z}^{m_2} | A_{i,2} \cdot I_2 \geq b_{i,2} \}, 1 \leq i \leq K' \) impose only restrictions on the space of tile origins, i.e., on the processor space. Equation (4.1) implies \( p = I_2 \), that is, \( I_2 \) is equivalent to the processor space \( P \) or, each tile origin is assigned to exactly one processor that will obtain the number (coordinates) of this tile origin. Now, we have conditions that cause different sets of equations and hence programs to be executed in different regions of the resulting processor array. We will call these implicitly defined sets processor classes. Decomposing the processor space \( P = I_2 \) based on the condition spaces that depend only on \( p = I_2 \), i.e., \( C'_i(I_2) = \{ I_2 \in \mathbb{Z}^{m_2} | A_{i,2} \cdot I_2 \geq b_{i,2} \}, 1 \leq i \leq K' \), will give us the set of disjoint but unique processor spaces \( I_{PC_j}, 0 \leq j < m \) associated with \( m \) different processor classes. Here, \( m \) is therefore the amount of unique distinct intersections of one or more subsets of the \( C'_i(I_2) \). Subsequently, for each processor class \( PC_j \) (\( 0 \leq j < m \)), we intersect each condition space \( C'_i(I_2) \), \( 1 \leq i \leq K' \) with \( I_{PC_j} \). For all candidates with non-empty intersection, we intersect the corresponding condition spaces \( C'_i(I_1) \) to find the sets of iterations with equal equations to be computed at a processor belonging to class \( PC_j \). These will be called program blocks (PB) of processor class \( PC_j \).

We explain the concept of processor classes and their generation in a more intuitive way by looking at the equations defining ‘\( y \)’ and ‘\( y_{\text{out}} \)’ of the partitioned FIR filter given in the following Listing 4.3.

<table>
<thead>
<tr>
<th>Listing 4.3: Excerpt of the partitioned FIR filter program.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
4.4 Processor Classes and Program Blocks

Due to partitioning, the equations defining ‘y’ and ‘y_out’ are split into multiple equations as shown in Figure 4.9, which also results in extra input and output ports. For variable ‘y’, an extra output port OD3 and an extra input port ID3 are created, as can be seen also in Figure 4.3. If we would write assembly code for each processor, it might look as shown in Figure 4.9 (bottom) and it can be easily understood that different I/O caused to guard ‘y’ and ‘y_out’ will divide the entire processor space into three mutually exclusive sets, i.e., \{PE(x, 0), PE(x, 1), PE(x, 2), PE(x, 3)\} where \(x \in \{0, 1, 2, 3\}\). If we consider each equation at a time, the existing set of processor classes will get divided further into a total of nine processor classes for FIR filter example. After extraction of all processor classes, it is sufficient to consider the problem of code generation for each processor class’s candidate element and load the same program to all the processors belonging to the same processor class.

4.4.1 Determination of Processor Classes

Algorithm 4.4 \[BHT14^*\] proposes a systematic way of decomposing a given processor space \(\mathcal{P}\), based on the condition spaces that depend only on \(I_2\) (i.e., condition spaces \(\mathcal{C}'_i(I_2)\), \(1 \leq i \leq K'\)), into \(m\) processor classes.

At the beginning, Algorithm 4.4 selects a half-space \[Zie95, MS71\] from the first condition space \((\mathcal{C}'_1(I_2))\) and divides \(\mathcal{P}\) into at most two disjoint spaces. These two spaces are subsequently decomposed by considering the next half-space and so on, i.e., this procedure is iteratively continued until all half-spaces
Before partitioning:

\( S_7: y[i, j] = x[i, j] \) if \((j==0)\);

\( S_8: y[i, j] = y[i, j-1]+x[i, j] \) if \((j>0 \text{ and } j<N-1); // N=32\)

\( S_9: y_{\text{out}}[i, j] = y[i, j-1]+x[i, j] \) if \((j==N-1)\);

After partitioning:

\( S_{10}: y[i_1, j_1, i_2, j_2] = x[i_1, i_2, j_2] \) if \((j_1==0 \text{ and } j_2==0)\);

\( S_{11}: y[i_1, j_1, i_2, j_2] = y[i_1, j_1-1, i_2, j_2] + x[i_1, i_2, j_2] \) if \((j_1>0 \text{ and } j_1<N-1); // N_1=8\)

\( S_{12}: y[i_1, j_1, i_2, j_2] = y[i_1, j_1-1, i_2, j_2] + x[i_1, i_2, j_2] \) if \((j_1=N_1-1 \text{ and } j_1<N_2-1); // N_2=4\)

\( S_{13}: y[i_1, j_1, i_2, j_2] = y[i_1, j_1+N_1-1, i_2, j_2-1] + x[i_1, i_2, j_2] \) if \((j_1==0 \text{ and } j_2>0)\);

\( S_{14}: y_{\text{out}}[i_1, j_1, i_2, j_2] = y[i_1, j_1-1, i_2, j_2] + x[i_1, i_2, j_2] \) if \((j_1=N_1-1 \text{ and } j_2=N_2-1)\);

**Figure 4.9:** Decomposition of processor space into mutually exclusive sets, i.e., \( \{PE(x, 0)\}, \{PE(x, 1), PE(x, 2)\}, \{PE(x, 1)\} \) where, \( x \in \{0, 1, 2, 3\} \) based on variable ‘y’ and output variable ‘ŷ_out’.
4.4 Processor Classes and Program Blocks

Algorithm 4.4: Decomposition of the processor space $\mathcal{P} = \mathcal{I}_2$ into $m$ disjoint processor classes.

**Input:** $\mathcal{P}$ and $\mathcal{C}_2 = \{\mathcal{C}'_1(I_2), \mathcal{C}'_2(I_2), \ldots, \mathcal{C}'_{K'}(I_2)\}$

**Output:** Disjoint decomposition of $\mathcal{P}$, i.e., set of subspaces each denoting a processor class ($PC$): $I_{PC} = \{I_{PC_0}, I_{PC_2}, \ldots, I_{PC_{m-1}}\}$

1. $I_{PC} = \{\mathcal{P}\}$
2. foreach $\mathcal{C}'_i(I_2) \in \mathcal{C}_2$ do
   3. $I_{tmp} = \emptyset$
   4. foreach $\mathcal{D} \in I_{PC}$ do
      5. foreach half-space $\mathcal{C}'_i^{ij}(I_2) \in \mathcal{C}'_i(I_2)$ do
         6. $\mathcal{D} = \mathcal{D} \cap \mathcal{C}'_i^{ij}(I_2)$
         7. if $\mathcal{D} \neq \emptyset$ then
            8. $I_{tmp} = I_{tmp} \cup \{\mathcal{D}\}$
            end
         9. $\mathcal{D} = \mathcal{D} \cap \mathcal{C}'_i^{ij}(I_2)$
        10. if $\mathcal{D} \neq \emptyset$ then
            11. $I_{tmp} = I_{tmp} \cup \{\mathcal{D}\}$
            end
      end
   12. $I_{PC} = I_{tmp}$
   end
end

of all condition spaces have been considered. More specifically, consider a condition space $\mathcal{C}'_i(I_2)$, $1 \leq i \leq K'$:

$$\mathcal{C}'_i(I_2) = \{I_2 \in \mathbb{Z}^n \mid A_{i,2} \cdot I_2 \geq b_{i,2}\}$$

where $A_{i,2} \in \mathbb{Z}^{l_{i,2} \times n}$

$$= \left\{I_2 \in \mathbb{Z}^n \mid \begin{pmatrix} A_{i,2}^1 \\ A_{i,2}^2 \\ \vdots \\ A_{i,2}^{l_{i,2}} \end{pmatrix} I_2 \geq \begin{pmatrix} b_{i,2}^1 \\ b_{i,2}^2 \\ \vdots \\ b_{i,2}^{l_{i,2}} \end{pmatrix} \right\}$$

(4.3)

with $\mathcal{C}'_i^{ij}(I_2) = \{I_2 \in \mathbb{Z}^n \mid A_{i,2}^{ij} \cdot I_2 \geq b_{i,2}^{ij}\}$, $1 \leq j \leq l_{i,2}$, that is, $\mathcal{C}'_i^{ij}(I_2)$ denotes the $j^{th}$ half-space of $\mathcal{C}'_i(I_2)$. $\mathcal{C}'_i^{ij}$ is defined by $\mathcal{C}'_i^{ij}(I_2) = \{I_2 \in \mathbb{Z}^n \mid A_{i,2}^{ij} \cdot I_2 < b_{i,2}^{ij}\}$.

The step-by-step application of this decomposition algorithm to the FIR filter example is visualized in Figure 4.10.
4. Automatic Code and Interconnect Generation

**Figure 4.10:** A step-by-step decomposition of the processor space into processor classes for FIR filter example.
4.4 Processor Classes and Program Blocks

For the space-time mapping according to Equation (3.14), the corresponding 4×4 processor array (|P| = 16) target is shown in Figure 4.10(a). Algorithm 4.4 initializes the set of iteration spaces associated to processor classes by P, i.e., I_{PC} contains one single set including all 16 processor indices initially. For the first condition spaces after partitioning for ‘a_in’ and ‘a’, the processor space is divided into three disjoint sets of processors, each denoting a processor class as shown in Figure 4.10(b). Similarly, considering the condition spaces after partitioning for ‘y’ and ‘y_out’, these three disjoint sets are further divided into a total of nine disjoint sets as shown in Figure 4.10(c). Finally, considering condition spaces for ‘u_in’ and ‘u’, there is no further increase in the number of these disjoint sets.

4.4.2 Determination of Program Blocks

Now, for each processor class, the decomposition of the iteration space I_1 considering all condition spaces C'_i(I_1) that are defined at this processor class leads to the mutually exclusive iteration spaces asso ciated with the program blocks. Algorithms 4.5 and 4.6—which is very similar to the processor class decomposition algorithm Algorithm 4.4—describes the procedure that determines the disjoint program blocks.

For the FIR filter example and processor class PC_0, Algorithm 4.5 finds the following nine mutually exclusive iteration spaces (program blocks):

\[
\begin{align*}
I_{PB_0} &= \{i_1 = 0 \land j_1 = 0\} \\
I_{PB_1} &= \{i_1 = 0 \land 1 \leq j_1 \leq 6\} \\
I_{PB_2} &= \{i_1 = 0 \land j_1 = 7\} \\
I_{PB_3} &= \{1 \leq i_1 \leq 6 \land j_1 = 0\} \\
I_{PB_4} &= \{1 \leq i_1 \leq 6 \land 1 \leq j_1 \leq 6\} \\
I_{PB_5} &= \{1 \leq i_1 \leq 6 \land j_1 = 7\} \\
I_{PB_6} &= \{i_1 = 7 \land j_1 = 0\} \\
I_{PB_7} &= \{i_1 = 7 \land 1 \leq j_1 \leq 6\} \\
I_{PB_8} &= \{i_1 = 7 \land j_1 = 7\}
\end{align*}
\]

The nine distinct program blocks per PE(p_1 = i_2; p_2 = j_2) have been shown already in Figure 4.6.

In the following, we consider PE(0,0) and only variable ‘y’ after partitioning (see Figure 4.9) in order to explain the notion and generation of program blocks. The condition spaces associated to variable ‘y’ divide the iteration space of PE(0,0) into three mutually exclusive sub iteration spaces (program blocks) as shown in Figure 4.11(a), where each one is denoted by a different color. For PE(0,0), the reduced dependency graph can be derived from the initial RDG.
4. Automatic Code and Interconnect Generation

Figure 4.11: (a) Iteration space assigned to PE(0,0) and different program blocks considering only the variable ‘y’. Each differently colored rectangular box corresponds to a unique set of equations defined at these iterations. (b) RDG corresponding to processor PE(0,0). (c) Source operands for node ‘y’ in program block where \( j_1 = 0 \). (d) Source operands for node ‘y’ in program block where \( 1 \leq j_1 \leq 6 \). (e) Source operands for node ‘y’ in program block where \( j_1 = 7 \).
**Algorithm 4.5:** Decomposition of iteration space $\mathcal{I}_1$ for a given processor class $PC$ into program blocks.

**Input:** $\mathcal{I}_1$, $C_1 = \{C'_1(I_1), C'_2(I_1), \ldots, C'_{K'}(I_1)\}$, $C_2 = \{C'_1(I_2), C'_2(I_2), \ldots, C'_{K'}(I_2)\}$, and $PC$

**Output:** Disjoint decomposition of $\mathcal{I}_1$, i.e., set of subspaces each denoting a program block (PB): $\mathcal{I}_{PB} = \{\mathcal{I}_{PB_0}, \mathcal{I}_{PB_1}, \ldots\}$

/* Determine condition spaces that are relevant for a given program class $PC$ with corresponding processor space $\mathcal{I}_{PC}$ */

1. \( C_{cand} = \emptyset \)
2. for \( i = 1 \) to \( K' \) do
3.     if \( (\mathcal{I}_{PC} \cap C'_i(I_2)) \neq \emptyset \) then
4.         \( C_{cand} = C_{cand} \cup \{C'_i(I_1)\} \)
5.     end
6. end
7. \( \mathcal{I}_{PB} = \{\mathcal{I}_1\} \)
8. foreach \( C'_i(I_1) \in C_{cand} \) do
9.     \( I_{tmp} = \emptyset \)
10. foreach \( D \in \mathcal{I}_{PB} \) do
11.     foreach half-space \( C'_i(I_1) \in C'_i(I_1) \) do
12.         \( \mathcal{D} = D \cap C'_i(I_1) \)
13.         if \( \mathcal{D} \neq \emptyset \) then
14.             \( I_{tmp} = I_{tmp} \cup \{\mathcal{D}\} \)
15.         end
16.         \( D = D \cap C'_i(I_1) \)
17.         if \( D \neq \emptyset \) then
18.             \( I_{tmp} = I_{tmp} \cup \{D\} \)
19.         end
20.     end
21. end
22. \( \mathcal{I}_{PB} = I_{tmp} \)
23. end

by hiding unnecessary nodes as shown in Figure 4.11(b). If we look at node ‘y’ in this RDG, it assigned to the functional unit ADD0, it has three incoming edges, one from node ‘x’ as a first operand and both CONST 0 as well as node ‘y’ itself as a second operand. Since the adder can add only two inputs at a time, the two sources of the second operand should be mutually exclusive as shown in Figures 4.11(c) and 4.11(d). Furthermore, node ‘y’, due to partitioning splits between processors resulting in writing the results of addition to an output
Algorithm 4.6: Decomposition of iteration space $I_1$ for a given processor class $PC$ into program blocks.

**Input:** $I_1$, $C_1 = \{C'_1(I_1), C'_2(I_1), \ldots, C'_{K'}(I_1)\}$, $C_2 = \{C'_1(I_2), C'_2(I_2), \ldots, C'_{K'}(I_2)\}$, and $PC$

**Output:** Disjoint decomposition of $I_1$, i.e., set of subspaces each denoting a program block (PB): $I_{PB} = \{I_{PB_0}, I_{PB_1}, \ldots\}$

/* Determine condition spaces that are relevant for a given program class $PC$ with corresponding processor space $I_{PC}$ */

```
C_cand = ∅
for $i = 1$ to $K'$ do
  if ($I_{PC} \cap C'_i(I_2)) \neq ∅$ then
    $C_cand = C_cand \cup \{C'_i(I_1)\}$
  end
end
$I_{PB} = \{I_1\}$
for each $C'_i(I_1) \in C_cand$ do
  $I_{tmp} = ∅$
  foreach $D \in I_{PB}$ do
    $\overline{D} = D \cap C'_i(I_1)$ where $\overline{C'_i(I_1)} = I_1 - I_1 \cap C'_i(I_1)$
    if $\overline{D} \neq ∅$ then
      $I_{tmp} = I_{tmp} \cup \{\overline{D}\}$
    end
    $D = D \cap C'_i(I_1)$
    if $D \neq ∅$ then
      $I_{tmp} = I_{tmp} \cup \{D\}$
    end
  end
$I_{PB} = I_{tmp}$
```

port as shown in Figure 4.11(e). Figures 4.11(c) to 4.11(e) correspond to three mutual exclusive program blocks concerning the variable ‘y’ as explained above. The differing assembly codes to compute node ‘y’ for the three program blocks can also be seen in three different cases in Figure 4.11. If we consider only the condition space of variable ‘y’, it divides the iteration space into three program blocks. If we consider all the equations or variables, we end up in the program blocks as shown in Figure 4.7(a).

In Algorithm 4.5, we use the half-spaces of the intra-tile condition spaces in decomposing the iteration space assigned to a processor class. Whereas in Algorithm 4.6, we use directly the intra-tile condition spaces themselves. This has
4.4 Processor Classes and Program Blocks

(a) Resulting program blocks according to Algorithm 4.5 for Listing 4.4.

(b) Resulting program blocks according to Algorithm 4.6 for Listing 4.4.

Figure 4.12: Program blocks for Listing 4.4.

an impact on the number of program blocks created. For Listing 4.4, applying both algorithms result in the different number of program blocks as shown in Figures 4.12(a) and 4.12(b) assuming it is mapped on to a single PE.

Listing 4.4: An example PALUA program.

1  program example_different_programblocks
2  {
3    variable A 2 in signed integer<16>;
4    variable Y 2 out signed integer<16>;
5    variable a 2 signed integer<16>;
6    par (i >= 0 and i <= 15 and j >= 0 and j <= 15)
7    {
8      a[i, j] = A[i, j]+1 if(j==0 and i>=0 and i<=3);
9      a[i, j] = A[i, j]+2 if(j==0 and i>=4 and i<=7);
10     a[i, j] = A[i, j]+3 if(j==0 and i>=8 and i<=11);
11     a[i, j] = A[i, j]+4 if(j==0 and i>=12 and i<=15);
12     a[i, j] = a[i, j-1]*2 if(j>=1 and j<=10 and i>=0 and i<=3);
13     a[i, j] = a[i, j-1]*3 if(j>=1 and j<=10 and i>=4 and i<=7);
14     a[i, j] = a[i, j-1]*4 if(j>=1 and j<=10 and i>=8 and i<=11);
15     a[i, j] = a[i, j-1]*5 if(j>=1 and j<=10 and i>=12 and i<=15);
16     a[i, j] = a[i, j-1]+10 if(j>=11 and j<=15);
17     Y[i, j] = a[i, j]+5 if(j>=11 and j<=15);
18    }
19  }
4. Automatic Code and Interconnect Generation

Obviously, it is clear from the example that the Algorithm 4.5 may not result in an optimum number of program blocks whereas Algorithm 4.6 will result in the optimum number of program blocks. If we use the program blocks generated using Algorithm 4.5 in the code generation, the generated code might not be optimum in its code size compared to the code generated using program blocks according to Algorithm 4.6. However, the program block control flow graphs generated in both cases have different characteristics, which will be discussed in the next sections.

Once the program blocks are created, instructions for these program blocks may be generated as shown in Figure 4.8 for FIR filter of Listing 3.5. If we generate such codes for each processor, few processors will have the same code resulting in processor classes. Therefore, we identify processor classes first and for each processor class its set of program blocks. Next, we explain how to create a program block control flow graph of these program blocks which is subsequently used for compaction of the code sequences of each program block of each processor class and as well as in the compact code generation of a processor class.

4.5 Program Block Control Flow Graph

From the discussions above, it is understood that the iteration space assigned to a processor class candidate element is divided into mutually exclusive iteration space blocks called program blocks. If $I_{PC}$ denotes the iteration space of a processor class candidate element and $I_{PBi}, 0 \leq i < m$, denote the iteration space of a program block, then

$$I_{PC} = I_{PB0} \cup I_{PB1} \cup \ldots \cup I_{PBm-1}$$  \hspace{1cm} (4.4)

where $I_{PBi} \cap I_{PBj} = \emptyset$ for all $0 \leq i, j \leq m, i \neq j$.

Executing a given loop program means that executing the iterations assigned to a processor according to the given schedule. It also means that executing the program blocks in a particular order according to the same schedule since the iteration space is divided into program blocks. Intuitively, by using a given schedule and program blocks, a program block control flow graph $G_{CF}$ can be generated where each node corresponds to a program block. The number annotated to each edge indicates how many times the execution is transferred from the predecessor to the successor program block in the overall traversal of the iteration space assigned to the respective processor. We want to exploit this fact by creating a control flow graph that captures consecutive program block execution without code replication for each block. For a given schedule, a scanning order imposed by intra-tile schedule determines how the iterations are scanned.
4.5 Program Block Control Flow Graph

within a tile or processor. This scanning order within a tile is captured in a path stride matrix defined below.

**Definition 4.2 (Path stride matrix [Han09])** The path strides of an n-dimensional iteration space \( I = (i_1, i_2, \ldots, i_n) \), given by an n-dimensional parallelootope, are represented by a matrix \( S \in \mathbb{Z}^{n \times n} \) consisting of \( n \) (column) vectors \( s_i \in \mathbb{Z}^n \) and given as \( S = (s_1, s_2, \ldots, s_n) \). Here, \( s_i \) are vectors, which are added to the iteration point to get the next iteration. The step \( s_1 \) is added to the iteration point \( I \) until it crosses the iteration space boundary in direction \( i_1 \). The stride \( s_2 \) takes it back into the iteration space. Likewise, stride \( s_j \) is added to iteration point \( I \) if it crosses the iteration space in direction \( i_j \). Such conditions determining the selection of strides are called stride condition spaces, denoted by \( I_{SCS} \).

**Example 4.2** For the FIR filter shown in Figure 4.6, the path stride matrix is given by

\[
S = \begin{pmatrix}
0 & 1 \\
1 & -7 \\
\end{pmatrix}
\]

where each tile is scanned using \( S \), i.e., iterations belonging to a processor are scanned or executed as per the order determined by \( S \). Considering \( PE(0,0) \), \( 0 \leq j_1 \leq 6 \) irrespective of \( i_1 \) every iteration is scanned using \( s_1 = (0, 1)^T \). If \( j_1 = 7 \) irrespective of \( i_1 \), scanning direction is changed using \( s_2 = (1, -7)^T \).

So, these two vectors describe how the whole tile iteration space is scanned and executed, respectively. It is to be noted that there may exist multiple scanning orders for a tile that must not violate any data dependencies. In [Han09], the author describes a method to find the best path stride matrix, which basically minimizes the total execution time of the tile.

In the following, we present an Algorithm 4.7, see also [TBHL15*], to generate such a control flow graph from a given loop description using program blocks and the determined schedule, i.e., the path stride matrix.

The main idea behind this algorithm can be summarized as follows: In the proposed algorithm, we start from a \( PB \), add each time a column vector \( s \) from \( S \) to the iteration space \( IS \) of the \( PB \), and intersect it with all other iteration spaces of corresponding program blocks, including its own. If the intersection is empty (\( \emptyset \)), no transition between the program blocks occurs, else, there is a transition between them. If the intersecting program blocks are different, the number of points in the intersection space denote the number of times the execution is transferred from one program block to the other. If the intersecting program blocks are the same, then we are interested in knowing the number of consecutively executed iterations whenever the execution enters a program block, which is given by an *Ehrhart Polynomial* (EP) [Eug62]. The process of intersection has to be repeated for all program blocks as described in Algorithm 4.7 which will result in a program block control flow graph \( G_{CF} \).
Algorithm 4.7: Generating a Control Flow Graph (GCF)

Input : Set of program blocks, path stride matrix \( \vec{S} \)
Output: Control flow graph \( G_{CF} \)

1. \( G_{CF}(V, E) = \emptyset \) /* Graph initialization */

2. foreach \( PB_i \in \{ PB_1, PB_2, \ldots, PB_m \} \) do
3.   foreach \( s \in S \) do
4.     \( I_{img} = \{ I \in I_{PB_i} \mid I + \vec{s} \} \)
5.       /* \( I_{PB_i} \) is the iteration space of \( PB_i \) */
6.       /* \( I_{img} \) is a translated iteration space */
7.      foreach \( PB_k \in \{ PB_1, PB_2, \ldots, PB_m \} \) do
8.        if \( (I_{img} \cap I_{PB_k} \neq \emptyset) \) then
9.          if \( v_i \notin V \) then \( V = V \cup v_i \)
10.         if \( (PB_i == PB_k) \) then
11.            create a self edge \( e \) on \( v_i \), i.e., \( E = E \cup (v_i, v_i) \)
12.            add Ehrhart polynomial (EP) corresponding to \( PB_i \) as weight to the edge \( e \) /* in case of rectangular iteration spaces, EP is a constant number */
13.          end
14.        else
15.          if \( v_k \notin V \) then \( V = V \cup v_k \)
16.            create an edge \( e \) from \( v_i \to v_k \), i.e., \( E = E \cup (v_i, v_k) \)
17.            add weight \( |I_{img} \cap I_{PB_k}| \) to the edge \( e \)
18.          end
19.      end
20.  end
21. end

For the FIR filter of Figure 4.6, applying the Algorithm 4.7 results in the program block control flow graph as shown in Figure 4.7(b). For Listing 4.4, the control flow graphs generated are shown in Figure 4.13. One can clearly see that the \( G_{CF} \) in Figure 4.13(b) has less nodes (program blocks), however, the branching possibilities for \( PB2 \) are increased.

In the following, we briefly explain the relevance of Ehrhart Polynomials and their use in our compilation approach. As aforementioned, our compilation approach is based on the polyhedron model—the iteration space essentially represents a polytope and iterations correspond to integral points in the polytope. A path stride matrix determines the execution order of these integral points
in the polytope without violating any data dependencies. For the FIR filter example shown in Figure 4.7(a), iterations are scanned in row major order, i.e., in $j_1$ direction first and then in $i_1$ direction. Start times of the iterations are also shown in Figure 4.7(a). In other words, the entire iteration space is sliced with respect to the $i_1$ direction. During this scanning, iterations in the different program blocks are executed and in turn they also get sliced with respect to $i_1$. In the polyhedron model, this slicing of program blocks can be achieved by converting the polytope denoted by it into a parameterized polytope considering $i_1$ as a parameter. Here, the number of integral points in each slice is given by a Ehrhart Polynomial. If program blocks iteration spaces are rectangular in shape, each slice of its rectangular space will have the same number of iterations given by Ehrhart Polynomial which is just a constant number. In case of non-rectangular program blocks, each slice of it may have distinct number of iterations or integral points where Ehrhart Polynomial is a pseudo polynomial.
denoting this distinct number of iterations in each slice. In this dissertation, we focus only on rectangular program blocks and for further details on Ehrhart Polynomials and parameterized polytopes, we refer to Appendix B.

Once the program block control flow graph $G_{CF}$ is generated, it can be used for the compact code generation. However, in few cases, loops in the control flow graph have to be unrolled partially to handle the proper cyclic allocation of registers to variables while traversing through the nodes, i.e., program blocks in the control flow graph. In the next section, we explain this control flow graph unrolling process with an intuitive example and present algorithms for unrolling of a control flow graph.

### 4.5.1 Partial Control Flow Graph Unrolling

In Section 4.2, while explaining the register allocation, it is understood that if a variable life time is longer than the iteration interval, it needs multiple registers where distinct registers are used in distinct iterations for the same variable. We also introduced the concept of the SII, which denotes the minimum number of iteration intervals required for all variable’s register allocation to repeat cyclically. This cyclic allocation of registers has to be respected while traversing the control flow graph, respectively while generating the code from it. For Listing 4.1, a complete RDG with resource, and register allocation including the LTs is shown in Figure 4.1(a) and register allocation in form of a Gantt chart for the same listing is shown in Figure 4.1(b). For the same listing, distinct program blocks are shown in Figure 4.14(a) and the control flow graph in Figure 4.14(b). The control flow or execution alternates between $PB_0$ and $PB_1$ since an iteration either belongs to $PB_0$ or $PB_1$. Whenever the execution enters $PB_0$, only one iteration is executed, whereas 6 iterations are executed consecutively in the case of $PB_1$. Execution is transferred from $PB_0 \rightarrow PB_1$ 6 times and from $PB_1 \rightarrow PB_0$ 5 times until the whole iteration space is scanned (see Figure 4.14(a) and Figure 4.14(b)). Figure 4.14(c) and Figure 4.14(d) show the RDGs as well as assembly codes corresponding to $PB_0$ and $PB_1$, respectively. Therefore, after identifying the different program blocks and corresponding RDGs for each program block of a given loop program, the assembly instructions can be generated for all nodes in the RDG. For instance, Figure 4.14(c) shows the two assembly instruction sequences, denoted as $PB_0_{\xi_0}$ and $PB_0_{\xi_1}$ for $PB_0$. Similarly, Figure 4.14(d) shows the two assembly instruction sequences, denoted as $PB_1_{\xi_0}$ and $PB_1_{\xi_1}$ for $PB_1$. The reason for requiring these two sequences is that the SII is 2. Now, if the control flow graph is used in the code generation, for each program block the two code sequences are used in an alternating fashion and which one to use out of these two code sequences depends on the current iteration that is being executed, which has to be tracked always. This alternating use of assembly code sequences is due to the fact that
4.5 Program Block Control Flow Graph

(a) Iteration space corresponding to Listing 4.1 showing different program blocks and their iteration spaces. All iterations belong to a tile which is executed on a single processor.

(b) Control flow graph $G_{CF}$ for the same Listing 4.1 and its transformed control flow graph $G_{TCF}$ where the loop from $PB0 \rightarrow PB1$ is unfolded once.

(c) RDG corresponding to $PB0$ and its assembly code sequences.

(d) RDG corresponding to $PB1$ and its assembly code sequences.

**Figure 4.14:** Iteration space, distinct program blocks and their RDGs as well as assembly code sequences, control flow graph $G_{CF}$ and its transformed control flow graph $G_{TCF}$ for Listing 4.1.
in \( PB0 \), there is only one iteration that is executed which uses \( RD0 \) register assignment in its assembly code sequence \( (PB0_0) \) for the first time and then execution jumps to \( PB1 \) where register allocation enforces it to use \( RD2 \) for register assignment in its assembly code sequence \( (PB1_1) \) for the first time for the first iteration. Later on, for the rest of the iterations assembly code sequences containing \( RD0 \) and \( RD2 \) \((PB1_0 \text{ and } PB1_1)\) are used in \( PB1 \) in an alternating way. However, after executing 6 iterations in \( PB1 \) the execution jumps back to \( PB0 \) enforcing it to use \( RD2 \) as register assignment in its assembly code sequence \( (PB0_1) \) for the single iteration that gets executed in it. This enforces \( PB1 \) to use \( RD0 \) as register assignment in its assembly code sequence \( (PB1_0) \) for the first iteration the execution jumps to \( PB1 \). For the rest of the iterations in \( PB1 \), this time \( PB1_1 \) and \( PB1_0 \) are used in an alternating way. In this way, this alternating use of assembly code sequences for \( PB0 \) and \( PB1 \) repeats which has to be tracked always to respect the alternating register assignment. Instead of tracking, we propose to rather unfold the loops in the control flow graph as shown in Figure 4.14(b). This graph repeats the alternate use of the two assembly code sequences for each program block.

Obviously, the unrolling factor depends on both the super iteration interval and the total number of iterations in a loop body\(^8\). Consider a loop and its loop body in \( G_{CF} \), the total number of iterations in the loop body, denoted by \( IS_{LB} \). If \( IS_{LB} \) is not divisible by \( SII \), whenever the execution enters the loop, a different allocation of registers for variables is needed and a different code needs to be generated for each re-entrance of the loop. Intuitively, the maximum number of such different codes per program block is \( SII \). For the \( G_{CF} \) of Listing 4.1, the loop \( PB0 \rightarrow PB1 \) has in total 7 iterations in the loop body, which is not divisible by \( SII \). Therefore, the loop \( PB0 \rightarrow PB1 \) in the \( G_{CF} \) is unrolled once in the transformed control flow graph \( G_{TCF} \), as shown in Figure 4.14(b).

In other words, \( G_{CF} \) may require some partial unrolling of loops to express the cyclic allocation of variables to registers. Transforming \( G_{CF} \) into \( G_{TCF} \) that reflects also the required different codes due to different register assignments is very obvious in this example, but for another example, as shown in Figure 4.15, the transformed control flow graph can be quite complex. Here, \( SII = 3 \). Finally, the graph \( G_{TCF} \) is used for code generation; therefore, we propose a systematic way of transforming a \( G_{CF} \) into a \( G_{TCF} \) in the next section.

**Transforming \( G_{CF} \) into \( G_{TCF} \)**

In the following, we present our transformation of the control flow graph \( G_{CF} \) into \( G_{TCF} \) as shown in Algorithm 4.8 [BHT13*]. Here, we use lot of terms from

---

\(^8\)Here, we refer to the loop and the loop body in the context of control flow graphs [ASU88].

Loop body refers to a sub-graph with all nodes and edges in a loop excluding the back edge. Loop refers to the loop body including the back edge.
Figure 4.15: A complex control flow graph and its transformed control flow graph showing the unfolding of the loop from PB5 → PB6 three times since the SII=3. It is also unfolded three times whenever the loop from PB3 → PB6 is unfolded. It should be noted that loop from PB2 → PB6 does not need to be unfolded since whenever the control flow enters the loop, it may start with the same register assignment. In other words, the total number of iterations in this loop is divisible by SII=3.

count graph theory and for definitions of these terms, we refer to Appendix C. In order to automate this loop unrolling or replication process, first of all, loops in GCf have to be identified. It is also required to know which loop is included inside which loop, i.e., a loop nest tree of GCf.

In the proposed Algorithm 4.8, Steps 1 to 4 deal with this. We proceed with a brief explanation of the algorithm. First, in Step 1, a depth first ordering of the nodes in GCf is determined. In Step 2, the back edges are identified by building the dominator tree [ASU88]. Once the back edges are determined, the loops in the graph are identified. Self-loops in GCf are not considered as we are interested in loops between different program blocks/nodes. In Step 3, we use the approach proposed by Paul Havlak in [Hav97] to build a loop nest tree GLNT, but we consider only reducible loops [Muc97] in our approach, as the control flow graphs, which we are dealing with, are always reducible since removing all back edges results in a directed acyclic graph. In Paul Havlak’s approach, each node in GCf is assigned an attribute loop_type, which can be of self, reducible or nonheader. In Step 4, the nodes in the loop nest tree GLNT are assigned levels. Example loop nest trees for GCf in Figure 4.14(b)
4. **Automatic Code and Interconnect Generation**

and Figure 4.15 are shown in Figure 4.16(a) and Figure 4.16(b), respectively.

**Algorithm 4.8:** Transforming $G_{CF}$ into $G_{TCF}$

<table>
<thead>
<tr>
<th>Input</th>
<th>Control flow graph $G_{CF}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Transformed control flow graph $G_{TCF}$</td>
</tr>
</tbody>
</table>

begin

1. Step 1: Depth first ordering of the nodes of $G_{CF}$
   
   // during this ordering each node is assigned an attribute
   
   dfs_number, ranging from 0 to max_dfsnumber

2. Step 2: Build dominator tree and identify the back edges and natural loops and create a set of back edges and their corresponding loops. In our analysis we ignore the self-loops and self back edges.

3. Step 3: Find the loop nest tree of $G_{CF}$ using Paul Havlak’s approach in [Hav97]; the loop nest tree is also a graph, denoted by $G_{LNT}$.

4. Step 4: assign levels to the nodes in $G_{LNT}$, use topological sorting.

5. Step 5: call Algorithm 4.9

6. Step 6: $G_{TCF} = G_{CF}$

7. for ($j = 0 ; j \leq \text{max}_d \text{fnumber} ; j + +$) do

   8. foreach node $v_i \in G_{CF}$ do

   9.     if (loop_type[$v_i$] is reducible) \& (dfsnumber[$v_i$] = $j$) \& (body_it[$v_i$] % SII \neq 0) then

   10.     (i) remove the back edge in $G_{TCF}$, replicate the loop body SII times, and connect end node of loop body to the start node of newly replicated loop body like a chain.

   11.     (ii) While replicating, index the original loop body as 0, and for the next replicas increment the index by 1

   12.         // maximum number of such replicas are weight[back edge[$v_i$]] + 1 or SII including the original loop body

   13.     (iii) Mark the original loop body end node as “X” and the network/sub-graph that is going out from “X” as $G_X$

   14.     (iv) if weight[back edge[$v_i$]] + 1 > SII, connect last node of the loop body for which the index equal to (weight[back edge[$v_i$]]) % SII, to the original loop body start node. Mark this loop body end node as “Y”

   15.     (v) detach $G_X$ from “X” and attach it to “Y”

end

118
In Step 5, for each loop, the total number of iterations in the loop body ($IS_{LB}$) is calculated using Algorithm 4.9 and stored as an attribute $body_it$ to the node where the loop starts in $G_{CF}$. We also calculate the total number of iterations that is executed in a loop and store it as a $loop_it$ attribute to the same node. $loop_it$ of a node is required to calculate the variable $body_it$ of an outer loop if this node (the loop starting at this node) is included in another outer loop. In Step 6, the $G_{CF}$ is cloned as $G_{TCF}$ initially and the nodes of $G_{CF}$ are processed in an increasing order of their $dfs_number$. Next, we go to each reducible node (the node where a loop starts) and if $body_it$ of the node is not divisible by SII, we make SII copies of the loop body in $G_{TCF}$ as explained in Algorithm 4.8. Eventually, a $G_{TCF}$ is obtained, which is traversed adhering to a schedule to generate the final compact code. In the rest of the dissertation, the words transformed control flow graph and control flow graph are used interchangeably.

Finally, we address the problem of inserting branch instructions in the final code. Here, we need to know the succeeding program blocks of a given program block so that the execution jumps to exactly one of them. In the rest of the dissertation, these succeeding program blocks are also referred to as reachable program blocks, denoted by a set $PB_R$. Furthermore, this transfer of control flow from one program block to the other may take place in a part of the program block’s iteration space, also called transfer condition space, denoted by $I_{TCS}$, which is also required later on in the global controller. Whenever the code corresponding to a program block is executed on a processor and a transfer condition
space is met, the global controller will be responsible to issue the proper control signals so that the jump takes place between correct program blocks. In the next section, we explain how to get such reachable program blocks and their transfer condition spaces from a given program block using their iteration spaces, stride vectors and stride condition spaces.

Algorithm 4.9: Process Loop Nest Tree for $\mathcal{IS}_{\mathcal{LB}}$

<table>
<thead>
<tr>
<th>Input</th>
<th>$G_{CF}, G_{LNT}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>$G_{CF}$</td>
</tr>
</tbody>
</table>

1. \begin{algorithmic}
   \begin{align*}
   &\text{begin} \\
   &\quad \text{for } (j = \text{max}_\text{level} ; j \geq 0 ; j --) \text{ do} \\
   &\quad \quad \text{foreach node } v_i \in G_{LNT} \text{ do} \\
   &\quad \quad \quad \text{if } (\text{level}[v_i] == j) \text{ then} \\
   &\quad \quad \quad \quad \text{node\_it} = 0 \\
   &\quad \quad \quad \quad \text{node\_it} = \text{node\_it} + \text{weight[Selfarc of } v_i] \\
   &\quad \quad \quad \quad \quad \text{// no self arc weight is 1} \\
   &\quad \quad \quad \quad \text{foreach } v_j \in \text{out\_edges}(v_i) \text{ do} \\
   &\quad \quad \quad \quad \quad \text{if } \left( \text{loop\_type}[v_j] == \text{self} \lor \text{loop\_type}[v_j] == \text{nonheader} \right) \text{ then} \\
   &\quad \quad \quad \quad \quad \quad \text{node\_it} = \text{node\_it} + \text{weight[Selfarc of } v_j] \\
   &\quad \quad \quad \quad \quad \quad \text{end} \\
   &\quad \quad \quad \quad \quad \text{else} \\
   &\quad \quad \quad \quad \quad \quad \text{node\_it} = \text{node\_it} + \text{loop\_it}[v_j] \quad \text{// reducible} \\
   &\quad \quad \quad \quad \quad \quad \text{end} \\
   &\quad \quad \text{end} \\
   &\quad \text{if } \text{loop\_type}[v_i] == \text{reducible} \text{ then} \\
   &\quad \quad \text{body\_it}[v_i] = \text{node\_it} \\
   &\quad \quad \text{loop\_it}[v_i] = \text{node\_it} \times (\text{weight[Backedge}[v_i]] + 1) \\
   &\quad \quad \quad \text{// back edge that is not a self arc in } G_{CF} \\
   &\quad \quad \text{end} \\
   &\quad \text{else} \\
   &\quad \quad \quad \text{body\_it}[v_i] = \text{node\_it}; \text{loop\_it}[v_i] = \text{node\_it} \\
   &\quad \quad \text{end} \\
   &\quad \text{end} \\
   &\text{end} \\
   &\text{annotate loop\_it, body\_it values from } G_{LNT} \text{ to } G_{CF} \\
   &\text{end}
   \end{align*}
\end{algorithmic}

120
4.5.2 Reachable Program Blocks, Transfer Condition Spaces, and Control Signals

Given a program block $PB$, we are interested in knowing the program blocks that can be reached from this program block. It is to be noted that a program block may have a self edge which makes itself a reachable program block of its own. Figure 4.17 explains this intuitively where $PB4$ has a self edge and an edge to $PB5$—resulting in two reachable program blocks. In fact, every outgoing edge from a program block is a reachable program block. Thus, the number of outgoing edges determines the number of control signals needed in a branch instruction. In Figure 4.17, the number of outgoing edges of $PB4$ is two. So, at least one control signal is required in deciding a jump either to $PB4$ or $PB5$ when the actual code corresponding to $PB4$ is running on a processor.

In this case, the semantics of the branch instruction would be $BRU \text{ IF } IC0 \ PB5 \ PB4$, which means that if the control signal in register $IC0$ is true (1), the execution will jump to $PB5$ else it will jump to the beginning of the $PB4$ again.

In case of some other program blocks, we might not need any control signal since they may have at most one outgoing edge. In Figure 4.17, for program blocks $PB0$, $PB2$, $PB3$, $PB6$, and $PB8$, there is only one outgoing edge from these program blocks. Therefore, no control signal is required in the issued branch instruction. In this case, the branch instruction would be $BRU \text{ NEXT}$.

An important result from this discussion is that the maximum number of control signals, denoted by $N_{IC}$, required to implement a given program on a TCPA is given by the following equation:

$$N_{IC} = \left\lceil \log_2 \left\{ \max_{1 \leq i \leq K-1} \{NOE_i\} \right\} \right\rceil$$

(4.5)

where $NOE_i, \ 0 \leq i \leq K-1$, denotes the maximum number of outgoing edges of a program block, $PB_i$ in control flow graph. For Listing 4.4, we need two control signals for the control flow graph shown in Figure 4.13(a). Whereas, three control signals are required for the control flow graph shown in Figure 4.13(b). The main point here is that if we use the half-spaces of intra-tile condition spaces in creating the program blocks according to Algorithm 4.5, we might end up in more program blocks but we always find a half-space that separates two program blocks. If we use the Algorithm 4.6 in creating program blocks, we might end up in optimum number of program blocks but the resultant nodes in the control flow graph may have more outgoing edges (branching). The way we create program blocks and construct the program block control flow graph has an impact on the Global Controller (GC) hardware. For instance, in case of $G_{CF}$ of Figure 4.13(a), a branch instruction can be deduced easily by constructing a Binary Decision Diagram (BDD) [And97, FMK91] at a program block and calculating the transfer condition spaces. Whereas in case of $G_{CF}$ of Fig-
4. Automatic Code and Interconnect Generation

Figure 4.17: Refined control flow graph of running FIR filter example shown in Figure 4.7(b) enriched with transfer condition spaces $I_{TCS}$ annotated to the outgoing edges of the program blocks.
Algorithm 4.10: Generating transfer condition spaces $\mathcal{I}_{TCS}$ and control signals to jump from one program block to another.

**Input**: Control flow graph $G_{CF} = (V, E)$, path stride matrix $S$, stride condition space ($\mathcal{I}_{SCS}$) for each stride vector $s_j$

**Output**: Transfer condition spaces on the outgoing edges of each program block in $G_{CF}$

1. **foreach** $v_i \in V$ do
   2. **foreach** $s_j \in \{s_0, s_1, \ldots, s_m\}$ do
      3. $A = \mathcal{I}_{PB_i} \cap \mathcal{I}_{SCS_j}$
      4. if ($A \neq \emptyset$) then
         5. $\mathcal{I}_{img} = A + s_j$
         6. **foreach** $v_k \in V$ do
            7. $B = \mathcal{I}_{img} \cap \mathcal{I}_{PB_k}$
            8. if ($B \neq \emptyset$) then
               9. annotate $B - s_j$ as transfer condition space to the edge $(v_i, v_k)$
            10. end
         11. end
      12. end
   13. end

we are interested in knowing not only which program blocks can be reached from a particular program block but also the exact transfer condition spaces to reach them. As a first step, the stride condition spaces are intersected with the program block iteration spaces to determine which stride vectors are valid for a program block. Later on, a valid stride vector is added to the program block iteration space and the resultant iteration space is intersected with all other program blocks including its own. If this intersection is not empty, there is a transition from this program block to the one with the intersected. Subtracting the stride vector from the resultant iteration space gives exactly the transfer condition space from this program block to the one with the intersected. Finally, this condition space is saved as an edge attribute in $G_{CF}$. This intersecting procedure has to be repeated for all program blocks and for all valid stride vectors of a program block as stated in Algorithm 4.10. For our running example of FIR filter given in Listing 3.5, Figure 4.17 shows the calculated transfer condition spaces at each program block according to Algorithm 4.10.
4. Automatic Code and Interconnect Generation

It is to be noted that for each reachable program block, there exists a distinct transfer condition space. Once the transfer condition spaces are determined, a BDD or MTMDD is constructed at each program block to decode the control signals. This helps in inserting the branch instructions. Later on, the same BDD/MTMDD can be used to generate the configuration data for the global controller. In the following, we explain these two approaches and their pros and cons using examples shown in Figures 4.18 and 4.19.

**BDD Approach:**

**Example 4.3**  Figure 4.18(a) shows a partial control flow graph of Figure 4.13(a), where PB2 has three outgoing edges $PB2 \rightarrow PB2$, $PB2 \rightarrow PB0$, and $PB2 \rightarrow PB3$. Therefore, two control signals are required in the branch instruction and the semantics of the branch instruction would be as follows:

\[
\text{BRU \ IF\ IC1\ IC0\ JMP\ } PB3\ PB0\ PB2\ PB2
\]

Here, depending on the values of IC1 and IC0, the execution jumps to one of the target program blocks in the branch instruction. It is to be noted that the PBx as targets in the branch instructions are just placeholders. In the final code, they are replaced by absolute instruction memory addresses that carry the first instruction of each target program block. For the branch instruction in this example, depending on the control signal values (as shown under the braces) execution jumps to one of the target program blocks. Next, these reachable target program blocks should be encoded systematically as jump targets in the branch instructions—given the transfer condition spaces and the scanning order of the variables. These transfer condition spaces can be automatically determined using Algorithm 4.10 given the program blocks, stride vectors, and stride condition spaces. Let us assume that these transfer condition spaces are already calculated and are annotated to the edges as shown in the Figure 4.18(b). An expected final BDD is shown in Figure 4.18(d), where dotted lines denote false edges in the BDD and solid lines denote the true edges. Here, depending on the control signal values, the appropriate target program block can be reached as shown in Figure 4.18(d) adhering to the branch instruction mentioned above. Once such a BDD is constructed at each program block, deducing a branch instruction at that program block is trivial.

In the following, we present Algorithm 4.11 that constructs such a BDD systematically given the transfer condition spaces to each reachable program block. The main idea of the algorithm lies in the fact that given a program block and its set of reachable program blocks, a scanning variable may divide the set of reachable program blocks into two mutually exclusive disjoint sets. If a scanning variable divides the set of reachable program blocks into two sets, it is taken as the decision variable in constructing the BDD and one of the sets goes to the
false edge and the other goes to the true edge, cf. Figure 4.18(c). Furthermore, these disjoint sets may divide again into two distinct disjoint sets by considering the next scanning variable in the order. Similarly, a BDD node and true and false edges are created with the scanning variable as a node in the next level of the BDD, cf. Figure 4.18(c). This procedure is repeated until we end up finally in a set containing only a single program block. At any stage, during the construction of the BDD, if a considered scanning variable does not divide the set of reachable program blocks at that stage, it is discarded and the next variable in the scanning order is considered. This ensures that we always construct only Reduced Order Binary Decision Diagrams (ROBDDs) [Kro99]. For variable ordering in the BDD, the scanning order of the variables is used. The maximum
4. Automatic Code and Interconnect Generation

depth or level of such BDD would obviously be the number of variables that are involved in the BDD construction, which is also equal to the number of control signals required.

While creating disjoint sets of reachable program blocks w.r.t. a scanning variable, we consider only the sub indexspaces\(^9\) of the transfer condition spaces w.r.t. to the scanning variable and intersect all the transfer condition spaces of all reachable program blocks. If the intersection of any two transfer condition spaces is not empty, we group the corresponding reachable program blocks by this transfer condition space into the same set. Else, we put them in two distinct disjoint sets. For example, in Figure 4.18, w.r.t. iteration variable \(j\) reachable program blocks from \(PB2\) are divided into two disjoint sets \{\{PB2\}, \{PB0, PB3\}\}. The transfer condition spaces for these two disjoint sets are given by \(11 \leq j \leq 14\) and \(j = 15\), respectively as shown in Figure 4.18(c). In other words, there exists two distinct transfer condition spaces which separate these two distinct groups. These transfer condition spaces are also saved as edge attributes in the BDD. The same procedure is applied for all the scanning variables while constructing the BDD and creating the disjoint sets of reachable program blocks. In Figure 4.18, disjoint set of program blocks \{\(PB0, PB3\)\} mentioned above is further divided into two disjoint sets \{\{PB0\}, \{PB3\}\} w.r.t. to variable \(i\) as shown in Figure 4.18(c).

In this approach of constructing a BDD, we need a data structure for representing a collection of disjoint sets [DD08]. A common way of representing a disjoint set is as a forest, or collection of trees, with one tree corresponding to each set of the partition or the disjoint set [DD08]. The disjoint set structure has to support the following operations:

- **MAKE-SET(\(x\))**: Create a new set \{\(x\)\} containing the single element \(x\). The object \(x\) must not appear in any other set in our collection. The leader of the new set is obviously \(x\). The leader is nothing but the representative of the set.
  
  - In our approach, we will also use MAKE-SET(\(x\)) to create singleton sets where \(x\) denotes a reachable program block. The leader of this new set is obviously \(x\) itself.

- **FIND-SET(\(x\))**: Find (the leader of) the set containing \(x\).
  
  - In our approach, FIND-SET(\(x\)) will return the leader of the set containing program block \(x\).

---

\(^9\)A sub indexspace is a sub space of the indexspace, i.e., for a given list of iteration variables the corresponding constraints are returned.
Algorithm 4.11: Disjoint set decomposition of reachable program blocks.

**Input**: Set of reachable program blocks $PB_R$ as well as their corresponding set of transfer condition spaces $I_{TCS}$, and a scanning variable $i_v$.

**Output**: Two disjoint sets of reachable program blocks.

/* for this algorithm we use the disjoint set forest data structure */

begin

foreach $PB_R \in PB_R$ do

MAKE-SET($PB_R$)

/* making singleton set for each reachable program block in the set $PB_R$ */

/* initializing forest structure as well */

end

foreach $PB_R \in PB_R$ do

if $PB_R == FIND-SET(PB_R)$ then

$PB_{\text{temp}} = \text{getIntersectedReachablePBs}(PB_R, PB_R, I_{TCS}, i_v)$

if $PB_{\text{temp}} \neq \emptyset$ then

foreach $PB_{\text{temp}} \in PB_{\text{temp}}$ do

UNION($PB_R, PB_{\text{temp}}$)

end

end

else

/* $PB_R$ is already an element of the set or tree rooted by $FIND-SET(PB_R)$ */

end

end

/* trees in the forest are nothing but the disjoint sets */

end

• **UNION (A, B)**: Replace two sets A and B in our collection with their union $A \cup B$. For example, **UNION (A, MAKE-SET (x))** adds a new element x to an existing set A.

  - In our approach, $A \cup B$ replaces the two sets with a single set and leader of the union set is same as the leader of the set A. The two sets $A, B$ are disjoint; hence $A \cup B$ will also be disjoint.

Algorithm 4.11 summarizes the procedure of dividing set of reachable program blocks into two disjoint sets w.r.t. a particular scanning iteration variable $i_v$. Now, applying Algorithm 4.11 for each scanning variable in the given order
4. Automatic Code and Interconnect Generation

**Function 4.12:** getIntersectedReachablePBs(PBR, PB_R, ITCS, iv)

**Data:** A program block PBR, set of reachable program blocks PB_R as well as set of transfer condition spaces set ITCS, and a scanning iteration variable iv.

**Result:** Intersecting reachable program block set PB_ret.

```plaintext
begin
PB_ret = ∅
foreach PB_c ∈ PB_R do
    if PB_c ≠ PBR then
        ℐ_tmp = subIndexSpace(ℐ_PBR, iv) ∩ subIndexSpace(ℐ_PBC, iv)
        if ℐ_tmp ≠ ∅ then
            PB_ret = PB_ret ∪ {PB_c}
        end
    end
end
return PB_ret
```

and constructing the BDD as aforementioned for the example in Figure 4.18(b) results in exactly the BDD as shown in Figure 4.18(c). The sub indexspaces of the transfer condition spaces are also annotated to the edges of the BDD as shown in Figure 4.18(c).

**MTMDD Approach:**

**Example 4.4** Figure 4.19(a) shows part of the control flow graph of Figure 4.13(b) where PB2 has six outgoing edges PB2 → PB2, PB2 → PB0, PB2 → PB3, PB2 → PB5, PB2 → PB7, and PB2 → EXIT. Therefore, three control signals are required in the branch instruction and the semantics of the branch instruction would be as follows:

```
BRU IF IC2 IC1 IC0 JMP
```

Here, depending on the values of IC2, IC1, and IC0, the execution jumps to one of the target program blocks in the branch instruction. Next, these reachable target program blocks should be encoded systematically as jump targets in the branch instructions—given the transfer condition spaces and the scanning order of the variables. These transfer condition spaces can be automatically determined using Algorithm 4.10 given the program blocks, stride vectors, and stride condition spaces. Let us assume that these transfer condition spaces are already calculated and are annotated to the edges as shown in the Figure 4.19(b).
expected final MTMDD is shown in Figure 4.19(c), where depending on the control signal values, the appropriate target program block can be reached adhering to the branch instruction mentioned above. Once such a MTMDD is constructed at each program block, deducing a branch instruction at that program block is trivial.

![Control Flow Graph](image)

**Figure 4.19:** An exemplary control flow graph showing its reachable program blocks as well as their corresponding transfer condition spaces for illustrating the need of a MTMDD to deduce the branch instruction.

**BDD Approach Vs MTMDD Approach:**

As aforementioned, either to use the BDD approach or MTMDD approach depends on the way the program blocks are created. However, both approaches
have their pros and cons. In the BDD approach, we need at most two control signals to decide a jump among the program blocks as shown in Figure 4.13(a). In case of the MTMDD approach, we need three control signals to decide a jump among the program blocks as shown in Figure 4.13(b). For the BDD approach, GC hardware is simple and we need to implement only either true branch or false branch. For instance, at most we need four comparators for each program block for the example of Figure 4.13(a). For the MTMDD approach, GC hardware is costly and we need to implement all the edges and evaluate them in parallel. For instance, we need 24 comparators in the worst case for each program block for the example of Figure 4.13(b). Furthermore, generating code from the $G_{CF}$ of Figure 4.13(b) might be difficult later—as there are more than one forward edge compared to $G_{CF}$ of Figure 4.13(a) where at most there is only one forward edge. It also implies that the dominator tree in case of Figure 4.13(a) do not have multiple branches compared to the dominator tree in case of Figure 4.13(b). This makes it easier to generate the code subsequently in the next steps. Therefore, in the rest of the dissertation, we focus more on the BDD approach where program blocks are generated using the Algorithm 4.5.

In the next step, the BDD is converted into an equivalent BDD by a) replacing node names by control signals $ICx$, b) converting the edges to either true or false. Figure 4.18(d) shows this similar BDD for Figure 4.18(c) where transfer condition spaces on the edges are converted to either true or false edges to help in encoding the branch instruction. It is to be noted that the condition spaces stored as an attributes of edges in a BDD are used in the global controller for generating the control signals when code corresponding to this program block is getting executed by a processor. Pseudo code in Algorithm 4.13 summarizes the construction of this equivalent BDD and the procedure of generating the branch instructions where each branch target is a reachable program block. Following the procedure in Algorithm 4.13 results in the exact branch instruction given again below for the example of Figure 4.18(d), which contains the control signal flags as well as branch targets, i.e. reachable program blocks, and control signal values that lead to these reachable program blocks.

$$\text{BRU \ IF \ IC1 \ IC0 \ JMP \ } \{\begin{array}{c} PB3 \\
PB0 \\
P2 \end{array} \}$$

These branch instructions are also referred to as Branch Lines (BL) in the rest of the dissertation. These branch lines are also saved as attributes of program blocks for future usage.

Putting it altogether, so far the program block control flow graph as well as the program block assembly codes and necessary branch instructions are computed, cf. overview of code generation approach in Figure 4.5. In the next section, an approach for code compaction using all this knowledge is presented.
Algorithm 4.13: Conversion of the BDD into branch instruction with placeholders.

**Input**: BDD of a program block.

**Output**: Branch instruction of a program block containing control signals as input and the set of reachable program blocks as branch targets.

1 begin
2 Step 1: Assign levels $l_i$ only to the the non-leaf nodes in the BDD with root node level as zero. Compute the maximum level $l_m$ in the BDD.
   
   /* Maximum level $l_m$ is equal to the number of required control signals */
3 Step 2: Annotate the non-leaf nodes with a control signal identifier $IC + l_m - l_i$.
4 Step 3: Interpret the one of the outgoing edge at each node as true and the other as false.
5 Step 4: Visit all the leaf nodes starting from the root node. Binary values along the edges represent position of the leaf node, i.e., reachable program block in the branch instruction.
6 Step 5: Control signal flags in the branch instruction are given by the control signal identifier strings of the nodes in the descending order.
7 Step 6: Finally, generate the branch instruction with the control signal flags and the branch targets.
8 end

### 4.6 Compact Code Generation

For each processor class, a control flow graph $G_{CF}$ whose nodes represent the program blocks, is generated as explained in Section 4.5. In order to fulfill the register assignment according to a given schedule, it is sometimes necessary to unfold some loops in the control flow graph $G_{CF}$ using procedure outlined in Section 4.5.1. For each program block, its basic assembly code sequence is known already and an appropriate branch instruction is computed where only branch targets are still placeholders (Section 4.5.2). Compact code for each processor class can be then generated in three steps as given below, using $G_{CF}$ and annotated assembly code sequences of each program block:

1. In the first step, for each node ($PB$) in the $G_{CF}$, instruction sequences of multiple iterations are packed contiguously adhering to the given schedule
to generate so-called *Overlapped Codes*\(^\text{10}\) for each node (PB). In \(G_{CF}\), these overlapped codes are saved as a node attribute.

2. In the second step, these overlapped codes are packed contiguously again with other overlapped codes from adjacent nodes while adhering to the given schedule and according to the execution order determined by \(G_{CF}\). Although we are using the term overlapped code, note that only in the last step, ‘real’ code is emitted.

3. In the third and final step, the branch targets (placeholders pointing to different reachable program blocks) in the branch instructions will be replaced by absolute numbers pointing to the same reachable program blocks.

In the following, each of these steps is explained in detail with examples.

### 4.6.1 Overlapped Code Generation of Individual Program Blocks

Generating the compact PB code is essential since the available instruction memory of the PEs of a massively parallel processor array is limited and most importantly the code should be independent of the iteration space size or problem size. The process of code compaction involves identifying repeating code sequences while contiguously arranging the instructions of distinct iterations according to the schedule and inserting the loop or branch instructions so that the same code sequence is executed multiple times.

Exemplary overlapped codes for the program blocks of our running FIR filter example are shown in Figure 4.20. The overlapped code of PB0, denoted as PB0\(_{OC}\) is shown in Figure 4.20(a). Here, each horizontal line denotes a single VLIW instruction. As there is only one iteration that gets executed when the execution enters PB0, there is no possibility for any contiguous arrangement of instructions from different iterations. Whereas for PB1, the overlapped code (PB1\(_{OC}\)) is shown in Figure 4.20(b) in which instructions from six different iterations may be arranged contiguously since six iterations are executed consecutively whenever the execution enters PB1. Similarly, overlapped codes of other program blocks PB2, PB3, PB4, PB5, PB6, PB7, and PB8, denoted by PB2\(_{OC}\), PB3\(_{OC}\), PB4\(_{OC}\), PB5\(_{OC}\), PB6\(_{OC}\), PB7\(_{OC}\), and PB8\(_{OC}\), are shown in Figures 4.20(c) to 4.20(i).

A closer look at the overlapped code of PB1, PB1\(_{OC}\) shown in Figure 4.20(b) reveals that there exist repetitive patterns of VLIW instructions, e.g., word (E, B, D) repeats. Indeed, the instruction (3), (4), and (5) are the same as

\(^{10}\) If \(G_{CF}\) contains a node executing multiple iterations consecutively, there exists a possibility to pack contiguously the instructions from distinct iterations for that node.
Figure 4.20: Overlapped codes of individual program blocks for the partitioned and scheduled FIR filter example and processor class 0, $PE(0,0)$. 
instruction (2). Therefore, the code for PB1 can be optimized as shown in Figure 4.20(j) by introducing a loop for such equal repetitive instructions. In this figure, a branch (jump) instruction denoted by an arc is introduced from instruction (2) to (2) without any timing penalty as our PEs do have a branch unit with zero overhead. It is the responsibility of the global controller to generate appropriate control signals to control the branching to either instruction (2) or instruction (3) according to the given schedule. In other words, introducing such a branch instruction in order to compact the code preserves the given schedule. Similarly, for program blocks PB4 and PB7, optimized overlapped codes can be also generated as shown Figures 4.20(k) and 4.20(l), respectively.

For the example of Listing 4.1, for the schedule of iterations in Figure 4.14(a), six consecutive iterations of PB1 are executed whenever the execution enters PB1. So, by packing contingously six consecutive iterations from PB1, overlapped code can be generated as shown in Figures 4.21(c) and 4.21(d) denoted as PB1_OC0 and PB1_OC1, respectively. When the execution enters PB1, a question arises which instruction sequence, either PB1_0 or PB1_1, is to be used for the first iteration to start packing contingously the instructions from the other iterations, cf. Figure 4.14(d). For instance, considering PB1_0 for the first iteration and PB1_1 for the second iteration, alternately for 6 iterations and packing instructions from different iterations will result in PB1_OC0. Similarly, by considering PB1_1 for the first iteration and PB1_0 for the second iteration and so on, will result in PB1_OC1, i.e., in total two overlapped codes for PB1. The same explanation holds true for PB0 to have both, PB0_OC0 and PB0_OC1, as shown in Figures 4.21(a) and 4.21(b), respectively. However, they are same as the non-overlapped codes since there is no self-edge on PB0; thus, there is no possibility to pack the instructions from different iterations of the same program block.

A closer look at PB1_OC0 and PB1_OC1, see Figures 4.21(c) and 4.21(d), reveals that after instruction 3, the code starts repeating. For instance, instruction 4 is the same as instruction 2, and instruction 5 is the same as instruction 3. Thus, PB1_OC0 and PB1_OC1, can be further optimized as shown in Figures 4.21(e) and 4.21(f), respectively, and required branch instructions, in the form of arcs, are inserted. It is the responsibility of the global controller to generate the proper control signals for branching to an appropriate next instruction. However, introducing such a branch instruction does not affect the other data flow operations since the given schedule is untouched.

From the above two intuitive examples, a few important observations can be made. The first one is that once the execution enters a program block, it has to execute a minimum number of iterations denoted by \( N_{PB_{min}} \) so that the overlapped VLIW instructions are repeated [BHT13*]. This minimum number of iterations whose values depends on the values of local latency, iteration interval, and the super iteration interval can be calculated as shown in Function 4.14.

134
4.6 Compact Code Generation

This function checks whether a code repeats or not in a program block by calculating the minimum number of iterations $N_{PB_{\min}}$ and comparing it with the actual number of iterations $N_{PB}$ that are consecutively executed, given by the weights on the self edge of a program block. If a program block satisfies the relation $N_{PB} \geq N_{PB_{\min}}$, then the code repeats and a branch instruction is expected in the code otherwise all the instructions from distinct iterations are overlapped and contiguously packed.

The second observation is that contiguous packing of the instructions from distinct iterations into looped execution depends also on the iteration interval and super iteration interval. The iteration interval determines when to start overlapping the instruction from the next iteration and the super iteration interval determines the assembly code sequence that have to be picked for the considered program block adhering to the cyclic register allocation. While overlapping

---

**Figure 4.21:** Overlapped codes of individual program blocks for the scheduled program of Listing 4.1. It is to be noted that for each program block there exists two assembly code sequences due to the super iteration interval; thus, there exist also two overlapped codes for each program block.
Function 4.14: isCodeRepeatsInProgramBlock($N_{PB}$)

Data: Number of consecutively executed iterations in a program block $N_{PB}$, Local Latency (LL), Iteration Interval (II), and Super Iteration Interval (SII)

Result: Boolean repeats that denotes whether code repeats or not in a program block.

1 begin
2     Boolean repeats ;
3     $N_{PB_{min}} = \lceil \frac{LL}{II} \rceil - 1 + 2 \times SII$ ;
4     if $N_{PB} \geq N_{PB_{min}}$ then
5         repeats = true ;
6     end
7     else
8         repeats = false ;
9     end
10    return repeats ;
11 end

the instructions from distinct iterations, after few iterations only the stable and repetitive code [RST92] pattern emerges. The code before this repetitive code is also called prologue [RST92] (cf., Figure 4.20(b)). The stable code is repeated for few iterations and when overlapping instructions of all the $N_{PB}$ iterations is finished, eventually an epilogue [RST92] code is generated. The main crux of our compaction approach lies in identifying the repeating pattern and replacing with a looping or branching instruction. Therefore, it is sufficient to overlap only the instructions from the required number of iterations so that the first repeating code pattern occurs. Here, this required number of iterations is given by $N_{PB_{min}} - 1$ in the case of program block for which the code repeats. However, in the case of a program block where code does not repeat, instructions from all $N_{PB}$ iterations are overlapped.

The pseudo code in Algorithm 4.15 summarizes this overlapping code generation process for an individual program block. Here, for a program block more than one overlapped code is generated in the case of SII $\neq$ II; thus for each program block there may exist SII overlapped codes, cf., Figure 4.21. Each of these overlapped code is generated by overlapping only the required number of iterations as mentioned above and by choosing appropriate code sequence of the program block for each iteration as given in Algorithm 4.15.

The third observation while generating overlapped code is that branch locations need to be computed. In Algorithm 4.15, $\text{branch}_{\text{out}}$ denotes the exact
Algorithm 4.15: Overlapped Code Generation for an Individual Program Block

**Input**: Program block $PB$, Program block code sequences $PB_{\Psi_{0,1,\ldots,SII-1}}$, Number of iterations $N_{PB}$

**Output**: Overlapped Code $PB_{OC_{0,1,\ldots,SII-1}}$, Branch locations in $branch_{in}$, Branch location out $branch_{out}$

```plaintext
1 begin
2    /* Iteration Interval (II), Super Iteration Interval (SII), Local Latency (LL) */
3    Boolean repeats = isCodeRepeatsInProgramBlock($N_{PB}$)
4    int numit = 0
5    if repeats then
6        numit = $N_{PB_{min}} - 1$
7    end
8    else
9        numit = $N_{PB}$
10   end
11    /* Overlapping the code sequences of different iterations up to numit. */
12    for i = 0; i < numit; ++i do
13        for index = 0; index < SII; ++index do
14            /* Cycling through the different code sequence of program blocks. */
15                for l = 0; l < LL; ++l do
16                    From $PB_{index}$ choose instruction $l$ and add it to $PB_{OC_{index}}$ at
17                        position $i \times \Pi + l$
18                end
19            end
20        end
21    end
22    // Store the branch locations.
23    if repeats then
24        $branch_{in} = LL - \Pi$
25        $branch_{out} = (LL - \Pi) + (SII \times \Pi) - 1$
26    end
27    else
28        $branch_{out} = numit \times \Pi - 1$
29    end
30 end
```

instruction location where branching happens (branch instruction location) and $branch_{in}$ corresponds to the instruction within the generated overlapped code.
that points to the start of the repeating code. In Figure 4.20(j), both \textit{branch\textsubscript{out}} and \textit{branch\textsubscript{in}} both points to the same instruction 2 whereas in Figure 4.21(e) \textit{branch\textsubscript{out}} is pointing to instruction 3 and \textit{branch\textsubscript{in}} is pointing to instruction 2. It is to be noted that the instruction numbers shown beside the overlapped codes of program blocks are relative to their starting positions. In the next step, when overlapped codes of different program blocks are overlapped again adhering to the control flow graph $G_{CF}$ order, instruction numbers within overlapped code of a program block may change because instructions from the other program blocks may precede or succeed. Therefore, the \textit{branch\textsubscript{out}} and \textit{branch\textsubscript{in}} are just pointers to the respective instruction locations, which get automatically adjusted later on.

\subsection{4.6.2 Combining Overlapped Codes of All Program Blocks}

Once the optimized overlapped codes for individual program blocks are generated, it is now feasible to merge these into one single program code with Algorithm 4.16 adhering to the execution order defined in the program block control flow graph. However, exact branch locations, i.e., instruction numbers are known only after this step. Therefore, insertion of branch instructions needs to be delayed to the final step in Section 4.6.3.

For the FIR filter of Figure 4.6, the control flow graph is shown in Figure 4.7, the optimized overlapped codes are shown in Figure 4.20. Figure 4.22(a) shows the overlapping of the optimized overlapped codes of different program blocks according to the order defined in the control flow graph $G_{CF}$. Similarly, for the Listing 4.1, the partially unrolled control flow graph is shown in Figure 4.14(b), the optimized overlapped codes are shown in Figure 4.21. Figure 4.23(a) shows the overlapping of the optimized overlapped codes of different program blocks according to the order defined in the transformed control flow graph $G_{TCF}$. For illustration purposes, in Figures 4.22(a) and 4.23(a) branches are still shown as arcs. In Algorithm 4.16, while combining the overlapped codes from different program blocks in the execution order, an appropriate overlapped code is chosen for each program block. This chosen overlapped code is appended to a single program code and the starting positions of the program blocks are also stored to replace the branch target locations later on. For the last program block, as there is no succeeding program block to continue, starting of the final program’s epilogue code is required so that it can be inserted as a branch target location in its branch instruction which is calculated as shown in Algorithm 4.16. After this step, a final program code is obtained whose length is fixed. In the last step, branch instructions are inserted into this code.
4.6 Compact Code Generation

(a) Resulting program code after combining overlapped codes from all program blocks. Branch lines are still shown as arcs in this code.

(b) Branch lines are inserted into the program code shown on the left.

Figure 4.22: Result of combining overlapped codes of all program blocks into an assembly code for the FIR filter description of Listing 3.3 before and after inserting branch instructions.

4.6.3 Branch Line Insertion

In the previous section, starting positions of the program blocks are calculated, in Section 4.6.1 where to insert the branch instructions are calculated, and in Section 4.5.2 the branch instructions itself are derived. Now, in this final step of the compact code generation, branch lines are inserted into the final program code. Here, placeholders for the branch targets that point to the distinct program blocks are replaced by absolute instruction numbers since the starting positions of the program blocks are available now. The pseudo code in Algorithm 4.17 summarizes this branch line insertion procedure where placeholders are replaced by starting position of the program blocks. If the placeholder itself is a self jump then it is replaced by the addition of its program block start position and $\text{branch}_{in}$ else it is replaced by the start position of the program blocks. These translated branch lines are inserted in pre-calculated $\text{branch}_{out}$ positions in the code. For all the program blocks with only one successor, "NEXT" is
inserted as the branch instruction and the remaining all instructions in the final program code are also inserted with "NEXT" as branch instruction as given in Algorithm 4.17. Applying this algorithm for the code in Figures 4.22(a) and 4.23(a) results in Figures 4.22(b) and 4.23(b), respectively.

To summarize, for the example of FIR filter given in Listing 3.3 and shown in Figure 4.6, our approach of compact code generation produces a compact Final Optimized Code (FOC) of 17 VLIW instructions only as shown in Figure 4.22(b) whereas flat yet lengthy code would need 66 VLIW instructions for the same sizes of filter taps \( N \) and samples \( T \) and same space-time mapping. Similarly, for the Listing 4.1 even though the control flow graph is partially unrolled as shown in Figure 4.14(b), our approach results in FOC of 12 VLIW instructions whereas flat code requires 44 VLIW instructions. It can be verified easily that the code size will remain constant and independent of the size of the iteration space. For instance, either increasing the number \( N \) of filter taps or the number \( T \) of input samples will only change the number of iterations in the program blocks, i.e., program blocks may grow only in size and the number of program blocks remain the same. An increased number of iterations in a program block may not increase its code size since the repeating code in a program block is just executed few more times. Therefore, only configuration parameters
Algorithm 4.16: Combining Overlapped Codes of All Program Blocks.

**Input**: For each program block overlapped codes \( PB_{OC_{0...SII-1}} \), Number of iterations \( numit \), Branch instruction location \( branch_{out} \) calculated in Algorithm 4.15, and the control flow graph \( GC_{F} \).

**Output**: Combined program code “code”, Start positions of program blocks “pos”

```plaintext
define begin
  pbStartCodePosition = 0
  lastPBStartCodePosition = 0
  pbBranchOut = 0
  /* |V| denotes the number of nodes in the GC_{F} and nodes are considered in the execution order */
  for \( j = 0; j < |V|; + + j \) do
    pos(PB\(_j\)) = pbStartCodePosition
    /* Appropriate code from the available overlapped codes is chosen to satisfy the register allocation, PB\(_{OC_{\xi}}\) */
    for \( k = 0; k < |PB_{OC_{\xi}}|; + + k \) do
      1. Add each of \( PB_{OC_{\xi}} \) code lines to the end of the code accordingly.
      2. If the execution jumps from this program block to another target program block (except the first program block) and at that target program block if more than one incoming program blocks are present, epilogue codes from all the incoming program blocks should be the same.
      3. If the target program block is first program block, epilogue code of this program block is also merged with the starting program block as well as with the succeeding program block.
    end
  lastPBStartCodePosition = pbStartCodePosition
  /* Iteration Interval (II) */
  pbStartCodePosition = pbStartCodePosition + numit\(_j\) \times II
  pbBranchOut = branch_{out}\(_j\)
end
```

of the global controller are changed but the assembly codes of the PEs remain the same—making the generated code independent of the iteration space size or problem size.
4. Automatic Code and Interconnect Generation

Algorithm 4.17: Branch Line Insertion.

Input: Program code “code”, start positions of program blocks “pos” calculated in Algorithm 4.16, branch_in as well as branch_out calculated in Algorithm 4.15, and computed branch line BL of each program block calculated in Section 4.5.2, and the control flow graph G_CF.

Output: Combined Program Code “code” inserted with branch instructions.

1 begin
2     /* |V| denotes the number of nodes in G_CF and nodes are considered in the execution order */
3     for j = 0; j < |V|; ++j do
4         if PB_BL_j is empty then
5             /* Nowhere to jump, place NEXT at the end. */
6                 continue
7         end
8         if PB_BL_j has only one entry then
9             /* Unconditional jump for a PB irrespective of the control signal it will jump to the succeeding program block */
10            /* BRU: IF IC0 JMP cycle<target>, cycle<target> */
11             end
12         else
13             /* Prefix appropriate number of signals in descending order to line as mentioned in Section 4.5.2. */
14             /* Replace each branch target with: */
15             if target is PB_j, i.e., self jump then
16                 pos_PB_j + branch_in_j
17             end
18             else
19                 /* start of the target program block */
20                 pos_target
21             end
22             /* Add this modified branch line at position pos_PB_j + branch_out_j. */
23         end
24         /* Insert "BRU: NEXT" into remaining code lines. */
25     end
4.7 Functionality of the Global Controller

As aforementioned, the global controller according to Figure 2.8 is an important component in the overall array architecture. It needs to generate the control signals that steer the branches taken in the compacted code of each PE and must therefore arrive at the right cycle. If the currently executing VLIW instruction of a PE includes a branch based on such control signals, the execution will jump to the appropriate target instruction. In case of rectangular tiles, the controller functionality may be described by a set of loop counters where the innermost counter tracks the iterations in the first scanning direction of the tile and the outer most counter tracks the iterations in the last scanning direction. For the FIR filter example shown in Figure 4.7(a), we need two nested counters as described in Listing 4.5, the first counter for scanning direction \( j \) counts from 0 to 7. Once the inner counter reaches its bound 7, the second counter which tracks the direction \( i \) is incremented by one and the first counter is reset to zero. The second counter upper bound is also 7. The global controller generates the control signal \( IC0 = 1 \) whenever \( j = 6 \) else 0.

### Listing 4.5: A simple software emulation of the global controller for the FIR filter example

```plaintext
for(i1=0, i1 <= 7, i1 =i1+1) { 
  for(j1=0, j1 <= 7, j1 =j1+1) 

    if(1 \leq i1 \leq 6 \land 1 \leq j1 \leq 6) { 
      if (j1==6) { 
        IC0 = 1;
      } else { 
        IC0 = 0;
      }
    }

  }
}
```

For instance, in Figures 4.17 and 4.22(b), when VLIW instruction (8) is executing, there are two possibilities for the next instruction: The execution may jump to instruction (8) belonging to PB4 again if we are executing the loop body at iterations \( j_1 < 6, 1 \leq i_1 \leq 6 \) in the iteration space or continue with instruction (9) if we are at iterations \( j_1 = 6, 1 \leq i_1 \leq 6 \) (see Figure 4.7(a), PB4) in the iteration space. In the global controller, whenever \( j_1 = 6 \), it generates the control signal \( IC0 = 1 \) else 0. In fact, \( IC0 \) is a binary control signal which is evaluated by the branch unit of a PE. In instruction (8), if \( IC0=1 \), the program
jumps to instruction (9) else instruction (8) is executed again. The key benefit of our global control approach is that the mentioned control signal sequences are needed by all processors. Therefore, we proposed to implement the controller outside the processor array and generated control signals are propagated to all processors in a linearly staggered fashion with appropriate delays, more details to follow in Chapter 5.

4.8 Interconnect Generation

For a given TCPA architecture, the number of PEs, their configuration and the number of links (data and control) between the PEs are already specified and considered as a resource constraint during scheduling. Moreover, as our compilation approach is based on the polyhedron model, interconnections among different PEs can be automatically synthesized. The processor array interconnection structure is responsible for the propagation of data and control signals within the processor array. The interconnect architecture consists of dedicated links with the delay shift registers for each \(d_{ji}\). One can determine the synthesis of the processor interconnection for a data dependency \(d_{ji}\) as implied by an equation \(x_i[I] = F_i(\ldots, x_j[I - d_{ji}], \ldots)\) by first determining the processor displacement, \(d_{ji}^p\), as

\[
d_{ji}^p = Q \cdot I - Q \cdot (I - d_{ji}) = Q \cdot d_{ji}
\]

where \(Q\) denotes the allocation matrix.

**Definition 4.3 (Inter-processor and Intra-processor dependencies)** A dependency \(d_{ji}\) is called inter-processor dependency if its processor displacement is non-zero, i.e., \(d_{ji}^p \neq 0\). If \(d_{ji}^p = 0\) then it is called intra-processor dependency. The vector \(d_{ji}^p\) denotes the distance in the processor space between the processor where \(x_j\) is computed and the processor where \(x_i\) is computed using this result.

The set of all processor displacement vectors of an LSGP partitioned and scheduled PRA excluding the zero vector, is denoted by \(D^p\) in the following. Then, depending on the schedule vector \(\lambda\), the temporal delay for each data dependency is determined by time displacement, \(d_{ji}^t\) which can be calculated using the following equation.

\[
d_{ji}^t = \lambda \cdot d_{ji} + \tau(x_j) + w(x_j) - \tau(x_i)
\]

In Equation (4.7), \(w(x_j)\) denotes the execution time of node \(x_j\), i.e., execution delay of the functional unit assigned to that node. Applying processor allocation and scheduling, the semantically equivalent equation is \(x_i[p, t] = F_i(\ldots, x_j[p - d_{ji}^p, t - d_{ji}^t], \ldots)\). In a processor array, intermediate results are propagated from
4.8 Interconnect Generation

the source PE $p_{src} = p - d_{pi}^j$ to the target PE $p_{tar} = p$ using delay registers whose length is given by the time displacement, $d_{pi}^j$. If the number of delay registers is large, controlled delay registers [Bed09, BT01], FIFOs, or embedded memories are used. Figure 4.24 shows that the processor displacement vectors for the FIR filter example is given by $D^p = \{(1\,0)^T, (0\,1)^T, (1\,1)^T\}$. To state in simple words, once the processor displacement is known for an inter-processor dependency, the source processor as well as the target processor are also known for an interconnect corresponding to this dependency. The delay register value by which the data from the source processor to the target processor is delayed is given by time displacement.

Interconnections can be deduced as mentioned above for each PE in the TCPA. However, this process is time consuming particularly in the case of larger array sizes. Furthermore, the configuration memory size as well as configuration time will also increase in accordance with array size. Instead, if we can group few PEs in so-called interconnect classes where all PEs in this class will have same interconnection scheme, both the configuration memory and configuration time can be reduced to a great extent. This kind of grouping and configuration is already supported in our multi-cast configuration scheme as mentioned in Section 2.1.3.

For the FIR filter given in Listing 3.3, Figure 4.24 shows the distinct interconnect classes where all processors in an interconnect class are ($P_{IC}$) are identified with same color shade under their processor names. For instance, interconnect class $P_{IC4} = \{1 \leq p_1 \leq 2 \land 1 \leq p_2 \leq 2\}$, all processors receive ‘a’ values from their just above processors at a processor displacement of $d_{pi}^j = (1\,0)^T$ and ‘y’ values from left neighbors at a processor displacement of $d_{pi}^0 = (0\,1)^T$. All the processors in this class get ‘u’ values from processors at displacements of $(0\,1)^T$, $(1\,0)^T$, and $(1\,1)^T$ as shown in Figure 4.24. Similarly, all the processors in this interconnect class will write to processors at same processor displacements. All processors in an interconnect class will have the same topology of interconnect structure as shown in Figure 4.24. Now, for each inter-processor dependency time displacement can be calculated using Equation (4.7). For variable ‘a’ inter-processor dependency $d_{jia} = (70 - 10)^T$ time displacement $d_{jia}^t$ is zero which means that it is propagated directly from a source processor to a target processor without any delay register in between. For this dependency, time displacement can be calculated using schedule vector $\lambda = (8\,1\,57\,8)$, $\tau(x_j) = 0$, $w(x_j) = 1$, and $\tau(x_i) = 0$. Similarly, for all other inter-processor dependencies time displacements can be calculated, which also result in zero. Therefore, in Figure 4.24, interconnections among processors are shown without any delay registers.

To summarize, once the processor displacements and time displacements are known for each inter-processor dependency interconnection structure can be derived from the given scheduled and partitioned loop program itself. In the
Figure 4.24: Interconnection of different processing elements with input registers (ID\(x\)) and output registers (OD\(x\)) for the FIR filter of Listing 3.3. Distinct interconnect classes are also depicted, where all processors belonging to the same interconnect class are having same background color under their PE names.

next section, extracting interconnect classes from the same given loop program is discussed in detail.

### 4.8.1 Interconnect Classes

In this section, we propose a systematic way of deriving a set of interconnect classes. In this approach, we use the fact that the condition spaces associated with the input as well as output statements of a given loop program will determine the processors for which these inputs and outputs are connected, cf., Equa-
4.8 Interconnect Generation

In other words, each input statement $S_{in}$ or output statement $S_{out}$ may decompose the processor space into at most two disjoint subspaces as given by following Equations (4.8) and (4.9), respectively.

$$\mathcal{P}_{in_t} = \{ p \in \mathcal{P} \mid p = Q \cdot I \land I \in \mathcal{I}_{in} \}$$  \hspace{1cm} (4.8)

$$\mathcal{P}_{in_t} = \mathcal{P} \setminus \mathcal{P}_{in_t}$$

$$\mathcal{P}_{out_s} = \{ p \in \mathcal{P} \mid p = Q \cdot I \land I \in \mathcal{I}_{out} \}$$  \hspace{1cm} (4.9)

$$\mathcal{P}_{out_s} = \mathcal{P} \setminus \mathcal{P}_{out_s}$$

Here, $\mathcal{P}_{in_t}$ denotes the target subspace of processors driven by an input $S_{in}$, $\mathcal{P}_{in_t}$ denote the rest of the processor space, and $\mathcal{I}_{in}$ denotes the condition space associated with $S_{in}$. Similarly, $\mathcal{P}_{out_s}$ denotes the source subspace of processors driving an output $S_{out}$, $\mathcal{P}_{out_s}$ denote the rest of the processor space, and $\mathcal{I}_{out}$ denotes the condition space associated with $S_{out}$. Furthermore, each interprocessor dependency may also decompose the processor space into disjoint subspaces. Each such dependency $d_{ji}$ may divide the given processor space into the following disjoint subspaces.

- A source only processor subspace where a dependency $d_{ji}$ can start at each processor in the subspace.
- A target only processor subspace where a dependency $d_{ji}$ can end at each processor in the subspace.
- A source and target processor subspace where a dependency $d_{ji}$ can both start as well as end at each processor in the subspace.
- A not source and not target processor subspace where a dependency $d_{ji}$ can neither start nor end at each processor in the subspace.

In summary, these disjoint subspaces denote different interconnect classes. In the following, we present Algorithm 4.18 where initially the given processor space $\mathcal{P}$ is recursively divided into disjoint subspaces by considering each input, output, and inter processor dependencies.

Applying this algorithm to the FIR filter of Listing 3.3 results in a total of nine interconnect classes as shown in Table 4.2 where all processors in an interconnect class are also assigned the same input and output registers as mentioned in Section 4.2. It is to be noted that for the FIR filter example, the interconnect classes are also same as the processor classes. It means that all the processors in a class are configured with programs first and followed by interconnect configuration—further reducing the configuration memory and the configuration time. Hardware support for this kind of configuration is already built into the configuration management as mentioned in Section 2.1.3.
Algorithm 4.18: Decomposition of the processor space into disjoint subspaces

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Input:} Processor Space $\mathcal{P}$, \(S_{\text{in}}\) be the set of inputs, \(S_{\text{out}}\) be the set of outputs, \(D_{ji}\) be the set of inter processor dependencies, and \(D_{ji}^P\) be the set of corresponding processor displacement vectors.
\State \textbf{Output:} Disjoint decomposition of \(\mathcal{P}\) into interconnect classes, i.e., disjoint subspaces \(\mathcal{P}_{IC} = \{\mathcal{P}_{IC1}, \ldots, \mathcal{P}_{ICr}\}\)
\begin{algorithmic}[1]
\Function{Decomposition}{Processor Space $\mathcal{P}$, $S_{\text{in}}$, $S_{\text{out}}$, \(D_{ji}\), \(D_{ji}^P\)}
\State $\mathcal{P}_{IC} = \{\mathcal{P}\}$
\For{$S_{\text{in}} \in S_{\text{in}}$}
\State $\mathcal{P}_{IC\text{tmp}} = \emptyset$
\For{$\mathcal{P}_{IC} \in \mathcal{P}_{IC}$}
\State $\mathcal{P}_{\text{int}} = \{p \in \mathcal{P}_{IC} \mid p = Q \cdot I \land I \in \mathcal{I}_{\text{in}}\}$
\State $\mathcal{P}_{\text{int}} = \mathcal{P}_{IC} \setminus \mathcal{P}_{\text{int}}$
\State $\mathcal{P}_{IC\text{tmp}} = \mathcal{P}_{IC\text{tmp}} \cup \{\mathcal{P}_{\text{int}}, \mathcal{P}_{\text{int}}\} \setminus \{\emptyset\}$
\EndFor
\State $\mathcal{P}_{IC} = \mathcal{P}_{IC\text{tmp}}$
\EndFor
\EndFor
\For{$S_{\text{out}} \in S_{\text{out}}$}
\State $\mathcal{P}_{IC\text{tmp}} = \emptyset$
\For{$\mathcal{P}_{IC} \in \mathcal{P}_{IC}$}
\State $\mathcal{P}_{\text{out}} = \{p \in \mathcal{P}_{IC} \mid p = Q \cdot I \land I \in \mathcal{I}_{\text{out}}\}$
\State $\mathcal{P}_{\text{out}} = \mathcal{P}_{IC} \setminus \mathcal{P}_{\text{out}}$
\State $\mathcal{P}_{IC\text{tmp}} = \mathcal{P}_{IC\text{tmp}} \cup \{\mathcal{P}_{\text{out}}, \mathcal{P}_{\text{out}}\} \setminus \{\emptyset\}$
\EndFor
\State $\mathcal{P}_{IC} = \mathcal{P}_{IC\text{tmp}}$
\EndFor
\For{$d_{ji} \in D_{ji}$}
\State $\mathcal{P}_{IC\text{tmp}} = \emptyset$
\For{$\mathcal{P}_{IC} \in \mathcal{P}_{IC}$}
\State $\mathcal{P}_{IC_s} = \{p \mid p \in \mathcal{P}_{IC} \land p \in \text{source}(d_{ji}, \mathcal{P}) \land p \notin \text{target}(d_{ji}, \mathcal{P})\}$
\State $\mathcal{P}_{IC_t} = \{p \mid p \in \mathcal{P}_{IC} \land p \in \text{target}(d_{ji}, \mathcal{P}) \land p \notin \text{source}(d_{ji}, \mathcal{P})\}$
\State $\mathcal{P}_{IC_{st}} = \{p \mid p \in \mathcal{P}_{IC} \land p \in \text{source}(d_{ji}, \mathcal{P}) \land p \in \text{target}(d_{ji}, \mathcal{P})\}$
\State $\mathcal{P}_{IC_{\neg st}} = \{p \mid p \in \mathcal{P}_{IC} \land p \notin \text{source}(d_{ji}, \mathcal{P}) \land p \notin \text{target}(d_{ji}, \mathcal{P})\}$
\State $\mathcal{P}_{IC\text{tmp}} = \mathcal{P}_{IC\text{tmp}} \cup \{\mathcal{P}_{IC_s}, \mathcal{P}_{IC_t}, \mathcal{P}_{IC_{st}}, \mathcal{P}_{IC_{\neg st}}\} \setminus \{\emptyset\}$
\EndFor
\State $\mathcal{P}_{IC} = \mathcal{P}_{IC\text{tmp}}$
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}
\end{algorithm}

4.9 Case Studies

We integrated our novel mapping and code generation techniques into the compiler framework PARO [HRDT08, SHTT14]. As aforementioned, it can parse loop specifications described in PAULA [Han09], our notation for nested loop programs. As a mapping platform, different configurations of TCPAs differing in size and capabilities were considered. In Table 4.3, we show the results produced by our proposed code generation technique. As can be clearly seen, the global latency of each application increases with increasing problem size but the emitted code size remains the same, see also [BHT13*, BHT14*]. The primary reason for this is that if the problem size increases, the number and size of each program block does not increase in LSGP partitioning and scheduling, only the number of loop iterations. Thus, the same number of instructions only gets executed a few more times with loop termination signals emitted by the global controller. In other words, the iteration space of a tile that is assigned and executed by a processor is increased but not at the expense of increased code size.
4. Automatic Code and Interconnect Generation

Table 4.2: Different interconnect classes and their corresponding processor elements for the example of FIR filter Listing 3.3.

<table>
<thead>
<tr>
<th>Interconnect class</th>
<th>Processing Elements (indices)</th>
<th>Input/Output ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0 , 0 }^T</td>
<td>ID0, ID1, OD1, OD2, OD3</td>
</tr>
<tr>
<td>1</td>
<td>{0 , 1 }^T, {0 , 2 }^T</td>
<td>ID0, ID2, ID3, OD1, OD2, OD3</td>
</tr>
<tr>
<td>2</td>
<td>{0 , 3 }^T</td>
<td>ID0, ID2, ID3, OD0, OD1, OD2</td>
</tr>
<tr>
<td>3</td>
<td>{1 , 0 }^T, {2 , 0 }^T</td>
<td>ID1, ID4, ID5, OD1, OD2, OD3</td>
</tr>
<tr>
<td>4</td>
<td>{1 , 1 }^T, {2 , 1 }^T, {2 , 2 }^T</td>
<td>ID2, ID3, ID4, ID5, ID6, OD1, OD2, OD3</td>
</tr>
<tr>
<td>5</td>
<td>{1 , 3 }^T, {2 , 3 }^T</td>
<td>ID2, ID3, ID4, ID5, ID6, OD0, OD1, OD2</td>
</tr>
<tr>
<td>6</td>
<td>{3 , 0 }^T</td>
<td>ID1, ID4, ID5, OD2, OD3</td>
</tr>
<tr>
<td>7</td>
<td>{3 , 1 }^T, {3 , 2 }^T</td>
<td>ID2, ID3, ID4, ID5, ID6, OD2, OD3</td>
</tr>
<tr>
<td>8</td>
<td>{3 , 3 }^T</td>
<td>ID2, ID3, ID4, ID5, ID6, OD0</td>
</tr>
</tbody>
</table>

Moreover, it can also be seen for the two shown applications that the number of processors also does not influence the code size per PE.

Next, we compared our approach with the multi-cluster VEX [Hew14] framework. In order to match our test environment with VEX, we have modified the machine model description in the following way: We have specified in each cluster three memory units (2 load, 1 store) as the considered applications need two inputs and one output simultaneously in order to avoid communication bottlenecks. Also, to make a fair comparison, we have specified three integer Arithmetic Logic Unit (ALU) units in each cluster for address generation and loop counter incrementation. We have also modified the delay (DEL) value of these functional units to single cycle as our architecture template considers single cycle operations for each PE in our TCPA configuration an adder, a multiplier and four move units are allocated.

With these modifications, the setup in the VEX compiler matches well with our test environment. A single cluster from VEX is equivalent to a single PE. We compare a single cluster from VEX with a TCPA of size 1 × 1. Similarly, a VEX with four clusters with a TCPA of size 1 × 4. The results are also reported in Table 4.3. It is to be noted that the VEX compiler does not support more than four clusters. Therefore, our comparison is limited to four clusters only. As shown in Table 4.3, in all the considered cases, our generated codes produce a
### Table 4.3: Scalability and code size independence for different applications on a TCPA; see [BHT14*]

| Application   | $|\mathcal{P}|$ | Global Latency (GL) (#cycles) | code size (#instr/PE) (all clusters) |
|---------------|----------------|-------------------------------|--------------------------------------|
|               |                | TCPA                          | VEX                                 |
|               |                | TCPA                          | VEX                                 |
| **FIR**       |                |                               |                                      |
| - 65,536 samples, 64 taps | 1 x 1 | 4,194,306 17,825,822 | 15 33 |
|                | 1 x 2        | 2,097,186 13,631,518         | 15 46 |
|                | 1 x 4        | 1,048,626 12,320,798         | 15 69 |
|                | 1 x 8        | 524,346 –                    | 15 – |
| - 65,536 samples, 128 taps | 1 x 1 | 8,388,610 34,603,038 | 15 33 |
|                | 1 x 2        | 4,194,370 25,690,142         | 15 46 |
|                | 1 x 4        | 2,097,250 22,544,414         | 15 69 |
|                | 1 x 8        | 1,048,690 –                  | 15 – |
|                | 1 x 16       | 524,410 –                    | 15 – |
| - 65,536 samples, 256 taps | 1 x 4 | 4,194,498 42,991,646 | 15 69 |
|                | 1 x 16       | 1,048,818 –                  | 15 – |
|                | 1 x 32       | 524,538 –                    | 15 – |
| **MM**        |                |                               |                                      |
| - 16×16       | 1 x 1        | 4,099 19,118                 | 16 29 |
|                | 2 x 2        | 1,047 12,206                 | 16 64 |
| - 24×24       | 1 x 1        | 13,827 61,302                | 16 29 |
|                | 2 x 2        | 3,487 49,782                 | 16 64 |
|                | 4 x 4        | 915 –                        | 16 – |
| - 32×32       | 1 x 1        | 32,771 141,630               | 16 29 |
|                | 2 x 2        | 8,231 95,550                 | 16 64 |
|                | 4 x 4        | 2,111 –                      | 16 – |
|                | 8 x 8        | 599 –                        | 16 – |
lower latency and code size. This is mainly due to our zero overhead loop control and efficient parallelization using the polyhedron model in our framework.

In the following experiments, we consider apart from an FIR filter, a matrix multiplication (MM), and a sum of absolute differences (SAD) application from H.264 video coding. The generated optimized codes (programs) were also simulated on a cycle-accurate C++ simulator. The proposed approach is compared to the results produced by the Trimaran compilation and simulation infrastructure [The05]. In Trimaran, modulo scheduling as well as rotating register files were selected. We mapped the above algorithms onto a single PE and configured Trimaran as a single clustered VLIW architecture. Each PE in the TCPA is allocated an adder, a multiplier and four move units. Since our considered architecture is a streaming architecture, special address generators (see Figure 2.8) that are responsible for either fetching input data from the input buffers or writing the results back to output buffers, are also needed. In case of the Trimaran architecture, only load/store units are needed. In order to finally match our test environment with Trimaran, we have modified the HPL-PD architecture description in the following way: We have specified three memory units as all the considered applications need two inputs and one output simultaneously in order to avoid communication bottlenecks. Also, to make a fair comparison, we have specified three integer units in Trimaran for address generation and loop counter incrementation. Since our architecture template considers single cycle operations, this option has been modeled also in the HPL-PD architecture. Finally, the target architecture also has been reduced to a single clustered VLIW architecture. With these modifications, the setup in Trimaran matches well with our test environment. In Table 4.4, (a) the average iteration interval \( \Pi_{avg} \), (b) the number of VLIW instructions (code size), and (c) the overheads\(^{12} \) for different applications of our approach are compared against Trimaran. Our experimental results reveal the following:

- The average iteration interval \( \Pi_{avg} \) obtained by our approach is up to 88% smaller than the Trimaran approach. The global latency GL is always less in our approach as we are able to achieve a better \( \Pi_{avg} \). In the Trimaran intermediate representation (IR), there are lot of basic blocks that get executed before the actual loop nest starts. While calculating the average iteration interval, we considered only the nested loop portion from the Trimaran IR and calculated the number of cycles needed to execute the

\(^{11}\)The average iteration interval is the average time between the start of two successive loop iterations. It is calculated by dividing the total execution time of a loop nest GL by the total number of iterations executed.

\(^{12}\)The overhead indicates the amount of time that is spent in executing other than the innermost loop compared to the total execution time.
Table 4.4: Comparison of proposed code generation approach with Trimaran [BHT13*, TBHL15*].

<table>
<thead>
<tr>
<th>Application</th>
<th>FIR 32 taps</th>
<th>MM $16 \times 16$ matrices</th>
<th>SAD $16 \times 16$ block size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Iteration Interval $H_{avg}$</td>
<td>Triman 2.3</td>
<td>2.5</td>
<td>8.4</td>
</tr>
<tr>
<td></td>
<td>Ours 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>%reduction</td>
<td>56.2</td>
<td>58.0</td>
<td>88.0</td>
</tr>
<tr>
<td>Number of VLIW Instructions</td>
<td>Triman 13</td>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>Ours 8</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>%reduction</td>
<td>38.5</td>
<td>5.9</td>
<td>64.3</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>Triman 14.1</td>
<td>26.4</td>
<td>4.6</td>
</tr>
<tr>
<td></td>
<td>Ours 0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

nested loop and the number of VLIW instructions in it to give a fair comparison.

- The number of VLIW instructions is always less compared with Trimaran. This is mainly due to the clear separation of control and data flow in our approach (compiler and hardware) as well as to our exploitation of zero-loop overhead code generation.

- Table 4.4 shows the overheads of Trimaran for different applications. In our method, the overhead is always zero, whereas for Trimaran, the overhead amounts in average to 15% of the total execution time.

In summary, our concept of splitting local and global control is novel and appealing. It is promising since it scales very well with the iteration space and the target program sizes are always smaller compared with Trimaran. We are even able to achieve a better average iteration interval $H_{avg}$ in all considered cases and a zero overhead for a given nested loop application, which greatly outweighs the overhead in hardware for realizing the global controller.

4.10 Summary

In this chapter, we presented a novel code generation, and code compaction approach for TCPAs. For a given loop nest, even if the array contains 1000s
of processors, we have to generate different assembly code sequences only for few processor types called *processor classes*. The same program is loaded and executed in each processor belonging to the same class. Moreover, code for each processor class can be generated by so-called *program blocks* where all iterations in a program block will execute the same sequence of instructions. Thereby, program execution is also represented by a control flow graph of these program blocks which is used in the code generation process. While generating the code for each processor class, repetitive patterns of instructions are identified and looping instruction are introduced to avoid code explosion and to compact the code. Our code compaction approach based on these program blocks exploits zero-overhead. A global controller orchestrates the program branches by issuing a set of control signals so that these looping instructions are executed multiple times in accordance with a given schedule. By this exploitation of looping, the generated code size is there by minimized in order to fit well in the instruction-memory limited processors which in turn reduces the area as well as power consumption. Our approach for multiprocessor code generation shows significantly better results compared with code produced by (a) the Trimaran compilation infrastructure, even for single PE architectures, and (b) the multi-cluster VEX framework. The generated code sizes have been shown to be independent of the loop problem size and also independent of the array size. Finally, we also presented a scalable approach for synthesizing interconnect configuration among processors in the array as well by identifying so-called *interconnect classes* where all processors in an interconnect class share the same topology of interconnect structure.
In the previous chapter, we presented a scalable and problem size independent compact code generation approach for nested loop programs when mapped onto Tightly Coupled Processor Arrays (TCPAs). The main crux of our compaction approach lies in identifying repeating code sequences. In Section 4.6, we discussed in detail how compact code for a processor class candidate element is generated from a given program block control flow graph. While generating code for an individual program block, repeating code sequences are identified already and relative branch instructions are introduced in the program block’s code. In the next step of compact code generation, codes from all program blocks are combined according to the execution order defined in the program block control flow graph and the relative positions of the branch instructions are adjusted accordingly. For a given loop program, this approach results in a compact code which preserves a given schedule of instructions.

Yet for this compact code to be executed correctly, control signals must be provided to steer the correct branching between program block execution at runtime so that the branching instructions are repeated—exactly the intended number of times—in accordance with the given schedule. In Section 4.4, it has been explained that during partitioning of the nested loop program, condition spaces that are dependent on processor coordinates result in distinct processor classes. However, the condition spaces that are independent of the processor coordinates can be implemented outside the processor array instead of inside in each processor to reduce the hardware cost. Here, by using the program block iteration spaces, stride vectors and stride condition spaces—the transfer condition spaces that transfer execution flow from one program block to the other may be calculated. While determining a branch instruction for a given program block, in order to encode different reachable program blocks from it, a Reduced Order Binary Decision Diagram (ROBDD) was constructed where sub indexspaces of the transfer condition spaces to generate the control signals are also calculated. Now, these program blocks and the sub indexspaces of the transfer condition spaces determine the functionality of the Global Controller (GC) hardware and control signals that need to be generated and propagated to the Processing Elements (PEs).
To illustrate this further, in Figure 5.1, an exemplary code generated for processor PE(0,0) of FIR the filter in Listing 3.3 is shown, where the iteration space assigned to this processor as well as distinct program blocks are also identified with distinctly colored rectangular boxes. The program block control flow which is used during code generation is also shown. For instance, for PB4, the prologue of its code corresponds to instruction 6 and 7 whereas epilogue code corresponds to instructions 9 and 10. Instruction 8 corresponds to the kernel of the PB4 code that repeats. Thus, a branch instruction must be introduced to compact the code. From a visual inspection of the generated code and program block PB4, it is evident that whenever $j_1 = 6$, the execution should continue with program block PB5. If $0 \leq j_1 \leq 6$, instruction 6 followed by instruction 7 is executed once whereas instruction 8 is executed repeatedly. At the end of executing iteration $j_1 = 6$, the execution jumps to instruction 9. In this context, GC generates proper control signals (IC[x]) that steer the correct program execution. In summary, these control signals are functions of the iteration being executed in turn a program block. Furthermore, while executing the instruction sequence of a program block, whenever a transfer condition space for another reachable program block is met, control signals are generated to transfer the program execution to the corresponding program block which essentially corresponds to an instruction in the generated code.

The most plausible questions in this context are: (a) How do we know the current iteration that is being executed? (b) How to identify the current iteration’s program block? (c) How to encode and generate proper control signals? If the current iteration and its program block are known, control signals can be generated using the sub indexspaces of the transfer condition spaces that are calculated already while generating code, see Section 4.5.2. Therefore, in the GC, we have to perform the following two main functions: Firstly, the entire iteration space has to be scanned or enumerated according to the computed schedule $\lambda$. In other words, at each iteration, we have to calculate the next iteration which will get executed. As the scanning order is the same for all processors, this iteration must be obviously generated only once. Secondly, based on the computed iteration and a program block, proper control signals have to be generated. As a requirement, the hardware of the GC has to be fully programmable—depending on the application, new configuration data is generated to configure it. Here, a credible question is: (d) How can we design such a programmable hardware controller and generate its configuration data automatically from a given input loop program and schedule so that it generates control signals at right times to steer the program running on a PE? (e) Moreover, the control signals from the global controller have to be propagated to all PEs through PEs with appropriate delays so that every program running on the PE get orchestrated correctly. In this chapter, we try to provide the fundamental concepts and design of such a programmable GC hardware.
Figure 5.1: A processor class iteration space showing distinct program blocks, program block control flow graph, and the generated compact code for the FIR filter in Listing 3.3.
5. **Global Controller**

The main contributions of this chapter can be summarized as follows:

1. We elaborate on the concepts behind the GC, present the functionality algorithmically as well as systematically convert this into a hardware implementation, and finally discuss implementation details.

2. We discuss how the configuration data for the global controller may be generated within the compiler framework—automatically from the scheduled loop program.

3. We also discuss how the control signals from the GC are propagated to all PEs through PEs with appropriate delays and extracting these delays automatically from the given schedule.

The rest of the chapter is organized as follows: In Section 5.1, the architecture of the GC is presented. In Section 5.2, a detailed description of how the iteration space is scanned and its corresponding implementation details are discussed in brief. A detailed explanation of how to generate a set of control signals and its hardware implementation details are discussed in Section 5.3. Since the GC has to be programmable, a configuration scheme and the required hardware to program the internal registers in the GC is presented in Section 5.4. In Section 5.5, a brief overview of how to generate the GC configuration data within the compiler framework is discussed. Control signal propagation to all the PEs is discussed in Section 5.6. Related work is discussed in Section 5.7 followed by results of the global controller implementation in Section 5.8. Finally, the summary of this chapter is presented in Section 5.9.

### 5.1 GC: Architecture Overview

Figure 5.2 shows an architectural block diagram of the GC showing an Iteration Space Scanner (ISS) and a Control Signal Generator (CSG). The ISS is responsible for traversing the iteration space according to the pre-computed schedule and outputting the next iteration vector to be executed within a tile of iterations by using proper stride vectors. The CSG is responsible for generating the proper control signals which are used to control the proper branching in the assembly code of a processor. The functionality of the ISS itself is subdivided into a stride determinator and a stride selector functions which are discussed in the next section. A clock divider/mod-Iteration Interval (II) counter is needed since, the ISS has to be updated only once per iteration interval. In addition to these blocks, initializer, re-initializer, and control Finite State Machine (FSM) subblocks are needed, whose details will follow in the next sections. For purpose of better visualization these subblocks are not shown in Figure 5.2. On the other hand, the CSG functionality is subdivided into a Program Block Selector
5.2 Iteration Space Scanner

In the previous chapter, the Locally Sequential Globally Parallel (LSGP) partitioning was introduced, which transforms a given \( n \)-dimensional iteration space into a \( 2n \)-dimensional iteration space, \( \mathcal{I} \xrightarrow{\text{LSGP partitioning}} \mathcal{T}' \), given by Equation (3.4). Assuming a loop program with a two-dimensional iteration space as an example, the global allocation and scheduling after partitioning is given by the following space-time mapping:

\[
\begin{pmatrix}
    p_1 \\
p_2 \\
t
\end{pmatrix} = \begin{pmatrix}
    0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\lambda_1 & \lambda_2 & \lambda_3 & \lambda_4
\end{pmatrix} \cdot \begin{pmatrix}
    I_1 \\
I_2
\end{pmatrix}
\]  

(5.1)

where \( I_1 = (i_1 \ j_1)^T \in \mathcal{I}_1 \) and \( I_2 = (i_2 \ j_2)^T \in \mathcal{I}_2 \). Here, \((p_1 \ p_2)^T = (i_2 \ j_2)^T\) denote that tile origins also represent the coordinates of a processing element.
In general, for practical reasons, the dimensions of $I_2$ are limited to either 1 or 2 since either one or two dimensional processor arrays can currently be realized. However, all concepts can be extended for higher dimensions. The schedule vector $\lambda = (\lambda_1 \lambda_2 \lambda_3 \lambda_4)$ provides the start time of each iteration point $I$. The schedule vector can be further divided into two components $\lambda_{\text{intra-tile}} = (\lambda_1 \lambda_2)$ and $\lambda_{\text{inter-tile}} = (\lambda_2 \lambda_3)$ denoting the start times of iterations within a tile and start times of the tiles, respectively. In the LSGP-partitioning scheme, iterations within a tile, i.e., $I_1$ are executed by a processor sequentially and all the tiles are executed in parallel by a distinct processor $p = I_2$. For the control of execution of the iterations within a tile by a dedicated processor, obviously the value of $I_1$ needs to be computed for each time step $t$ of a schedule. Given a space-time mapping in Equation (5.1), it is in general not possible to determine the value of $I_1$ from $p$ and time $t$ since $\theta$ is not invertible [Dut11].

Therefore, an ISS that enumerates all the iterations within a tile is needed, i.e., $I_1$ depending on the tile size and given schedule $\lambda_{\text{intra-tile}}$. In this section, designing such an ISS to produce the values of the iteration variables $I_1$ assuming $n$ dimensions is discussed in detail. This ISS can be implemented efficiently using the concepts of path strides [Han09], defined in the previous chapter Definition 4.2.

Algorithm 5.1 summarizes the algorithm of ISS, given an iteration interval, stride vectors and stride condition spaces to determine the sequence $I(t)$ of iterations to be started for execution at time $t$. Also given is the first iteration $I_{\text{init}}$ to be started. In order to calculate the next iteration, an appropriate stride vector is added to the current iteration to obtain the next iteration. It is to be noted that the ISS computes a next iteration point only once per iteration interval, until the whole iteration space has been scanned. In Algorithm 5.1, $I_{\text{final}}$ denote the last iteration within the tile and multiplying it with the intra-tile schedule vector gives the last iteration start time or in other words, duration of the ISS in iteration intervals. In the following, an example is given to explain the Algorithm 5.1 in a more intuitive way.

**Example 5.1** Figure 5.3 shows the iteration space of a tile assigned to processor PE(0,0) of the FIR filter given in Listing 3.3. Let $I_1 = (i_1 j_1)^T$ denote the iterations within a tile. In this figure, the numbers shown inside the iterations denote the start times where iterations in the direction of $j_1$ are started first and later in the $i_1$ direction. Two path strides $s_1 = (01)^T$ and $s_2 = (1−7)^T$ are used to describe this scanning of the iteration space. Two distinctly colored rectangular boxes in Figure 5.3 denote the two stride condition spaces, one for each stride vector. In the stride condition space $0 \leq i_1 \leq 7 \land 0 \leq j_1 \leq 6$ stride vector $s_1 = (01)^T$ is added to the current iteration to obtain the next iteration whereas in the stride condition space $0 \leq i_1 \leq 6 \land j_1 = 7$, stride vector $s_1 = (1−7)^T$ is added to the current iteration to obtain the next iteration vector.
5.2 Iteration Space Scanner

Algorithm 5.1: Iteration Space Scanner.

**Input**: Stride vectors \( s_i \) and their corresponding stride condition spaces \( \mathcal{I}_{SCS} \), first scheduled iteration of the tile \( I_{init} \), last scheduled iteration point of the tile \( I_{final} \), and Iteration Interval \( (II) \).

**Output**: Iteration point sequence \( I(t) \) of iterations to be started at time \( t \).

```
1 begin
2 for \( \{t = 0; t \leq (\lambda_{intra-tile} \cdot I_{final}); t = t + II\} \) do
3 if \( (t == 0) \) then
4 \( I = I_{init} \)
5 end
6 else
7 \( I = I + \begin{cases} 
    s_1 & \text{if } I \in \mathcal{I}^1_{SCS} \\
    s_2 & \text{if } I \in \mathcal{I}^2_{SCS} \\
    \vdots & \vdots \\
    s_n & \text{if } I \in \mathcal{I}^n_{SCS}
\end{cases} \)
8 end
9 end
```

To be executed. As per Definition 4.2, the stride vector \( s_1 = (0 1)^T \) is added to the current iteration until iteration bound of the stride condition space is reached in the \( j_1 \) direction and stride vector \( s_2 = (1 -7)^T \) brings back the iteration to the left border of the iteration space. For this example, next iteration can be computed according to Algorithm 5.1 given the stride vectors and stride condition spaces as shown below:

\[
I = I + \begin{cases} 
    \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \text{if } I \in \{0 \leq i_1 \leq 7 \land 0 \leq j_1 \leq 6\} \\
    \begin{pmatrix} 1 \\ -7 \end{pmatrix} & \text{if } I \in \{0 \leq i_1 \leq 6 \land j_1 = 7\}
\end{cases}
\]  \tag{5.2}

In Figure 5.3, the start times which can also be calculated using the intra-tile schedule vector \( \lambda_{intra-tile} = (8 1) \) for an \( II = 1 \) as shown. The final iteration \( I_{final} = (i_1, j_1) = (7, 7) \) starts at time cycle 63 as shown in the figure and this also implies that the ISS and corresponding arithmetic operations require to compute one \( n \)-dimensional iteration vector \( I \) every \( II \) cycles which implies its design in hardware.
Figure 5.3: Iterations within a tile assigned to $PE(0,0)$ of FIR filter given in Listing 3.3. Numbers shown in the iterations denote the start times of iterations. Red colored arrows denote the path strides. The two distinctly colored rectangular boxes denote the stride condition spaces one for each stride vector.

### 5.2.1 Hardware Design of the Iteration Space Scanner

In this subsection, the design of a programmable Iteration Space Scanner (ISS) that may compute an iteration vector within II cycles according to Figure 5.3 is presented. First of all, we need comparator circuits that determines if $I \in \mathcal{I}_{scs}$ given the stride condition space $\mathcal{I}_{scs}$ and the current iteration $I$. This functionality is therefore divided into two blocks called stride determinator and stride selector. The stride determinator determines the stride condition space to which the current iteration belongs and generates a stride select signal. Based on this stride select signal, the stride vector corresponding to selected stride condition space is added to the current iteration to get the next iteration in the stride selector. It is to be noted that these two blocks can be implemented as a pure combinatorial logic. Furthermore, an initial iteration point $I_{init}$ is required for starting the ISS algorithm which is loaded in a register in the initializer subblock shown in Figure 5.4. The exact implementation details of the stride determinator are discussed in Appendix E.1.1, the stride selector in Appendix E.1.2, and the initializer in Appendix E.1.3.

In Algorithm 5.1, the next iteration is calculated at a time step $t$ based on the current iteration, which also implies that the stride determinator and stride selector blocks must compute a new iteration vector each iteration interval. For
this purpose, a control FSM is designed whose state transitions are controlled by the iteration interval. In the beginning of the algorithm, we start from an initial iteration and for each iteration interval next iteration is calculated until the final iteration is reached. Therefore, we propose a control FSM with an initial, run, and final state apart from a reset state. Further implementation details of the control FSM are given in Appendix E.1.6. As aforementioned, the states in this control FSM are controlled by the iteration interval. For instance, if the iteration interval is two cycles, states in the control FSM should change only for every two cycles. This is the same as saying that the next iteration is calculated only every two cycles which enforces the design to have a dynamic clock divider or mod-II counter based on the iteration interval. We have opted for a clock divider based solution and implemented it as a separate block, where a register in this block is loaded with the iteration interval. Based on this register value, an appropriate clock is generated. In Algorithm 5.1, for stopping the ISS, also the final iteration point \( I_{\text{final}} \) is required which is loaded in a register in the re-initializer subblock. The exact implementation details of the clock divider are discussed in Appendix E.1.4, and the re-initializer in Appendix E.1.5. In order to be programmable, all described blocks have a specific configuration interface using a configuration clock and a configuration bus—by which the internal registers in this blocks are programmed sequentially. This configuration scheme is explained in brief in Section 5.4. Figure 5.4 shows the architectural block diagram of the ISS.

### 5.3 Control Signal Generator

The main functionality of the control signal generator block is as shown in Figure 5.2 to generate the bundle of control signals that are used as inputs in the branch instruction of a program running on a PE to decide to which program block to jump next. The control signals in the branch instructions therefore need to be generated at exactly the correct times. In Section 4.5.2, while inserting the branch instructions in the generated code, a ROBDD was constructed for each program block to encode the branch targets. Even the computed transfer condition spaces have been annotated to the edges in the ROBDD. For the FIR filter example, such a ROBDD for \( PB4 \) is shown in Figure 5.5 for further illustration.

For \( PB4 \), as shown in Figure 5.5(a) there are two reachable program blocks \( PB4 \) and \( PB5 \). One control signal is therefore enough to encode these two program block branches. For instance, if this control signal is true, execution shall jump to \( PB5 \), else to \( PB4 \), which is also shown in instruction 8 in the generated code shown in Figure 5.1. Here, the most important question is that at which iteration \( I \) or time step \( t \) the corresponding control signal \( IC0 \) should be true or
false? The answer to this question lies in computing the exact transfer condition spaces and may be obtained from the ROBDD of all reachable program blocks. In case of \( PB_4 \), the transfer condition spaces to either \( PB_4 \) or \( PB_5 \) are shown in Figure 5.5(b). A ROBDD is constructed as shown in Figure 5.5(c)—in which the first scanning variable \( j_1 \) is sufficient for detecting the correct program block branch to. During this construction process, as explained in Section 4.5.2, the exact sub indexspaces of the transfer condition spaces that separate these two disjoint sets are identified. They are also shown as edge attributes, see Figure 5.5(c). These sub indexspaces of the transfer condition spaces are encoded as either true or false edge in the topologically similar ROBDD as shown in Figure 5.5(d), which is used in generating the final branch instruction as given below:

\[
\text{BRU IF IC0 JMP } PB_5 \quad \text{or: BRU IF IC0 JMP 9 8}
\]

It is worth noticing that the numbers under the braces denote the values of the control signal. Once the branch target lines (BL) are determined, placeholders of \( PB_5 \) and \( PB_4 \) will be replaced by absolute branch address, 9 and 8 respectively as shown above. The control signal generator hardware must therefore output control signals in dependence of the currently executed iteration \( I \) and the transfer condition spaces. In case of \( PB_4 \), \( IC_0 \) is false when \( 0 \leq j_1 \leq 5 \) and it must be true whenever \( j_1 = 6 \).
5.3 Control Signal Generator

In the following, we summarize the requirements of the control signal generator block for which we propose a hardware implementation next.

For determining the values of a set of control signals, the following computations are required:

1. First the current program block \( PB(I) \) needs to be identified. This part will be implemented in a block called Program Block Selector (PBS),

---

**Figure 5.5:** Program blocks reachable from \( PB4 \), corresponding transfer condition spaces, and ROBDD used in deriving the branch instruction for \( PB4 \) of the FIR example of Listing 3.3.
see Figure 5.2. Here, the iteration space of each program block is needed which is available from the program block control flow graph.

2. Once $PB(I)$ is determined for the current iteration, the ROBDD of the corresponding program block is of $PB(I)$ is consulted, which was constructed and used during code generation. The current iteration $I$ is then checked against the sub indexspaces of the transfer condition spaces and the control signals are emitted accordingly. This part of the functionality is implemented in the block IC Signal Generator (ISG) according to Figure 5.2. Here, the sub indexspaces of the transfer condition spaces are stored in hardware and available as edge attributes of the ROBDD of program block $PB(I)$.

For exact hardware implementation details of the CSG using the subblocks mentioned above, we refer to Appendix E.2. It is noteworthy that all circuits must provide outputs within a single clock cycle.

5.4 Configuration Scheme

In this section, we explain the configuration scheme for the entire global controller in brief. All the subblocks either in the ISS or CSG have a configuration bus and a configuration clock interface by which various registers inside these blocks get programmed. All these hardware blocks are programmed one after the other using a small FSM called loader FSM, running synchronously with the configuration clock. It is assumed that the configuration data is present in a configuration memory and it is read word by word using another FSM—attached to the configuration memory—supplying data word by word to the loader FSM. Configuration data generated in the compiler is loaded into the configuration memory of the GC through a LEON core. We discuss the implementation of the configuration scheme in detail in Appendix E.3.

5.5 Configuration Data

In the previous section, it is assumed that the configuration data is available which can be programmed through a LEON core into the configuration memory. Later on, hardware of the configuration scheme takes care of putting the data onto the configuration bus and programming different subblocks. In this section, a brief overview of how this configuration data is generated is discussed. To recapitulate, as mentioned in Chapter 3 and according to Figure 3.4, a nested loop is scheduled with the given resource constraints in the front-end of the compiler. From the loop specification itself, for instance, initial iteration and
5.6 Control Interconnection Network

The control signals generated from the global controller need to be propagated with appropriate delays to each PE in a linearly staggered fashion so that programs running on each PE get orchestrated according to the inter-tile schedule $\lambda_{inter\_tile}$ by which PEs are delayed to start execution. The PE which starts executing first, will receive control signals directly from the global controller whereas all the other PEs get the same control signals in a delayed fashion. The number of delay registers for a processing element $p$ is equal to the number of time steps required by the global controller signals to travel to processing element $p$, i.e.,

$$\Delta_d = \lambda_{inter\_tile} \cdot p$$  \hspace{1cm}(5.3)$$

Equation (5.3) suggests that the control signals must be synchronized with the start time of the processing elements of the first iteration executed on each PE. By exploiting, the regularity of the processor array and in order to reduce the amount of delay registers, we propose that control signals issued from the global controller are also propagated through the processor array similar to data signals. In other words, each PE shall receive its control signals from a
neighboring PE, \((p - d_p)\), where the propagation vector \(d_p\) denotes a link or interconnection to a neighbor PE.

Here, as a first step, one needs to determine the appropriate subset of propagation vectors, denoted by \(D_p\), from a set \(L\) of all possible propagation vectors \(d_p\)—given a processor space \(\mathcal{P}\)—in a way that each \(p \in \mathcal{P}\) can be reached from the PE that starts executing first. For example, in case of a two dimensional array \(L\) consists of propagation vectors \(\{(1, 0), (0, 1), (1, 1), (-1, 0), (0, -1), (-1, 1), (1, -1), (-1, -1)\}\) and we need to determine exactly two propagation vectors from this set. Whereas in case of a one-dimensional array, only one propagation vector needs to be determined. It also implies that these number of propagation vectors are dependent on the array dimension and we need to determine exactly \(\dim\{\mathcal{P}\}\) propagation vectors, i.e., \(\dim\{\mathcal{P}\} = |D_p|\). In the following, we formulate two conditions to determine these \(\dim\{\mathcal{P}\}\) propagation vectors of set \(D_p\).

### Condition I

Reachability of each PE by a linear combination of appropriate propagation vectors:

\[
\begin{pmatrix}
p_1 \\
p_2 \\
\vdots \\
p_{\dim\{\mathcal{P}\}}
\end{pmatrix} = (d_{p_1}) \delta_1 + (d_{p_2}) \delta_2 + \ldots + (d_{\dim(\mathcal{P})}) \delta_{\dim(\mathcal{P})}, \quad \delta_i \in \mathbb{Z}_0
\]

### Condition II

For each \(d_{p_i}\) selected, it must hold that the delay between two interconnected PEs is strictly non-negative (or even strictly positive):

\[
\Delta_{d_{p_i}} = \lambda_{\text{inter tile}} \cdot d_{p_i} > 0
\]

**Lemma 5.1** In order to satisfy Condition I, at least \(\dim\{\mathcal{P}\}\) linearly independent vectors are necessary.

**Proof.**

\[
p^T = \begin{pmatrix} D_p := \\ d_{p_1}^T \\ d_{p_2}^T \\ \vdots \\ d_{\dim(\mathcal{P})}^T \end{pmatrix} \cdot \delta
\]

In order to reach all PEs, \(D_p\) must have full rank. This also implies, at least \(\dim\{\mathcal{P}\}\) vectors must be selected and in a way to be linearly independent. \(\square\)
Lemma 5.2 Choose any \( \dim \{ \mathcal{P} \} \) linearly independent vectors \( d_{pi} \) from \( L \) such that satisfy Condition II.

Example 5.2 In Figure 5.6, we have \( \lambda_{\text{inter-tile}} = (57 \ 8) \). Hence, due to Condition II, we may not use the candidates \( d_{p1}^T = (-1 \ 0)^T \) and \( d_{p2}^T = (0 \ -1)^T \) as \( \Delta_{dp_i} < 0 \).

Example 5.3 Choose \( d_{p1}^T = (1 \ 0)^T \) and \( d_{p2}^T = (0 \ 1)^T \). Hence, both Conditions I and II are satisfied.

Example 5.4 Choose \( d_{p1}^T = (1 \ 1)^T \) only. Hence, Condition II is satisfied but not Condition I, as we may not reach processor \( (p_1 \ p_2)^T = (1 \ 0)^T \) using \( d_{p1}^T \), i.e.,

\[
\begin{pmatrix} p_1 \\ p_2 \end{pmatrix} \neq \begin{pmatrix} 1 \\ 1 \end{pmatrix} \cdot \delta
\]

In a 2-D mesh architecture, one may simply use the either positive or negative unit vector for propagation depending on Condition II.

In the second step, once the subset of appropriate propagation vectors \( D_p \) is found, the number of delay registers for a propagation vector \( d_{pi} \) in this set is given by \( \Delta_{dp_i} = \lambda_{\text{inter-tile}} \cdot d_{pi} \). However, we have to choose only one propagation vector \( d_p \) which minimizes the length of delay registers for a PE. For \( (p_1 \ p_2)^T = (3 \ 3)^T \) in Figure 5.6, we choose \( d_{p2}^T = (0 \ 1)^T \) instead of \( d_{p1}^T = (1 \ 0)^T \). The pseudo code in the following Algorithm 5.2 chooses the appropriate propagation vector and computes the delay registers in a array size independent way.

Algorithm 5.2: Determining propagation vector \( d_p \) and number of delay registers \( \Delta_{dp} \).

\[
\begin{align*}
\text{Input:} & \quad \text{Processor Classes } PC, \ 	ext{inter-tile schedule vector } \lambda_{\text{inter-tile}}, \ \text{propagation vector set } D_p. \\
\text{Output:} & \quad \text{Propagation vector } d_p, \ 	ext{delay registers } \Delta_{dp}.
\end{align*}
\]

1. foreach \( PC \in PC \) do
   2. 1). Let \( p \) be the first PE executed in the \( PC \).
   3. 2). foreach \( d_p \in D_p \) do
      4. 3). Select \( d_p \) such that \( \Delta_{(p-d_p)} < \Delta_p \) and \( \Delta_{(p-d_p)} \geq \Delta_{(p-s)} \forall \Delta_{(p-s)} < \Delta_p \), where \( s \in D_p \).
      4. 4). Delay register length \( \Delta_{dp} \) is given by \( \lambda_{\text{inter-tile}} \cdot (d_p) \).
   5. 5). For the rest of the PEs in the \( PC \) use the same \( d_p \) and \( \Delta_{dp} \).
end
5. Global Controller

Figure 5.6: Propagation of control signals from the global controller to all the processors in a 2-D processor array using appropriate control propagation vector $d_p$ and appropriate delay registers $\Delta_{d_p}$.

As an example, the control network interconnection scheme for the FIR filter of Listing 3.3 mapped onto a $4 \times 4$ processor array is shown in Figure 5.6. For $PE(0, 0)$, control signals are taken from the global controller directly. For the rest of the PEs in the array, control signals are propagated from neighbor PEs as shown in Figure 5.6 with appropriate delays. For instance, in the case of $PE(0, 1)$ and $PE(1, 0)$ control signals are propagated through $PE(0, 0)$ with delay registers of length 8 and 57, respectively using the two selected propagation vectors $(0 1)^T$ and $(1 0)^T$. These delay register values can be calculated as follows:

$$\Delta_{(01)^T} = \begin{pmatrix} 57 & 8 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 8$$
\[ \Delta_{(10)^r} = \begin{pmatrix} 57 & 8 \\ 1 & 0 \end{pmatrix} = 57 \]

### 5.7 Related work

The execution of loop nests may incur a significant overhead, which is due to increment and branch instructions to initiate a new iteration of the loop. To confront this problem in embedded computing either in the form of general-purpose processors or Digital Signal Processors (DSPs), the use of customized loop controllers has been suggested. In literature, many code-improving transformations to decrease execution time at the expense of significant increase in code size and power consumption [BHPS00, BHK+97] exist. One such transformation is loop unrolling, which decreases the loop overhead at the expense of code size [DJ96]. However, Reduced Instruction Set Computer (RISC) processors or DSP processors are typically used for applications in embedded systems that have strict code size and power limitations. Space increasing transformations such as loop unrolling are often unacceptable for many applications due to these limitations.

Considering architectural approaches, there are two different approaches to alleviate the looping overhead. The first one involves the use of a small instruction buffer or cache, also called Zero Overhead Loop Buffer (ZOLB) [UWW+99], which holds the frequently executed loop programs. This buffer can be used to increase the speed of applications with no increase in code size and often with reduced power consumption. In the first pass of a loop, instruction from the main memory are fetched, decoded and copied to the loop buffer, which are also used in the subsequent passes of the same loop. When executing from the loop buffer, the program memory and instruction decoder can remain idle, which has a positive impact on the system power consumption. In addition, the same memory bus used to fetch instructions can sometimes be used to access data when certain registers are dereferenced. Thus, memory bus contention can be reduced when instructions are fetched from a ZOLB. A compiler or assembly writer attempts to execute many of the innermost loops of programs from this buffer. A ZOLB can be viewed as a compiler controlled cache whose state is preserved and special instructions are used to load instructions into it. This technique is already proposed in [LMA99, WHH06] where a buffer cache fill mechanism is controlled by either a state machine for detecting a single loop [LMA99] or a stack based controller supporting nested loops [WHH06]. In general, this technique is not able to reduce the total execution time, since the branch instruction cannot be surpassed. However, the power consumption is reduced due to the small capacitance switched when the internal buffer is accessed.

The second type of approaches reduce the cycle overheads due to looping by using zero-overhead loop instructions that are specialized hardware units
5. **Global Controller**

that may update iteration variables and switching between loops. This concept can provide means for a better performance with expected gains in the system energy consumption. This approach is often found in DSPs [UWW+99, DSP95] where hardware mechanisms are provided for zero overhead switching between loops. In [Bom02], a microprogrammed control unit that accounts for nested loops is presented. The Motorola DSP56300 [DSP95] supports seven levels of nesting using a system task. There is a five-cycle overhead to prepare a loop for this type of hardware control.

Another architectural enhancement proposed in [TJB03] and [Tal01]. In these works, a hardware unit also called *MediaBreeze* is used to handle loop nesting up to five levels. It also contains hardware support for efficient address generation, looping, and data organization (permute, packing/unpacking, transpose, etc.). Basically, this architecture facilitates the Single Instruction Multiple Data (SIMD) computation unit, which is attached to a general-purpose processor, to achieve high throughput and efficiency where conventionally overhead/supporting instructions dominate the dynamic instruction stream. To eliminate branch instruction overhead, MediaBreeze employs a zero-overhead branch processing using a dedicated hardware loop control unit. All branches related to loop increments (based on indices used for referencing data) are handled by this technique. This is done in many conventional DSP processors such as Motorola DSP56300 [DSP95] and TMS320C5x from Texas Instruments [LBSL98]. The overall energy consumption of the MediaBreeze architecture is less than that of a superscalar processor with SIMD extensions because the Breeze instruction reduces the total dynamic instruction count. Once a breeze instruction is interpreted, the instruction fetch, decode, and issue logic in the general-purpose processor can be shutdown to save power. Its main advantage is that the successive last iterations of the loop are performed in a single cycle and it support fully nested loops. The area of the hardware units grows in proportion to the considered number of loops.

Configurable processors such as Xtensa [Gon00] incorporate zero-overhead mechanisms for just one innermost loop. Unfortunately, a fixed processor template is provided to which alternate control flow mechanisms cannot be added. In [KN08], an architectural solution for Zero Overhead Loop Controller (ZOLC) is presented, which is used at the instruction fetch stage to eliminate the loop overheads and can be applied to arbitrarily complex loop structures. A portable specification of a ZOLC unit is also provided, which can be utilized for the automatic generation of the aforementioned enhancements to different embedded processors. In addition, a region-based task control flow formalism for software synthesis of ZOLC initialization and configuration is given and a practical compilation flow for ZOLC code generation has been implemented based on open source tools. This ZOLC unit has been incorporated to various RISC processors: xiRisc [CCG01], a 32-bit configurable microprocessor, distributed as a VHDL
softcore, as well as to Architectural Description Language (ADL) models of other processors (e.g., MIPS R3000). The benefit from using ZOLC is that the increment and compare-and-branch operations can be removed and replaced by a task switching operation associated to the instruction fetch of the last useful instruction in the inner loop. When a ZOLC architecture is incorporated in the control path of a typical RISC processor, its purpose is to provide a proper candidate Program Counter (PC) target address to the PC decoding unit for each substituted looping operation. Typically, the design of the instruction decoder, the PC decoding unit and the register file architecture require modifications in presence of the ZOLC hardware. However, to support the potential use of the ZOLC unit, proper development tools are required in order to ease code generation. In [KM10], an architectural approach to designing efficient parametric hardware looping units (HWLUs) targeted to Field-programmable Gate Arrays (FPGAs) is presented that provide zero-cycle overhead implementations for perfect loop nests.

Related work mentioned so far is targeted towards general-purpose processors such as (ARM, MIPS32) and DSP architectures featuring architectural characteristics suitable to portable platforms. In these processors, the most performance-critical computations occur in various forms of nested loops whose overhead can be alleviated by aforementioned architectural enhancements. In case of mapping nested loop programs onto processor arrays such as TCPAs, partitioning is applied which tackles the problem of hardware matching, i.e., mapping the given iteration space onto a fixed-size processor array. However, naturally partitioned programs have a more complex control flow (especially, if the original loop has loop-carried data dependencies) and requires the design of efficient controllers to orchestrate the correct execution of the program on a processor array. The systematic design of control units of full-size processor arrays was first introduced in [TT91] and [Xue92]. However, both methodologies are restricted to simple space-time mappings obtained by a projection. The authors in [DSRV02] introduced a method for the automatic generation of control code in case of clustering with rectangular tiles and tight linear schedules. In [DHT06a], the authors introduced a general methodology for control path generation on hierarchical loop partitioning with congruent tiles with maximum reuse of control predicates in massively parallel architectures. However, the methodology is computationally expensive due to modulo and division operators in the global controller and counter. In [DHRT07], a new method for generating the global counter which avoids computationally expensive operators and thus reduces significantly the hardware costs compared to [DHT06a] is proposed. However, both methods proposed in [DHT06a] and [DHRT07] synthesize a dedicated controller for each application and lack any programmability. The global controller approach proposed in this dissertation is similar to [DHRT07], targeting rectangular iteration spaces. However, for the first time, a fully recon-

173
5. Global Controller

A configurable controller whose configuration data is generated automatically within the compiler framework has been developed.

5.8 Implementation Results

In this section, we discuss implementation results of the global controller. In particular, we focus on area, performance, and power in dependence of the tile dimension $n$. Throughout these evaluations, the XC5VLX330 device (FF1760 package and "-1" speed grade), which is one of the largest available Virtex-5 devices is considered as a target. The capacity of XC5VLX330 is 51,840 slices (which translates to 207,360 6-input LUTs and Flip-Flops), 576 18-Kb block RAMs (BRAMs) and 192 DSP48E data path blocks. Both BRAMs and DSP48E blocks remain unused by the global controller. In order to report area numbers, Synopsys’ Synplify Premier and for power numbers, Xilinx’ Xpower is used.

5.8.1 Area Consumption

Figure 5.7 shows the required area of the GC in terms of the required percentage of LUTs and slice registers for different tile dimensions. With increasing number of dimensions, the area of the GC increases as expected since with each extra dimension, additional logic is added in the hardware blocks of the controller. However, with up to $n = 10$, the area requirement is around 14% of the target device. It is worth noticing that with increase in $n$, the LUT usage is more compared to the slice register usage. This behavior is as expected since an extra dimension adds more combinatorial logic compared to the registers which get translated into LUTs and slice registers, respectively. The area overhead of the global controller might be more compared to the TCPA itself for smaller and light-weight array sizes but for larger array sizes it will come down drastically since the same controller is used for all the other additional processors. Figure 5.8 shows the normalized area of the GC w.r.t. to a single PE having 1 adder, 1 multiplier, 5 DPUs, 64 instructions, 16 RD, and 10 I/O registers. Both in Figures 5.7 and 5.8, while calculating the area of the GC, a fixed maximum number of nine program blocks and one control signal are considered. However, an increase in either the number of program blocks or the number of control signals will also increase the area of the GC. It is to be noted that for each application mapped onto a TCPA, a dedicated controller is needed.
5.8 Implementation Results

**Figure 5.7:** Area of the global controller for different tile dimensions $n$ for a fixed maximum number of nine program blocks and one control signal.

**Figure 5.8:** Normalized area of the global controller w.r.t. to a single PE for different tile dimensions $n$. 

175
5.8.2 Performance Measurements

In Figure 5.9, the variation of the maximum achievable clock frequency with varying number of tile dimensions is shown just for the GC. The clock frequency drops from 122 MHz to 85 MHz with an increase in the tile dimensions from 2 to 10. The critical path obviously goes through the stride determinator, stride selector, and control signal generator hardware blocks. The frequency of the global controller can be further improved by some architectural changes. For instance, in the current architecture, the stride determinator block determines the stride condition space to which the current iteration belongs to. Based on this information, a stride vector corresponding to the stride condition space is added to get the next iteration vector to be executed. However, the stride determinator and the stride selector blocks can both work in parallel, i.e., all possible stride vectors can be added to the current iteration to get all the possible next iteration vectors in stride selector block while the current iteration’s stride condition space is determined in the stride determinator block. Finally, based on the stride condition space information—from all the possible next iteration vectors—the next iteration vector which is already calculated based on the stride vector corresponding to the selected stride condition space is chosen. This optimization definitely improves the timing at the expense of increase in the area of the stride selector block as all additions are performed in parallel, which obviously requires more resources.

5.8.3 Power Consumption

Figure 5.10 shows the average power requirements of the global controller with increasing number of tile dimensions. The average power obviously increases with increasing number $n$ of dimensions even though the clock frequency drops since the area increase (LUTs and slice registers) is higher compared to the drop in the clock frequency. While calculating the average power, a default switching factor of 20\% has been considered.

The amount of configuration data required and configuration time of the global controller need lot of implementation details such as number of configurable registers in the subblocks, etc. Therefore, we refer to Appendix E.4 for more details.

5.9 Summary

In this chapter, the importance of the control signals generated from a global controller in orchestrating the program executing on each processor in a TCPA was emphasized. Few potential questions were raised which lead to a fully programmable global controller architecture consisting of two main components:
the iteration space scanner and the control signal generator. The iteration space scanner enumerates the iteration space according to a given schedule while the control signal generator generates a control signal bundle based on the current iteration and the current program block executed. Subsequently, an algorithm to enumerate the iteration space was presented which is converted into hardware in a step-by-step approach consisting of different hardware subblocks such as stride determinator, stride selector, initializer, clock divider, reinitializer, and control FSM. The functionality of all these blocks was explained. Later on, the control signal generator block implementation was explained with emphasis on how the control signals were generated with the help of sub indexspaces of transfer condition spaces among the program blocks in the control flow graph, which was used in generating the code for a processor. The configuration scheme of the global controller as well as how to generate the configuration data of the global controller either in the front end or back end of the compiler was also discussed. Furthermore, a concept to realize a control interconnection network was presented where the control signals emanating from the global controller were propagated in a linearly staggered fashion to all the processors in the array with appropriate delays. Finally, related work and implementation results of the global controller such as area, clock speed, and power were discussed. To
Figure 5.10: Average power consumption of the global controller with varying number of tile dimensions $n$.

conclude, our concept of global controller is novel and appealing which makes the generated code independent of the problem size. For the first time, a fully reconfigurable zero-overhead loop controller whose configuration data is generated automatically within the compiler framework has been developed.
Address Generators

In the last decade, the ever increasing demand for intelligent systems has lead to an explosion of the embedded system market which now represents more than 98% of the world wide production of microprocessors [Tur06]. These systems are dominated by multimedia applications such as video and image processing which are often characterized by a large number of data accesses. These applications require high performance specific computations that general purpose processors cannot deliver at a reasonable power consumption. Programmable accelerators with Very Long Instruction Word (VLIW) architecture as discussed in this thesis may satisfy the computational needs at low power by exploiting Instruction-Level Parallelism (ILP) and Loop-Level Parallelism (LLP). With this increasing parallel computation, the access to data is becoming the main bottleneck that limits the available parallelism. This conventional wisdom is also known as memory wall in literature. Accelerators also need to address this problem in order to hide the memory latency. Therefore, the data must be stored in local buffers, memory banks or scratchpad memories close to the accelerators.

Moreover, memory access have to be parallelized to provide enough data to the processors so that they are not starved in terms of data. For this purpose, special hardware units working in parallel to the Processing Elements (PEs) and the Global Controller (GC) are necessary and will be called Address Generators (AGs) in the following. These are used to generate the right address sequences for accessing (read/write) the data from/to the local memory. In typical embedded processors like a LEON3 or ARM, the address generation is done by issuing an address in the form of memory read/write instructions over a bus interface. This gives extreme spatial and temporal flexibility by allowing access to random locations (spatial flexibility) inside the memory at random time (temporal flexibility). The drawback of such an approach is that contention and the bandwidth bottleneck requiring a serialization of memory and compute operations. Furthermore, memory address computations can significantly decrease the performance and increase the power consumption. For instance, 50%-70% of the power consumption in an embedded multimedia system is due to memory access alone [WCNDM96, MNCDM97]. Thus, most of the modern Digital Signal Processors (DSPs) like the TMS320C55x [Ins02] and many other VLIW
accelerator architectures employ AGs to ensure efficient feed/storage of the data and to parallelize the memory operations as well.

In the case of a Tightly Coupled Processor Array (TCPA), PEs do not have any direct access to a global memory. In order to hide the memory latencies in a TCPA, local buffers are provided surrounding the array. Data transfers to and from these memories are therefore performed through the border PEs only, see Figure 6.4 for illustration. These buffers can be configured to either work as simple FIFOs or as RAM-based addressable memory banks. Configurable AGs needed to be developed to generate the address sequences for these buffers, again adhering to a global schedule. However, the address generation pattern depend not only on the implementation style of a particular algorithm but also on the way data is stored in the memory which in turn depend on the schedule and partitioning scheme used. Moreover, the number of address generators and their address sequences depend on the given loop program itself. Here, a plausible question is: How can we determine the address sequence and enable signals for each input and output buffer based on a given loop program and schedule? Therefore, programmable address generators are needed, which can be configured and instantiated multiple times depending on the requirements.

The main contributions of this chapter can be summarized as follows:

1. We focus mainly on the design of programmable AGs and generating their configuration data automatically from a given scheduled and partitioned loop program.

2. We review and classify different address generator models and address generator architectures found in the literature.

The rest of the chapter is organized as follows. In Section 6.1, modeling of AGs and their classification are briefly discussed. Different address generation requirements considering the TCPA architecture are introduced in Section 6.2. A design space exploration with respect to different AG architectures found in the literature is presented in Section 6.3. Hardware implementation of selected AGs and their comparison in terms of area, performance, and average power consumption is discussed in Section 6.4. Extracting the configuration data for the AGs from a given loop specification is presented in Section 6.5 followed by related work in Section 6.6. Finally, this chapter ends with a brief summary in Section 6.7.

6.1 Address Generator (AG) Model and Classification

In general, programmable architectures exploiting parallelism often follow the VLIW paradigm where the Functional Units (FUs) running in parallel follow
6.1 Address Generator (AG) Model and Classification

A pre-computed schedule [Leu00]. These architectures focus on real-time performance and often deal with infinite, continuous streams of data. AGs work in parallel with the main computational units or processors to ensure efficient feed/storage of data from/to the main memory. Although access time and parallel access constraints to memory exist, an important problem is also the efficient generation of address sequences for a given application. Such a address sequence can be generated from an address equation which is defined below.

Definition 6.1 (Address Equation (AE)) An Address Equation (AE) is a function extracted from the description of an algorithm where the parameters are iteration vectors of a nested loop program as given in Equation (6.1). The address equation is only executed conditionally for a subset of iteration vectors \( I \). This is expressed by a condition space also called Address Condition (AC) which specifies for which iteration vectors \( I \) the generated addresses are valid. Typically, this condition space defined by a polytope as given below in Equation (6.2).

\[
\begin{align*}
\text{(AE)} : \quad \text{Addr} &= f(I) \\
\text{(AC)} : \quad A \cdot I &\geq b \quad \text{where} \quad A \in \mathbb{Z}^{r \times n}, \quad I = (i_1 \ i_2 \ldots \ i_n)^T \in \mathbb{Z}^{n \times 1} \quad \text{and} \quad b \in \mathbb{Z}^{r \times 1}
\end{align*}
\]

In a broader sense, the task of an AG is to compute an address sequence—given a sequence of iteration vectors \( I \) such as from scanning a polyhedral domain and computing these addresses using Equation (6.1) while checking additionally for each vector \( I \) using Equation (6.2) whether the address is valid or not. Furthermore, this AC can be used as a chip select/enable signal in the case of implementing the buffers using Random Access Memory (RAM) or FIFOs. The design of a hardware unit to compute address sequences according to Equation (6.1) depends on the considered class of function \( f \) to be supported according to Equation (6.1). Therefore, in the following we present the classification of address equations.

6.1.1 Address Equation Classification

Depending on the type of application an address equation can be classified in terms of its regularity or flexibility [TJCC08]. The regularity of the AE expresses the complexity of the index expression whereas the flexibility gives an idea of the elasticity needed by the AG hardware.

**Affine AE:** An AE said to be in affine form when it can be expressed as a linear combination of iteration variables \( i_j \) and constants \( c_j \) as given below.

\[
\begin{align*}
\text{AE} : \quad \text{Addr}(I) &= c_1 \cdot i_1 + c_2 \cdot i_2 + \ldots + c_n \cdot i_n + c_0
\end{align*}
\]
6. Address Generators

For instance, an affine AE could be the indexing function of a Piecewise Regular Algorithm (PRA) as introduced in Section 3.1.1.

\[ x_i[I] = F_i(\ldots, x_j[CI - C_0], \ldots) \]

For example, \( x[i] = y[3i - 2] \) with \( c_0 = 2 \), and \( c_1 = 3 \).

**Piece-wise Affine AE:** An AE is called piece-wise affine AE when it can be written as linear combination of indexes and constants in each part of the region where an address sequence has to be generated. This is the most general case of an AE in a nested loop with conditional statements or condition spaces based on the iteration variables as shown in the below example.

\[ x[i] = y[4i - 2] \quad \text{if} \quad i \leq 4 \]
\[ x[i] = y[2i - 1] \quad \text{if} \quad i > 4 \]

**Non-linear AE:** This is the most general case of an AE where \( f \) may be a non-linear function of \( I \). As an example, \( x[i] = y[i^2] \) would be an equation with a non-linear address function for variable \( y \).

**Predefined with Constants:** This kind of AEs have predefined address sequences which are known prior to the hardware synthesis. This requires the knowledge of the program and enables the optimized construction of the AG.

**Predefined with Parameters:** In this kind of AE, address sequences are controlled by a few parameters. This kind of AE is encountered in domain specific designs where the AG has to support a limited number of applications.

**Dynamic Address Equation:** In this kind of AE, the address sequence depends on its own computed results and external events. Here, the address sequence may only be constructed at runtime.

Figure 6.1 summarizes and shows different possible combinations of regularity and flexibility of the AE. The complexity of building an efficient AG increases as we move away from the origin of the graph. In this dissertation, we focus mainly on affine equations (see Equation (3.3)) and an AG with parameters since most of the applications targeted to TCPA fit well into this criteria.

### 6.1.2 AG Classification

There is a wide range of AGs that we can find in the literature which are mainly categorized into two types of AG architectures, the ones based on tables and the ones based on a synthesized data path.
6.1 Address Generator (AG) Model and Classification

![Diagram of Address Generator Classification]

**Figure 6.1:** Design space of AG based on the regularity and flexibility of the AE. (source [TJCC08]).

**Table-based AGs**

This type of AG architecture uses a Look-Up Table (LUT) to implement the AE. All the possible values of the AE are computed before the hardware synthesis and stored in the LUT. An additional controller that can be either a simple counter or a Finite State Machine (FSM) is used to generate the address sequence in a deterministic way. This architecture is more suitable for simple and short (not a lengthy address sequence) AE [TJCC08]. The size of the LUT increases with the complexity of the AE. The following paragraphs describe the two main types of table-based AGs which can be implemented as either programmable or custom logic.

**Memory based AG:** The address sequences are directly mapped into a LUT and a counter is used to control the address generation. The block diagram of such an architecture is shown in Figure 6.2(a). This AG architecture can be applied to any type of AE but the address sequences have to be computed off-line. However, the address sequence has to be short else the size of the LUT becomes too big [TJCC08].

**Incremental AG:** If the AE is very regular and can be expanded in sequences of the address values, then it can be implemented by modulo/binary counters with
6. Address Generators

![Diagrams](a) Block diagram of a memory-based AG. (b) Block diagram of an incremental AG, adopted from [TJCC08].

**Figure 6.2:** Table-based AGs

The output modified by custom logic that can be implemented in a programmable logic array (PLA) as shown in Figure 6.2(b). This approach has smaller size compared to the memory based AG implemented with look-up table [TJCC08].

**AGs based on a Data Path**

If the AG has to support control flow or programmability, data-path-based AGs are the most suitable ones. Here, the address values might depend on its previous values, control signals, and other external conditions. This type of AG implementation covers a wide range of AE functions and can be reconfigured at runtime.

**CPU-based AG:** In this type of AG architecture, a small Central Processing Unit (CPU) is used to generate the address sequence. This CPU works like a co-processor and has a reduced instruction set and a relatively small instruction memory, which can be partially or completely reloaded at runtime. The address generation flow can be controlled by flags and control signals coming from external units. CPU-based AGs are the most suitable implementation for a complex AE with a long address sequence. In such cases, the address sequence is implemented as a small program, which is then loaded into the instruction memory of the CPU. While the flexibility of this architecture is very high but also the cost of the design in terms of area and power consumption is very high.

**Custom AG:** If the AE is an affine function of loop indices, it can be mapped directly onto arithmetical operators and thus can be implemented in custom hardware. This type of AEs are encountered in nested loop programs where the loop indices are combined to generate the address at each iteration of the loop program. A custom AG is an optimal architecture for regular memory accesses.
6.2 TCPA: Address Generation Overview

To recapitulate, nested loop programs written according to the notation of PRAs in PAULA language are mapped onto TCPA as described in Section 3.1.1. Here, the address equations are affine functions of the indices which are given by the index function of input/output variables specified in the algorithm as these will be mapped to the border of the processor array. Furthermore, an address condition which controls when to read/write from input/output variable is given by the condition space associated with that statement. This condition space denotes the part of the iteration space where the statement corresponding input/output variable is defined. The addresses as well as address enable conditions generated in the AGs will be used to address the surrounding buffers of the TCPA array as shown in Figure 6.4. When a given algorithm is executed according to the pre-computed schedule, the GC generates the corresponding index or iteration vector at each iteration which is forwarded to the AG. The addresses and address enable conditions are generated based on the iteration vector in the AG.
In the following, we explain the address equation and address condition with the help of a partitioned FIR filter program given in Listing 3.5.

**Example 6.1** In Listing 3.5, a FIR filter with $N = 32$ taps is scheduled for $T = 32$ samples and partitioned onto $4 \times 4$ processor array using the Locally Sequential Globally Parallel (LSGP) partitioning scheme. Therefore, iteration variables $i_2$ and $j_2$ define the processor space. Iteration variables $i_1$ and $j_1$ denote the iterations within a tile or a PE. It is to be noted that the global controller scans this tile iteration space and generates control signals as well as intra-tile iteration vectors as described in Chapter 5. Now, by considering the following statement from the partitioned program, it is explained how to deduce the address equation and address condition for input variable ‘a_in’.

\[
\begin{align*}
&\text{par}(0 \leq i_1 \leq 7 \text{ and } 0 \leq j_1 \leq 7 \text{ and } 0 \leq i_2 \leq 3 \text{ and } 0 \leq j_2 \leq 3) \\
&\cdots \\
&a[i_1, j_1, i_2, j_2] = a_{\_in}[i_1, j_1, i_2, j_2] \text{ if } (i_1 == 0 \land i_2 == 0); \\
\end{align*}
\]

(6.4)

In the above equation, consider the condition space condition $i_2 = 0$. It implies that an input ‘a_in’ is read by all the processors for which $i_2 = 0$, i.e., processors $\text{PE}(0, 0), \text{PE}(0, 1), \text{PE}(0, 2),$ and $\text{PE}(0, 3)$, see Figure 6.4. More details to follow in Section 6.5. Now, the first condition in the condition space of Equation (6.4) is $i_1 = 0$, which depends only on the intra-tile iterations, denotes the address condition. The address equation to be considered for each of the processing element reading $a_{\_in}$ is given by an index expression of only the intra-tile iteration variables $i_1 \cdot N_1 + j_1$ or simply $j_1^{13}$. Similarly, the address equation and address condition space for all the other input/output variables can be deduced and are given below:

\[
\begin{align*}
&A_{\text{E}_{a_{\_in}}} : \quad \text{Addr}_{a_{\_in}} = i_1 \cdot N_1 + j_1 \quad \text{or} \quad j_1 \\
&\quad \text{AC}_{a_{\_in}} : \quad i_1 = 0 \\
&A_{\text{E}_{u_{\_in}}} : \quad \text{Addr}_{u_{\_in}} = i_1 \cdot N_1 + j_1 \quad \text{or} \quad i_1 \\
&\quad \text{AC}_{u_{\_in}} : \quad j_1 = 0 \\
&A_{\text{E}_{y_{\_out}}} : \quad \text{Addr}_{y_{\_out}} = i_1 \cdot N_1 + j_1 \quad \text{or} \quad i_1 \\
&\quad \text{AC}_{y_{\_out}} : \quad j_1 = 7 \\
\end{align*}
\]

(6.5)

It is to be noted that for instance for input variable ‘u_in’, there are two possible AEs, which one to use depends on the how the data is organized in the memory.

---

13It is worth noticing that a two-dimensional iteration space is mapped one-dimensional memory space so the boundary of the iteration space in the direction of $j_1$ has to be taken into account in the address equation. Which bound to take into account also depends on the way the iteration space is scanned, for instance either row major order or column major order determined by the loop matrix as mentioned in Definition 3.5.
Figure 6.4: Mapping of partitioned FIR filter given in Listing 3.5 onto a $4 \times 4$ processor array. One address generator is allocated and configured for each I/O variable name of a loop. Data fed through RB located around the array. Address and enable signals are propagated in a delayed fashion to different buffers corresponding to an input or output [HSL+13]. The global controller is also shown including the control signal propagation and how the iteration vector is driving the AGs.
6. Address Generators

Here, the input ‘\textit{u\_in}’ also read by all processors for which $j_2 == 0$, i.e., PE(0, 0), PE(1, 0), PE(2, 0), and PE(3, 0). If the data for ‘\textit{u\_in}’ is stored in consecutive locations then $i_1$ should be used as AE but if it stored at intervals of eight addresses then $i_1 \cdot N_1 + j_1$ should be used as AE.

Therefore, the AE for input/output variables of the applications mapped onto TCPA can be formally expressed below as a scalar product of a row vector $C$ and the column vector $I$, which denotes the iteration or index vector.

$$ (AE) : \text{Addr} = c \cdot I + c_0 \quad \text{where} \quad c \in \mathbb{Z}^{1 \times n} \quad \text{and} \quad I \in \mathbb{Z}^{n \times 1} $$

The AC is given by the condition space denoted by the set of inequalities that indicate when the AE is valid. The generated address is valid only when all the inequalities in the AC are valid, which can be expressed as follows.

$$ (AC) : A \cdot I \geq b \quad \text{where} \quad A \in \mathbb{Z}^{r \times n}, I \in \mathbb{Z}^{n \times 1} \quad \text{and} \quad b \in \mathbb{Z}^{r \times 1} $$

In the TCPA architecture, AGs has some more requirements which are summarized below.

6.2.1 Reconfigurability

Since the TCPA is used as a programmable accelerator for an infinity of different applications, the AGs have to be reconfigurable. At synthesis time, the required number of AGs is instantiated in the TCPA design. This number is equal to the number of I/O variables of an application. Whenever an application has to be ported onto a TCPA, there is a configuration overhead as shown in Figure 6.5. One has to program the PEs in the array with the corresponding programs as well as interconnect, global controller, and address generators in the design with their configuration data. For this configuration, we use a LEON3 processor as depicted in Figure 2.8, and in Figure 6.5. Here, we only emphasize the order in which the different modules are getting configured in a TCPA tile and the exact configuration time depends on the application itself. This configuration time is typically negligible compared to the real execution time of an application.

6.2.2 Synchronization of Streams

In a TCPA, as many simultaneous read/write operations take as at the border of the array. A PE may even consume one or more input data or produce one or more output data per cycle depending on the loop nest. One or more compute resources computing an algorithm can consume two or more streams depending on the algorithm. In such cases, there is a need to synchronize these streams.
Figure 6.5: Exemplary timing diagram showing the configuration sequence of different modules in the TCPA tile. Once all the modules are configured, they work synchronously with each other. The exact configuration time depends on the application and the way it is mapped onto the TCPA array.
6. Address Generators

for correct computational results according to their local latencies which determine when to read/write from/to a stream. For the FIR filter application given in Listing 3.5, both ‘a_in’ and ‘u_in’ are read at the same time so both must be synchronized in terms of their arrival time, a similar concept is already proposed in [SH11]. For output variable ‘y_out’, it should be written to output buffers when it reaches the iterations where \( j = 7 \) in processors \( PE(0, 3), PE(1, 3), PE(2, 3), \) and \( PE(3, 3) \).

However, once the ‘y_out’ statement has finished executing, output buffers can be written. It has a local latency of two cycles as shown in Reduced Dependence Graph (RDG) of Figure 3.8(a). It also means that the generated address and address enable conditions must be also delayed by two cycles. This delay has to be programmable which also depends on the loop program and its local schedule. To implement this delay, a Configurable Shift Register (CSR) as shown in Figure 6.4 [HSL+13*] is used whose delay can be programmed before starting the application. More details on this CSR are given in Section 6.4.4.

6.3 Design Space Exploration

To address the requirements listed in the previous section, we propose three distinct address generator architectures, which are mostly derived from the ones found in the literature and are customized to best suit the TCPA architecture.

6.3.1 LUT based Address Generator

The solution proposed in this section is based on the counter/table based AG as described in Section 6.1.2. If the address sequences are short, they can be directly mapped in a LUT in case of a Field-programmable Gate Array (FPGA) target. The address sequences of an address equation are pre-computed and loaded inside a memory. A counter is used to read the addresses from the memory in a sequential manner and the read addresses are then forwarded to the output. Three configuration registers are used that define the starting address, the final address and the initial delay of the address sequence. This architecture has the advantage to cover a wide range of AEs from a simple affine address equations to a complex and non-affine address equations. If different address equations and address constraints have to be supported, these can also be pre-computed and all loaded inside the memory. By changing the values of the configuration registers, the AG can switch from one context to the other with a reduced delay. These basic AGs can be instantiated multiple times in TCPA where more than one address should be generated depending on the number of input and output variables. The main drawback of this architecture is that it is not scalable. If the address sequence is too long, this solution becomes too
6.3 Design Space Exploration

big [TJC08]. The storage required would also increase with the increasing loop bounds and the level of loop nesting.

6.3.2 Incremental AG

This solution is best suited for applications of streaming nature and very regular AE. Here, we refer to stream as a set of incoming/outgoing data elements which has a periodic delay between two successive data elements. In such cases, the generated address sequence also has a periodic characteristic, which can be defined with the following parameters as described in [SH11]:

- **Initial Address** \((a_0)\): Start address of the stream.
- **Middle Step** \((a_{step})\): Difference between the two successive addresses.
- **End Address** \((a_{end})\): Last address of the stream.

The address equation can be formally expressed as below:

\[
(AE) : \quad Addr = a_0 + x \cdot a_{step}
\]  

The proposed architecture consists of a counter with three configuration registers to map \(a_0\), \(a_{step}\), and \(a_{end}\). To deal with the synchronization problem, we define the following parameters as described in [SH11]:

- **Initial Delay** \((D_{init})\): Delay needed before starting of a stream.
- **Middle Delay** \((D_{middle})\): Delay needed between the generation of two successive addresses.
- **Repetition Delay** \((D_{rep})\): Delay used to delay a stream before repeating an address generation pattern. This kind of delay is used in scenarios where the current stream has been processed and the AG is waiting for new data to arrive so that it can start the same addressing pattern again.

This type of AG is highly scalable since it requires a small area and has a short critical path. The reconfiguration overhead is also very small. The address pattern can be changed just by programming the configuration registers. The reconfiguration delay is reduced to the time needed to program the configuration registers which in this case is very minimal. The synchronization problem can be solved by adequately choosing the delay registers \(D_{init}\), \(D_{middle}\), and \(D_{rep}\). In Figure 6.6, three streams of data \(S_0\), \(S_1\), and \(S_2\) are shown which have to be synchronized. The synchronization problem for these three data stream can be solved as follows. For arrival time synchronization of streams \(S_0\) and \(S_1\), the
6. Address Generators

**Figure 6.6:** Timing diagram showing the synchronization problem.

Initial delay of $S_1$ should be extended by $\Delta_{\text{init}}$. In order to match the periodic delay of streams $S_0$ and $S_2$, $D_{\text{middle}}$ of $S_0$ should be extended by $\Delta_{\text{middle}}$.

In order to integrate incremental AG into a TCPA, multiple address generators are instantiated depending on the number of input and output variables so that for each variable, addresses can be generated independently of the other. The main disadvantage of this solution is that it is restricted to simple AEs in the form described in Equation (6.6). It is best suited for very regular address equations.

### 6.3.3 Scalar Product-based AG

As we mentioned in the Chapter 3, TCPAs implement piece-wise regular algorithms whose AEs are affine functions as described in Equation (6.7).

\[
(AE) : \text{Addr} = c \cdot I + c_0 \quad \text{where} \quad c \in \mathbb{Z}^{1 \times n} \quad \text{and} \quad I \in \mathbb{Z}^{n \times 1} \quad (6.7)
\]

\[
(AC) : A \cdot I \geq b \quad \text{where} \quad A \in \mathbb{Z}^{r \times n}, I \in \mathbb{Z}^{n \times 1} \quad \text{and} \quad b \in \mathbb{Z}^{r \times 1} \quad (6.8)
\]

Thus, implementing a module that generates address sequences according to the AE and valid in an AC space is reduced to implementing a module that best implements a scalar product. This module can be used first to compute the address values directly from the AE as defined in Equation (6.7). Later on to compute the values of the half-space equations determining the AC as defined in Equation (6.8). The values of the vector $c$, matrix $A$, and vector $b$ are passed to the design through configuration registers. The iteration vector $I$ is generated in the GC which is also given as input to the address generators. The computation tree of a standard dot product between two vectors is depicted in Figure 6.7 where the overall performance is limited only by the number of multiplications and additions that could be done in parallel. In order to trade-off performance versus area, we discuss three different implementations of the scalar...
6.3 Design Space Exploration

Figure 6.7: Data flow graph of an AE according to Equation (6.7) for $n = 4$, $c_0 = 0$.

product-based address generators. Area, performance, and resource utilization of the design depends on the dimension of the address equation and address condition which should be taken into account while considering these different implementation choices.

The first solution is a straightforward approach where the computation data flow graph is directly mapped onto multipliers and adders. In this case, the scalar product becomes a huge combinatorial logic. The main advantage of this solution is that an address can be generated in a single clock cycle. However, the critical path of the design can be very long since the number of multipliers and adders required would also increase with the increase in the dimension of AE. Hence, pipelining can also be used.

In the second solution, both pipelining and parallel processing techniques are used to improve the performance of the scalar product as described in [Tan84, QAA09]. For the applications considered in this dissertation, massive throughput is a requirement and this is achieved by using a pipelined architecture that computes the scalar product of two vectors at every clock cycle. Pipelining is achieved by inserting registers at appropriate places in the design and saving results of the partial computation. Parallel processing is also exploited in the design where the scalar product operation is divided into several smaller operations, according to a computation tree, which are executed in parallel. Finally, the output is obtained by accumulating all the partial results generated from these parallel computations. This results in a higher frequency of operation. In case that we have only a limited number of multipliers available, different operations can also be scheduled in a way that only these number of multipliers/adders are used. However, this solution will result in an initial delay which
should be taken into account else it might lead to synchronization issues between the address generator and global controller.

In the third solution, we cover the entire data flow graph by a basic block made of two multipliers and one adder to minimize the computation delay at the expense of an initial delay. This gives us the possibility to compute large scalar products even though an adder and two multipliers are available.

**Combinatorial Solution**

In the proposed solution, each scalar product can be run in parallel, each one implementing a distinct address equation. The scalar product itself is implemented by a combinatorial circuit implementing a computation tree as shown in Figure 6.8. The index vector is given as an input to the design and the address equation parameters are programmed into the configuration registers. This design has the advantage of computing a new address at each clock cycle. The reconfiguration overhead is also very small since changing the address generation pattern is reduced to reconfiguring the registers in the design. It can also support a wide range of address generation patterns as long as their AEs are piece-wise regular.

If the dimension of the vector increases, the number of multipliers needed and the depth of the data flow graph increases. This leads to higher area utilization and long critical path in the design. To overcome these issues, pipelining can be
Figure 6.9: Combinatorial scalar product implementation where each stage of the computation tree is pipelined.

used where registers are inserted between the intermediate stage of the computation tree as shown in Figure 6.9, which significantly reduces the critical path delay.

Systolic Solution

The systolic scalar product solution exploits both pipelining and parallel processing techniques to improve the performance of a scalar product implementation. With a similar approach to the previous one, each scalar product can be run in parallel, each one implementing a distinct address equation. The implementation of the scalar product is very similar to a systolic scalar product proposed in [ABMS00, AB02, BAB05] except with a slight modification where only the loop indices are pumped into the computing elements. The coefficients for the AEs and the ACs are directly stored in the configuration registers and directly connected to the computing elements.

The block diagram of the proposed parallel architecture is shown in Figure 6.10. It consists of identical Multiplier Accumulator (MAC) units whose number depends on the dimension $n$ of vectors being multiplied. Each MAC performs a MAC operation and gives the partial sum to the next MAC unit. Each MAC unit operates independently and computes a part of the whole computation, similar to systolic processing where each computing element performs a small portion of the task and pass it to the next one. This approach greatly
helps in reducing critical path length which is reduced to the critical path of a multiplier accumulator operation. Beside the minimal area utilization and the short critical path, this solution is also scalable. The only drawback is the initial delay which should considered before the generated addresses are valid otherwise it may lead to synchronization issues between the global controller and the different address generators.

**Building-Block-based Solution**

The main idea behind this solution is to cover the entire computation data flow graph with a building block made of two multipliers and an adder. The computations of these blocks are scheduled in a way that minimizes the area needed and the computation delay. By doing this, the design is also adapted to the resource constraints. If we have enough multipliers and adders, the building blocks can work in parallel. If it is not the case, some blocks can be delayed allowing a pipelined execution at the block level. Figure 6.11 illustrates this idea. In Figure 6.11(a), we have enough resources and the basics blocks I and II can run in parallel while the block III is delayed due to the data dependency. In Figure 6.11(b), only two multipliers and one adder are available in the target architecture. Therefore, the block I is executed first, followed by block II, and III. It is to be noted that each time whenever a distinct block is executed, module inputs are changed.
6.3 Design Space Exploration

Figure 6.11: Building-block-based solution: mapping and scheduling of operations onto building block containing two multipliers and an adder.
6. Address Generators

The mapping and the scheduling of operations onto building blocks is done in a off-line phase since all the parameters are known before the hardware synthesis. A multiplexer logic is also used to switch between the different inputs to the modules. This solution uses the available resources with the best efficiency. It is also scalable since the critical path length remains the same. It is also recongurable since the resulting architecture mainly depends on the number of available resources and not on the recongurable registers or the index vector which is given as the input signal. The reconguration of the design is done by just programming the configurable registers in the design. The main disadvantage of this approach is that it may require many stages as well as many clock cycles to compute a single address which results in a lesser throughput.

6.4 Hardware Implementation

In this section, we discuss the implementation details pertaining to few of the above solutions. Different designs were compared in terms of area, maximum clock frequency of operation, and power dissipation. The main design constraint of a recongurable address generator is its flexibility. The flexibility of a design determines its ability to deal with different applications. In our case, the flexibility of an implementation is given by its ability to switch between two configurations. That means, it can be measured by the configuration time and the amount of configuration data required. Furthermore, we will also analyze the scalability of each solution which is defined as the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth [Bon00]. Finally, we present a small flow chart to decide which solution best implements a given application.

6.4.1 LUT-based Address Generator

The block diagram in Figure 6.12 [Rom15] shows the hardware implementation details of a LUT-based address generator. A Block Random Access Memory (BRAM) is used to store the pre-computed address sequences of the buffer. The size of the BRAM is specified by the parameter $B_{SIZE}$. A counter is used to generate the read address for the BRAM in a sequential manner. The counter starts from an initial address specified by a configurable register, $AG_{addr_{-}init}$, and counts up to a maximum address value specified by a configurable register, $AG_{max_{-}addr}$. A Configurable Shift Register (CSR) is used to set the initial delay of the address generator. In situations where the total address range is exceeds the BRAM size, it needs to be loaded multiple times with appropriate addresses. To facilitate this behavior, a configurable register named $AG_{limit}$ exists in the design whose value is always compared with the generated address.
### Table 6.1: List of all configurable registers in the design of LUT-based AG.

<table>
<thead>
<tr>
<th>Register</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG_addr_init</td>
<td>0</td>
<td>Reconfigurable initial address value of the counter where it starts.</td>
</tr>
<tr>
<td>AG_max_addr</td>
<td>1023</td>
<td>Reconfigurable maximum address value of the counter where it ends.</td>
</tr>
<tr>
<td>AG_limit</td>
<td>1023</td>
<td>Reconfigurable limit of the BRAM where the interrupt is generated.</td>
</tr>
<tr>
<td>AG_initial_delay</td>
<td>2</td>
<td>Initial delay of the address generator.</td>
</tr>
</tbody>
</table>

When the generated address reaches the limit value, a stop signal is generated to notify the processor array and an interrupt is generated to initiate data transfer to/from the external memory.

Table 6.1 lists all the configurable registers in the design with a brief description of each register. In order to validate the design as well as quantify the performance, a series of experiments were conducted where the address width is increased to 2, 4, 8, 10, 12, 16, and 18 bits. For the LUT-based AG, increasing the dimension of the index vector, or the address equation, or the address condition does not influence anything in the hardware since the addresses are pre-computed. Therefore, changing or increasing the dimension of the AE or the AC might increase only the address range, which is controlled already in the design using the configurable registers which affects only the size of the BRAM, controlled by $B\_SIZE$. In the experiments, we fixed the BRAM size as $2^{ADDR\_WIDTH}$ and synthesized the design for different address widths.

![Figure 6.12](image-url): Block diagram showing the hardware implementation of the LUT-based AG.
Implementation Results: Area, Performance, and Power

In the following, we report the area in terms of type and consumption of basic components of the Field-Programmable Gate Array (FPGA) such as LUTs, BRAMs, and registers. Throughout these experiments, the XC5VLX330 device (FF1760 package and “-1” speed grade), which is one of the largest available Virtex-5 devices is considered. We used the Xilinx’ ISE tool flow to report the area, performance, and power numbers. The maximum clock frequency and area is reported after performing synthesis using Xilinx’ XST tool whereas the Xpower tool is used in reporting the power numbers. Figure 6.13 shows the implementation results as mentioned above for the LUT-based AG.

Flexibility

The flexibility of the design is evaluated by the amount of configuration data needed and the required configuration time. The configuration data size, denoted by $CD_{LUT}$, is given by Equation (6.9). Similarly, the configuration time in cycles, denoted by $CT_{LUT}$, is given by Equation (6.10) where $CONFIG\_DATA\_WIDTH$ denotes the configuration data width which is equal to the configuration memory data width and $ADDR\_WIDTH$ denote the width of the address to generate.

\[
CD_{LUT} = CONFIG\_DATA\_WIDTH \cdot (4 + 2^ADD\_WIDTH) \tag{6.9}
\]

\[
CT_{LUT} = \frac{CD_{LUT}}{CONFIG\_DATA\_WIDTH} = 4 + 2^ADD\_WIDTH \tag{6.10}
\]

Figures 6.14 and 6.15 show the configuration data, and configuration time requirements for the LUT-based AG considering various address widths and configuration data width of 32 bits.

Scalability

From the results, it can be concluded that this design has an almost constant resource usage for small values of $ADD\_WIDTH$ in which case the size of the BRAM also remains small. However, if the address width is greater than 10 bits, then the size of BRAM will increase rapidly. This solution is not feasible for address width greater than 20 bits.

6.4.2 Incremental AG

The incremental type AG is based on a counter as aforementioned in Section 6.3.2 with an initial delay controlled by a CSR, see Figure 6.16 for details.
6.4 Hardware Implementation

(a) Maximum frequency and average power consumption of the LUT-based AG [Rom15].

(b) Resource utilization of the LUT-based AG [Rom15].

Figure 6.13: Implementation results of the LUT-based AG [Rom15].
This initial delay value is specified by a register $AG\_initial\_delay$. The address sequence starts from an initial address value specified in a configurable register $AG\_addr\_init$. In the subsequent cycles, a step value is added to the current address to get the next address. The step value of the counter is also specified in a configurable register $AG\_step$. A configurable register $AG\_counter\_dir$ is used to set the direction of counter, depending on its value, the counter will increment or decrement the output address at each clock cycle. In order to repeat the address sequences after reaching a cyclic address value, the generated address is always compared with a address value specified in a configurable register $AG\_cycle\_addr$. Once this address is reached, the counter starts again from the initial address value after a repetition delay as shown in Figure 6.16, which is also specified in a configurable shift register $AG\_repetition\_delay$. Moreover, the generated address value is also compared with the reconfigurable bound of the buffer specified in a configurable register $AG\_limit$. When the generated address is outside the buffer bound, AG generates the stop signal which will pause the processing array and fires an interrupt to initiate a data transfer to/from the external memory from/to the buffer. The required configuration registers and their brief description is given in Table 6.2.
6.4 Hardware Implementation

![Graph showing configuration time vs. ADDR_WIDTH](image1)

**Figure 6.15:** Required configuration time for LUT-based AG

![Block diagram of hardware implementation](image2)

**Figure 6.16:** Block diagram of the hardware implementation of an incremental type AG [Rom15].
### Table 6.2: List of all configurable registers in the design of incremental AG.

<table>
<thead>
<tr>
<th>Register</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG_addr_init</td>
<td>0</td>
<td>Reconfigurable initial address value of the counter where it starts.</td>
</tr>
<tr>
<td>AG_step</td>
<td>1</td>
<td>Step value of the counter.</td>
</tr>
<tr>
<td>AG_counter_dir</td>
<td>0</td>
<td>Counter direction, default is up.</td>
</tr>
<tr>
<td>AG_limit</td>
<td>1023</td>
<td>Limit of the buffer, after this address an interrupt is generated.</td>
</tr>
<tr>
<td>AG_initial_delay</td>
<td>2</td>
<td>Initial delay of the AG.</td>
</tr>
<tr>
<td>AG_repetition_delay</td>
<td>2</td>
<td>Repetition delay of the AG.</td>
</tr>
<tr>
<td>AG_cycle_addr</td>
<td>1023</td>
<td>Last address of an address sequence.</td>
</tr>
</tbody>
</table>

### Implementation Results: Area, Performance, and Power

To validate the implementation of this design and to quantify its performance, we synthesized it for different address widths starting from 2, 4, 8, 16, 20, and 32 bits. In general, the configuration registers are computed from the AE and the AC. Therefore, for this type of design, changing or increasing the dimension of the AE and the ACs is equivalent to changing the values of the configuration registers. In Figure 6.17(a), the maximum frequency of operation as well as average power dissipation, and in Figure 6.17(b), resource utilization of the design after synthesis for different address widths are shown.

### Flexibility

Like in the previous section, the flexibility of this solution is evaluated in dependence of configuration data size and the configuration time. The configuration data size $CD_{INC}$ is given by the Equation (6.11) and the configuration time $CT_{INC}$ by the Equation (6.12).

\[
CD_{INC} = 7 \cdot \text{CONFIG\_DATA\_WIDTH} \quad (6.11)
\]

\[
CT_{INC} = 7 \quad (6.12)
\]

### Scalability

From the implementation results, the resource utilization of the incremental-type AG scales linearly. Even the maximum frequency of operation is above 250 MHz even for larger address widths. The required configuration data and configuration time remain constant irrespective of different address widths. It is
6.4 Hardware Implementation

(a) Maximum frequency of operation and average power dissipation of an incremental type address generator for different address widths [Rom15].

(b) Resource utilization of an incremental type address generator for different address widths [Rom15].

Figure 6.17: Hardware implementation results for the incremental type AG [Rom15].
the optimal implementation of an address generator when the data to be read or written have streaming nature such as the nested loop programs mapped onto TCPA.

### 6.4.3 Scalar Product-based AG

In this solution both the Address Equation (AE) and the AC are expressed by Equation (6.7) and Equation (6.8), respectively. The AE represents a linear equation to compute the address, which is basically a scalar product of a row vector \( c \in \mathbb{Z}^{1 \times n} \) with the index vector \( I \in \mathbb{Z}^{n \times 1} \). The AC denotes the test in Equation (6.8) is necessary to check whether the computed address is valid. Here, each of the half-spaces can be implemented using a scalar product similar to the AE. In the following, an address equation and an address condition with each half-space implementing a distinct scalar product is given.

\[
\begin{align*}
AE &: \quad Addr = c \cdot I + c_0 \quad \text{where} \quad c \in \mathbb{Z}^{1 \times n}, \ I \in \mathbb{Z}^{n \times 1}, \ \text{and} \ c_0 \in \mathbb{Z} \\
AC_0 &: \quad A_0 \cdot I \geq b_0 \quad \text{where} \quad A_0 \in \mathbb{Z}^{1 \times n}, \ I \in \mathbb{Z}^{n \times 1}, \ \text{and} \ b_0 \in \mathbb{Z} \\
& \vdots \\
AC_{r-1} &: \quad A_{r-1} \cdot I \geq b_{r-1} \quad \text{where} \quad A_{r-1} \in \mathbb{Z}^{1 \times n}, \ I \in \mathbb{Z}^{n \times 1}, \ \text{and} \ b_{r-1} \in \mathbb{Z} 
\end{align*}
\]

Therefore, scalar product has to be implemented efficiently. Figure 6.18 shows our implementation which is based on this representation of the AE and the AC given above. Our implementation uses one of these basic elements (scalar product unit) to directly compute the addresses from a given AE. The design also instantiates \( R \) scalar product units in parallel to implement the AC. In implementing the AC, once a scalar product unit computes a value, it is compared to the corresponding element of \( b \) vector stored in a configuration register. The computed address is valid only if all the computed values of all the scalar product units of the AC are bigger than their corresponding elements in \( b \) vector. The configuration register \( c \) stores the value of the row vector \( c \). Each configuration register \( A_i \), \( 0 \leq i \leq r - 1 \) stores one row of the matrix \( A \). The index vector or iteration vector is directly given to the design as an input signal.

A CSR is used to implement the initial delay of the address sequence whose delay is controlled by a configuration register \( AG\_initial\_delay \). Furthermore, the generated address is always compared with the configurable buffer bound specified in a register \( AG\_limit \). If the generated address reaches this limit value, the design generates a stop signal and an interrupt is also fired to fill/drain the buffers from/to external memory. Our implementation also provides the possibility to pause the address generation sequence. We implemented the scalar product unit in two different ways:
Figure 6.18: Hardware implementation of a scalar product-based AG.
6. Address Generators

Table 6.3: List of all configurable registers in the design of matrix multiplication based AG.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG_limit</td>
<td>An interrupt is generated after reaching this limit.</td>
</tr>
<tr>
<td>AG_initial_delay</td>
<td>Initial delay of the AG.</td>
</tr>
<tr>
<td>c</td>
<td>Elements of the c matrix in the AE.</td>
</tr>
<tr>
<td>c₀</td>
<td>Offset value of the address, default is zero.</td>
</tr>
<tr>
<td>A</td>
<td>A matrix of the AC.</td>
</tr>
<tr>
<td>b</td>
<td>b vector of the AC.</td>
</tr>
</tbody>
</table>

- A **combinatorial logic implementation** where the scalar product is directly implemented as a huge combinatorial logic with no resource restrictions. It can compute a new address every clock cycle but requires a large amount of resources and the maximum frequency is relatively slow, see Figure 6.8.

- A **pipelined implementation with no resource restriction** which exploits both pipelining and parallel processing techniques to improve the performance of the scalar product unit. This design basically contains registers between each stage of the scalar product unit, see Figure 6.9.

The required configuration registers in the design and their brief description is given in the Table 6.3.

Implementation Results: Area, Performance, and Power

To validate our implementation and to quantify the performance of this design, we have conducted a series of experiments. The address width was set to 10 bits, the number of half-spaces in the address condition was set to four and the dimension of the index vector or iteration vector was increased to 2, 3, 4, 5, 6, 10, and 16. Figure 6.19 shows the area utilization of both the designs after performing the synthesis and Figure 6.20 shows the maximum frequency of operation. Variation of the average power with increasing index vector dimension is shown in Figure 6.21. It is evident from these results that for both designs, the resource utilization as well as average power goes up with increasing index vector dimension. In the pipelined design, the frequency scales very well as the maximum frequency of operation depends only on a single critical pipeline stage. In our case, this critical stage either includes an adder with three inputs or two inputs which depends on the index vector dimension. Therefore, the frequency of operation also oscillates between two values as shown in Figure 6.20.
6.4 Hardware Implementation

![Resource utilization of the scalar product unit with increasing index vector dimension—combinatorial implementation.](image)

![Resource utilization of the scalar product unit with increasing dimension—pipelined implementation.](image)

**Figure 6.19:** Resource utilization of the scalar product unit in both combinatorial and pipelined implementations.
6. Address Generators

![Graph showing frequency vs. dimension for Address Generators.]

**Figure 6.20:** Maximum frequency of operation with increasing index vector dimension for scalar product unit in both the implementations.

**Flexibility**

As aforementioned, the flexibility of our design was evaluated by the amount of required configuration data size and time. In the case of the scalar product-based address generator, the configuration data and time for the both the implementations is the same. The configuration data size, denoted by $CD_{MM}$, is given by Equation (6.14) and the configuration time, denoted by $CT_{MM}$, is given by Equation (6.15). In these equations, $n \in \mathbb{N}$ denotes the dimension of the iteration vector and $r \in \mathbb{N}$ denote the number of half-spaces in the AC. $CONFIG\_DATA\_WIDTH$ is the configuration data width which is equals to the configuration memory data width.

\[
CD_{MM} = CONFIG\_DATA\_WIDTH \cdot (3+n+n\cdot r+r) \quad (6.14)
\]

\[
CT_{MM} = \frac{CD_{MM}}{CONFIG\_DATA\_WIDTH} = 3 + n + n \cdot r + r \quad (6.15)
\]

**Scalability**

To compare the scalability of both the implementations, the number of half-spaces in the address condition can be increased along with the dimension $n$ of
6.4 Hardware Implementation

![Graph showing average power with increasing index vector dimension for scalar product unit in both the implementations.](image)

Figure 6.21: Average power with increasing index vector dimension for scalar product unit in both the implementations.

the iteration vector. The resource utilization of both the designs will increase further but the maximum frequency of operation remains the same as each scalar product is computed in parallel and the critical path remains the same. For smaller dimensions of the index vector, a combinatorial implementation is preferred. However, as the dimension of the iteration vector increases, its frequency of operation drops quickly. Therefore, a pipelined implementation is preferred in such cases. It should be noted that the pipelined implementation will have an initial delay, implemented by a CSR.

6.4.4 Configurable Shift Register

To implement programmable initial delays or to synchronize multiple different input streams, programmable delay elements are needed—implemented by CSRs as explained before. Furthermore, data partitioning onto neighboring processing elements and loop-carried data dependencies between iterations may introduce offsets between computations [HDT06], however, the addressing scheme for data access of each of the tile remains the same. Therefore, it is sufficient to delay the address and enable signal generated from a single AG through the CSR and just propagate it to neighbor buffers in a delayed fashion as shown in Figure 6.4. The
6. Address Generators

![Architecture of Configurable Shift Register (CSR) with maximum depth of 15 [HSL+13*].](image)

Figure 6.22: Architecture of Configurable Shift Register (CSR) with maximum depth of 15 [HSL+13*].

exact delay values depend on the given schedule, i.e., on the start time of the tiles or processors. The architecture of the CSR is shown in Figure 6.22 where the depth of the register is controlled by select signals, which are driven by a programmable register. Instead of enabling each part of the CSR individually, the amount of necessary control logic can be greatly reduced by following a logarithmic scheme as detailed in Figure 6.22, which also reduces the energy consumption. In a real hardware implementation, the maximum depth of the CSR has been fixed to 256. The same CSR is also used in delaying the control signals among the PEs as shown in Figures 5.6 and 6.4.

6.4.5 Address Generator Selection

Given a type of application, selecting a right AG depends on the application characteristics itself. Based on the implementation results, flexibility and scalability for the proposed designs, we propose a flowchart as shown in Figure 6.23 as guidance for selecting a suitable implementation of the AG design.

6.5 Configuration Data

In the back end of our compiler framework, the configuration data for the address generators is generated as shown in Figure 3.4. Once the partitioning, allocation and scheduling have been performed in the front end, all the information required for generating configuration data for different modules such as global controller and address generators is available. For address generators, these configuration data include: the total number of address generators, for each address generator the number and depth of buffers to which it is connected, address equation, address condition, and initial delay. In the following paragraphs, we will describe how this configuration data is extracted automatically from a given scheduled loop program.
6.5 Configuration Data

Figure 6.23: A flow chart to select a suitable AG implementation.

**Number of AGs** The required number of programmable AGs is equal to the number of I/O variables specified in the input loop program. Each I/O variable can be easily identified in the given PAULA program using the keywords *in*/*out*, respectively. As an example, an LSGP partitioned FIR filter program is given in Listing 3.5. It requires to configure exactly three AGs as shown in Figure 6.4.

**Number of Buffers & Depth** In order to know how many buffers are required per I/O variable, one needs to understand the process of *buffer mapping*. It considers the question how to distribute given input/output variables to different buffers available around the TCPA. In case of a programmable accelerator such as a TCPA, the buffer mapping may be conferred from the chosen space-time
mapping. As stated in Section 3.2.4, the space-time mapping determines the allocation of the loop iterations to the PEs through the allocation matrix $Q$, which in turn is decided by the chosen tiling strategy.

**Definition 6.2** (The processor mapping of I/O variables [Dut11]) defines the processor space where an I/O variable $x_i$ will be read/written from a connected RB, respectively. It is given by the following mapping $f : I \rightarrow p$.

$$P_i = \{ p \in P | p = Q \cdot I \land I \in I \land I \in C_i \}$$ \hspace{1cm} (6.16)

where $P$ denotes the processor space and $C_i$ denotes the condition space of those statements where $x_i$ is used (inputs), respectively defined (outputs).

Now, buffer mapping assigns a unique buffer module to each processor element above performing I/O operations.

**Example 6.2** Consider the partitioned FIR filter program given in Listing 3.5 with $Q = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ according to Equation (3.8) and whose space-time mapping is shown in Figure 3.7 with global iteration space as define below:

$$I = \{ (i_1, j_1, i_2, j_2)^T \in \mathbb{Z}^4 | (0 \leq i_1 < T_1) \land (0 \leq j_1 < N_1) \land (0 \leq i_2 < T_2) \land (0 \leq j_2 < N_2) \}$$

Let the condition space for each input and output variable are defined as below:

$$C_{a_{-in}} = \{ (i_1, i_2)^T \in \mathbb{Z}^4 | (i_1 == 0) \land (i_2 == 0) \}$$

$$C_{u_{-in}} = \{ (j_1, j_2)^T \in \mathbb{Z}^4 | (j_1 == 0) \land (j_2 == 0) \}$$

$$C_{y_{-out}} = \{ (j_1, j_2)^T \in \mathbb{Z}^4 | (j_1 == N_1 - 1) \land (j_2 == N_2 - 1) \}$$

Now, using Equation (6.16), we obtain processor mapping for each I/O variable as given below:

$$P_{a_{-in}} = \{ (p_1, p_2) \in P | (p_1 = 0) \land (0 \leq p_2 < N_2) \}$$

$$P_{u_{-in}} = \{ (p_1, p_2) \in P | (0 \leq p_1 < T_2) \land (p_2 = 0) \}$$

$$P_{y_{-out}} = \{ (p_1, p_2) \in P | (0 \leq p_1 < T_2) \land (p_2 = N_2 - 1) \}$$

The buffer space of variable, ‘a\_in’ is $P_{a_{-in}}$. Therefore, for $N_2 = 4$, four buffers need to be connected to PE(0,0), PE(0,1), PE(0,2), and PE(0,3), respectively as shown in Figure 6.4. Similarly, there are four buffers for variable ‘u\_in’ as well as ‘y\_out’ as shown in Figure 6.4.
The buffer depth or size is given by the number of iterations corresponding to an I/O variable assigned to a processor element connected to the same buffer. Therefore, the intersection of the condition space where the I/O variable is defined and the iteration space assigned to the processor connected to the buffer determines the depth. More formally, $|C_i \cap I_p|$ denotes the size of the buffer, where $C_i$ denotes the condition space of the I/O variable and $I_p$ denotes the iteration space of the processor connected to the buffer.

**Example 6.3** For the example in Figure 6.4, each buffer assigned to variable $a_{\text{in}}$ connected to each PE $(0, k) \forall 0 \leq k \leq 3$ are denoted by $|B_{0k}^{a_{\text{in}}}|$ and given by Equation (6.17). Similarly, each buffer size assigned to variable $u_{\text{in}}$ and $y_{\text{out}}$ are given by Equation (6.18) and Equation (6.19), respectively.

$$|B_{0k}^{a_{\text{in}}}| = |(i_1 = 0) \cap (0 \leq i_1 \leq 7 \land 0 \leq j_1 \leq 7)| = 8 \quad (6.17)$$

$$|B_{k0}^{u_{\text{in}}}| = |(j_1 = 0) \cap (0 \leq i_1 \leq 7 \land 0 \leq j_1 \leq 7)| = 8 \quad (6.18)$$

$$|B_{k3}^{\text{out}}| = |(j_1 = N_1 - 1) \cap (0 \leq i_1 \leq 7 \land 0 \leq j_1 \leq 7)| = 8 \quad (6.19)$$

**AE & AC** The required information related to generating an AE as well as AC for an AG for an I/O variable is already available in the given loop specification. The address is given by the affine access or index function of the I/O variable\(^{14}\). If and only if the current iteration vector lies in the AC, the address valid signal is generated by which corresponding buffers are enabled. It is worth mentioning that a GC generates the new iteration vector only every iteration interval. Therefore, a new address is also calculated only once per iteration interval.

Finally, one I/O variable may be mapped onto multiple buffers. The schedule order of I/O variable access is same for all buffers. Hence, only a single AG for each I/O variable is needed and for each buffer corresponding to the same I/O variable. The address and enable signals are propagated with appropriate delays to the buffers. These delays correspond to the offset of the start time of the neighboring buffers, which belong to the same variable. The length of the delay register is defined by the equation $\delta_d = \min \{ \lambda \cdot (I_1 - I_2) \}$, where $I_1$ and $I_2$ denote the iteration vectors accessing the neighboring buffers of the same I/O variable. In this way, only one AG per I/O variable is indeed needed. In the following, we use the partitioned FIR filter given in Listing 3.5 to illustrate these concepts.

\(^{14}\)It is to be noted that when an I/O variable is defined in PAULA, its dimension is also specified and as the buffer address space is one-dimensional, it has to be linearized with the appropriate loop bounds as mentioned in Section 6.2. In order to calculate the required address, iteration variables are provided as input to the AG hardware as mentioned before by the GC as shown in Figure 6.4. Furthermore, the AC is given by the condition space associated with the I/O variable.
Example 6.4 For the partitioned FIR filter, the AE and AC for different I/O variables are already discussed in Section 6.2 and given by Equation (6.5). For instance, input variable ‘a_in’ is mapped to four PEs and hence, buffers as shown in Figure 6.4. However, a single AG is used for generating the address sequences and enable conditions which are forwarded to neighbor buffers with appropriate delays as shown. Here, the delay between the two buffers connected to PE(0,0) and PE(0,1) for variable ‘a_in’ is eight cycles, which is the offset in their start times. Similarly, for variables ‘u_in’ and ‘y_out’, address as well as enable signals are also propagated using delay registers implemented by a CSR. These connections are already depicted in Figure 6.4. It should be finally noted that for output variable ‘y_out’, the address and enable signals have to be delayed not only by eight cycles due to the offset with a neighboring processor but also additional two cycles due to the local latency of the ‘y_out’ in the RDG, see Figure 3.8(a). This delay is nothing but the initial delay for variable ‘y_out’. However, for both input variables ‘a_in’ and ‘u_in’ local latency is zero; thus, there is no additional delay involved apart from the start time offset between the processors.

In this way, the configuration data for each address generators is generated. As mentioned before and shown in Figure 6.5—the actual computation of the algorithm can only be started—once all the modules in the TCPA are configured. Once the actual computation starts, GC generates the control signals to orchestrate the programs running on the PEs and iteration vectors for the AGs. At each cycle, instructions are executed in each PE according to the generated program and they will also read/write the right data at right times from/to the buffers.

6.6 Related Work

Multimedia applications such as video and image processing are often characterized by a large number of data accesses having index manipulations. Furthermore, the high throughput required for such applications imposes tight timing constraints on address generation. Therefore, use of AGs in parallel with the main computing units is highly recommended [SH11]. In [KOD+91], the authors proposed specialized programmable hardware address generator units (pAGUs) with several sophisticated addressing modes (2-D raster scan, block scan, eight different neighborhood search, etc.) which are suited for image processing. In [VS05], the authors proposed an address generator for stream-based computations that can switch between various addressing algorithms on-the-fly, without stopping. No empty time slots are introduced, neither within one addressing scheme nor when switching to another scheme. However, their implementation is restricted to regular mapping in three dimensions. A shift-register-
based address generator architecture which can generate many regular address sequences, especially those found in block-based image processing algorithms is proposed in [HCC02]. The combined address equation optimization (ADOPT) methodology for address generator synthesis described in [VS05, MCJDM96] is a general framework for optimizing address generators. ADOPT considers both counter-based and arithmetic-based address generator styles and also considers hardware sharing [MCJDM98]. Counter-based synthesis of address generators was proposed in [GDF89] for many signal processing applications where read/write addresses are strongly patterned. Here, the goal is to explore different architectural alternatives available when mapping array references in order to select the most promising ones in area cost. In [MCDM94], the authors propose an address equation multiplexing for real-time signal processing applications where address equations are optimized to exploit possible area savings by intelligent sharing. A survey on methods for memory addressing organization is presented in [HHM02] where address generators from commercial DSP processors are discussed along with the few used in reconfigurable architectures from academia. In [MD04], the authors proposed a distributed address generation and loop acceleration architecture for VLIW processors with multiple SRAMs. Using this approach, two-dimensional affine address equations can be calculated in one cycle whereas the main computing Arithmetic Logic Units (ALUs) are used in case of complex address equations. IMAGINE [KDR02] and Morphosys [LSL99] use a simple address generation scheme where the addresses are pre-computed and stored in a look-up table which is argued to be not scalable in terms of storage. A reconfigurable design composed of multiple PEs and pipeline registers that can be dynamically reconfigured to support various address generation patterns for VLIW and normal processors is proposed in [TRJ09]. This AG makes the address generation scheme very general but it is argued to be costly. Coarse-Grained Reconfigurable Arrays (CGRA) like MONTIUM [HSM03] have AGs that generate at runtime a burst of addresses if the loop indices are affine functions of the outer loop indices. In ADRES [MVV03], a centralized processor is used to compute the address constraints which might be bottleneck if the number of inner loops running in parallel is increased. AGs proposed in this dissertation are tailored to the needs of TCPA and programmable. A single AG is only required per input or output variable to address many buffers corresponding to that variable. Here, the address and enable signals are simply delayed by configurable shift registers to each buffer.

6.7 Summary

In this chapter, we enunciated the importance of address generators which work in parallel with the main computational units or processors to ensure efficient
6. Address Generators

feed/storage of data to/from the main memory by generating addresses. In addition, the general notion of an address equation, which generates the address sequence and address condition, which controls when to read/write from/to a memory are introduced. Later on, regularity and flexibility of the address equation has been discussed which influences the address generator hardware architecture. Furthermore, different address generator architectures found in the literature such as table-based, incremental type, custom and programmable address generator are discussed briefly with few hardware details. A motivating example is presented to give an overview of address generation in the case of a TCPA. How different modules are configured in the TCPA was discussed briefly along with the reconﬁgurability needed by the address generator for different applications. We discussed how the address as well as the enable conditions generated by a single AG are propagated to different buffers by programmable delays. In this context, we also introduced a conﬁgurable shift register by which programmable delays are realized. Different types of address generators such as LUT-based, incremental type and matrix-multiplication based are implemented in hardware and compared in terms of area, maximum frequency of operation, and power consumption. Based on the implementation results, we also proposed a ﬂow chart by which an appropriate AG architecture is selected for a given application. Finally, we discussed how the conﬁguration data for an address generator is generated automatically within the compiler framework.
Conclusions and Future Work

Heterogeneous architectures consisting of standard processors such as Reduced Instruction Set Computers (RISCs), domain-specific processors such as Graphics Processing Units (GPUs), application-specific custom accelerators, and programmable accelerators are inevitable to meet the ever increasing performance demands while providing highest energy efficiency. Here, programmable accelerators are of interest as they can be tailored to an application. However, such programmable architectures lack in programming models and mapping methodologies to compile applications onto them automatically. Such mapping methodologies or compiler frameworks are essential for increasing the productivity of designers as well as for shortening the development cycles. In this context, this dissertation presents novel contributions for automatically mapping nested loop programs onto a claim of massive parallel processor array called Tightly Coupled Processor Arrays (TCPAs), which can be integrated into a more traditional Multi-Processor System-on-Chip (MPSoC) design as programmable accelerators. In this chapter, we summarize the key contributions of this dissertation and an overview of possible future work is given.

7.1 Summary

In this dissertation, we considered TCPAs—a class of massively parallel coarse-grained reconfigurable arrays—suitable for accelerating compute-intensive loop specifications and unique for exploiting both instruction-level parallelism and loop-level parallelism and avoiding any sequential memory access bottlenecks of other massively parallel shared memory architectures by distributed memory computations. We extended an existing high-level synthesis compiler framework called PAR0 to map automatically nested loop programs onto TCPAs. The contributions of this dissertation lie in (a) generating compact code as well as (b) interconnect configuration for all the processors in the array in a way independent of the problem size. In addition, we designed additional required hardware components such as a programmable Global Controller (GC) and Ad-
Address Generators (AGs) for generating address sequences for border memory buffers with the proper compiler support.

We proposed an automatic compact code generation approach which preserves a pre-computed schedule of loop nest iterations and introduces zero timing overhead for the entire loop execution and also independent of the problem size. In our approach, even if the array contains 1000s of processors, we have to generate different code only for few processor types called processor classes. The same program is loaded and executed in each processor belonging to the same class. Later on, in order to avoid code explosion for a single processor class, mutually exclusive iteration space blocks, called program blocks, were identified in a processor class iteration space and represented by a unique control flow graph. Subsequently, code is generated by traversing this program block control flow graph, where repetitive calls to equal program blocks are identified as kind of sub programs. This avoids to replicate each program block code the number of times it will be executed and allows to generate the most compact code for a given loop program. The code compaction approach itself introduces no loop overhead while preserving a given schedule of iterations. Thereby, the required code size is minimized in order to fit well to instruction-memory-limited processors. As a byproduct, also the area as well as power consumption may be reduced. Finally, our approach for multiprocessor code generation has shown significantly better results compared with code produced by (a) the Trimaran compilation infrastructure, even for single PE architectures, and (b) the multi-cluster VEX framework. The generated code sizes are shown to be independent of the loop problem size and also independent of the array size. We also proposed a scalable interconnect generation approach, where the entire processor array is divided into so-called interconnect classes—all processors in a class will have the same type of connections. Later on, the interconnect structure for a single interconnect class is generated—consisting of dedicated links with delay shift registers for each inter processor dependency. The lengths of these shift registers which also depend on a given loop schedule is also determined automatically within the compiler framework.

In order to execute the generated code correctly while avoiding complex branch target computation in each individual Processing Element (PE), a GC is suggested to generate sequence of control signals which orchestrates the branching instruction of a PE. In this realm, we presented the fundamental concepts and proposed the design of a programmable GC which is configured automatically. We presented the functionality of the GC with appropriate algorithms and converted these algorithms systematically into hardware implementation. We also added the appropriate compiler support for generating the configuration data automatically from a given scheduled loop program. The control signals emanating from a GC are propagated to all PEs through the PEs in a linearly staggered fashion with appropriate delays, which are also determined automatic-
cally from a given schedule. Our concept of a global controller in the context of programmable processor arrays is novel and appealing. It is promising since it scales very well with the iteration space and the generated code is independent of the problem size and it avoids to do the same complex branch target computation in each single PE rather than once by the GC. Changing the problem size involves only modifying the loop bounds of the program, which results in either an increase or decrease in the size of program blocks but the number of program blocks remains the same.

Reconfigurable buffers surrounding the processor array are used to either feed or drain all processors at the border of a TCPA cycle by cycle with the correct data. This requires to generate the addresses for these buffers depending on the applied tiling strategy and the computed schedule also in parallel to the distributed execution of the loop nest. Therefore, we proposed the design of programmable AGs, which generate the addresses as well as enable signals for these buffers so that the data is read/written to/from them also in accordance with a given schedule. In addition, we discussed how the configuration data may be generated for these AGs within the compiler framework from a scheduled and partitioned loop program. We evaluated different architecture alternatives for AG design and compared them in terms of area, performance, and power consumption.

To summarize, the major contribution of the dissertation is to ease the compiler gap for the mapping of nested loop programs onto TCPAs by automatically generating compact codes as well as interconnect structure for processors and configuration data for other hardware components such as GC and AGs. For the first time, we proposed a novel code generation and compaction approach as well as the design of programmable GC and AGs.

7.2 Future Work Directions

The contributions in the area of compact code generation cut the first turf and open the way for lot of other interesting future extensions. The code size can be further reduced by noting the fact that the program blocks are due to different condition spaces of statements and moving this selection of different statements to the hardware in the form of multiplexers. However, this would require quite a few changes in the PE architecture and selection logic for these multiplexers which is to be orchestrated according to a schedule. One needs to evaluate the hardware overhead of the architectural changes compared to the area savings of the instruction memory.

In the current code generation approach, instructions from multiple iterations corresponding to distinct program blocks may overlap due to software pipelining. It might lead to multiple distinct overlapped codes for program blocks. This
might also result in increased code size and difficult to find out a repeating kernel of the code. In order to alleviate this problem, the PE architecture could be modified to have a separate instruction memory and program counter for each functional unit. We believe that it may reduce the overall instruction memory size but requires significant changes not only in the hardware but also in the compiler framework.

In the context of invasive computing [TTH14, TTH13], where the actual number of available processors is only known at runtime, we finally would want to exploit a recent breakthrough in symbolic loop parallelization and mapping techniques. Here, latency-optimal schedules are derived in a parametric way statistically including symbolic partitioning, symbolic scheduling, and the generation of symbolic assembly code without the need for the time consuming run-time compilation. In [BHTP11*], we demonstrated manually in a first case study that a single symbolic configuration stream can be generated, which can be used to configure all the PEs for a different number of allocated resources. Our approach shows promising results and is independent of the problem size. We believe that this is an important step towards symbolic configuration code where at runtime, symbols in the programs are replaced by actual values to modify bit streams on-the-fly in order to configure the PEs. The same ideas of the code generation proposed in this dissertation can be extended further towards the fully automated symbolic assembly code generation.

Finally, different architectural enhancements in the design of a GC and AGs are possible which might increase the performance yet at the expense of the additional area. Furthermore, since the data communication is the bottleneck in the TCPA tile, investigation of alternate approaches such as Direct Memory Access (DMA) is indispensable.
Appendices
LEFT_EDGE algorithm was proposed by Hashimoto and Stevens for channel routing [HS71], where intervals correspond to wire segments to be arranged in a channel. Very often, problems are directly specified by the set of intervals $I = \{[l_j, r_j]; j = 1, 2, 3, \ldots\}$ that induce the interval graph [Mic94]. The name stems from the selection of the intervals based on their left-edge coordinate, i.e., the minimum value of the interval. The algorithm first sorts the intervals by their left edge. It then considers one color at a time and assigns as many intervals as possible to that color by scanning a list of intervals in ascending order of the left edge before incrementing the color counter [Mic94].

**Listing A.1: Left Edge Algorithm, source [Mic94].**

```
1 LEFT_EDGE(I) {
2     Sort elements of $I$ in a list $L$ in ascending order of $l_i$;
3     $c = 0$;
4     while (some interval has not been colored) do{
5         $S = \emptyset$;
6         $r = 0$;  /* initialize coordinate of rightmost edge in $S*$/
7         while (exists an element in $L$ whose left edge coordinate is
8             larger than $r$) do {
9             $s =$ First element in the list $L$ with $l_s > r$ ;
10             $S = S \cup s$
11             $r = r_s$;
12             Delete $s$ from $L$;
13         }
14         $c = c + 1$;
15         Label elements of $S$ with color $c$;
16     }
}
```
As aforementioned, our compiling approach is based on the polyhedron model—the iteration space essentially represents a polytope and iterations correspond to integral points in the polytope. A path stride matrix determines the execution order of these integral points in the polytope without violating the data dependencies. For instance, Figure B.1 shows an iteration space assigned to a tile where iterations are scanned in the $j$ direction first and then in the $i$ direction. In other words, we can also say that the entire iteration space is sliced with respect to the $i$ direction. During this scanning, iterations in the different program blocks (same as condition spaces in our case) are executed in turn they also get sliced with respect to $i$. In the polyhedron model, this slicing of program blocks can be achieved by converting the polytope denoted by it into a parameterized polytope considering $i$ as a parameter.

Here, we are interested in knowing the number of integral points in each slice so that the repeating instructions are identified to insert the looping instructions. If a program block is rectangular, each slice of its rectangular space will have the same number of iterations or integral points. In case of non-rectangular program blocks, each slice of it may have different number of iterations. In this context, Ehrhart polynomials are used to count the number of iterations within a slice. In the next paragraphs, given a program block denoted by its iteration space, we will explain how the slicing is performed and integral points in the slice are determined.

**Figure B.1:** An exemplary tile and its iteration space.
Consider the iteration space of \( PB_1 \) given by Equation (B.1).

\[
\mathcal{I}_{PB_1} = \left\{ I = \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \mid \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ 1 & -2 \end{pmatrix} I \geq \begin{pmatrix} 1 \\ -9 \\ 0 \\ -8 \\ 0 \end{pmatrix} \right\}
\] (B.1)

The vertices of this polytope are: \( v_1 = (0.5, 0) \), \( v_2 = (9, 0) \), \( v_3 = (9, 4.5) \). Sorting of these vertices in \( i \) direction leads to a single piecewise linear section in the polytope given by Equation (B.2).

\[
0.5 \leq i \leq 9
\] (B.2)

However, there could be more than one piecewise linear section in the polytope. These piecewise linear sections are also called domains. Given this information, we are interested in the number of iterations (integral points) within each slice \( \{i = n \in \mathbb{Z}\} \cap \mathcal{I} \). That means:

\[
\mathcal{I}_{PB_1} = \left\{ I = \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \mid I + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \end{pmatrix} n \geq \begin{pmatrix} 1 \\ -9 \\ 0 \\ -8 \\ 0 \end{pmatrix} \right\}
\] (B.3)

\( \mathcal{I}_{PB_1} \) represents a parameterized polytope for which Ehrhart Polynomials give the number of integral points in each domain. It is to be noted that, for each domain in the polytope, there exists a distinct Ehrhart Polynomial. For the program block \( PB_1 \), there is only one domain, thus only one Ehrhart Polynomial is anticipated which can be calculated using polylib [Loe07]. In order to understand the Ehrhart Polynomials, we define below, one-dimensional periodic number as well as pseudo-polynomial.

**Definition B.1** (periodic number [Eug62, CLW97]) A one-dimensional periodic number \( u(n) = [u_0, u_1, \ldots, u_{p-1}] \) is equal to the item whose index is equal to \( (n \mod p) \). \( p \) is called the period of \( u(n) \).

\[
u(n) = [u_0, u_1, \ldots, u_{p-1}]_n = \begin{cases} u_0 & \text{if } (n \mod p) = 0 \\ u_1 & \text{if } (n \mod p) = 1 \\ \vdots & \vdots \\ u_{p-1} & \text{if } (n \mod p) = p - 1 \end{cases}
\] (B.4)
Definition B.2 (pseudo-polynomial [Eug62, CLW97]) A polynomial

\[ f(n) = c_k n^k + c_{k-1} n^{k-1} + \cdots + c_0 \]  

(B.5)
is called a (one-dimensional) pseudo-polynomial of degree \( k \) if its coefficients \( c_i \) are one-dimensional periodic numbers, and \( f(n) \) is integral for all \( n \). A pseudo-polynomial has three characteristics: its dimension, degree, and its pseudo-period.

For the parameterized polytope denoted by \( \mathcal{F}_{PB_1} \), given by Equation (B.3), calculating a Ehrhart Polynomial results in the following pseudo-polynomial given by Equation (B.6), whose period is 2. There exists a single domain for which parameterized vertices \( v_1 = (n, 0), v_2 = (n, n/2) \) denote the begin and end of a slice. It is to be noted that parameter \( n = i \).

\[
\text{Ehrhart Polynomial} = \frac{1}{2} n + [1, \frac{1}{2}]_n = 0.5n + [1, 0.5]_n \]  

(B.6)

It can be verified that by looking at Figure B.1 and \( PB1 \), in the \( n = 4 \) slice, there are three iterations or integral points given by Equation (B.6). Here, \( 4 \mod 2 = 0 \), so first number, i.e., 1 is chosen in the periodic polynomial \([1, 0.5]_n\) in Equation (B.6). Evaluating the parameterized vertices for \( n = 4 \) results in, \( v_1 = (4, 0) \) and \( v_2 = (4, 2) \) which shows the begin and the end of slice. Similarly, in \( n = 5 \) slice also three iterations exist. Vertices \( v_1 = (5, 0) \) and \( v_2 = (5, 2.5) \) denote the begin and end of the slice, respectively. Similarly, calculating the Ehrhart Polynomials for the program blocks denoted by \( PB2, PB0, \) and \( PB3 \) in Figure B.1 results in Table B.1—showing a single domain including vertices and its Ehrhart Polynomial for each program block. It is worth noticing that for rectangular program blocks \( PB0, \) and \( PB3 \) Ehrhart Polynomial is just a constant.

<table>
<thead>
<tr>
<th>Program Block</th>
<th>Domain</th>
<th>Vertices</th>
<th>Ehrhart Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>PB1</td>
<td>( n = 0 \land -n + 9 \geq 0 )</td>
<td>([n, 0], [n, 0.5n])</td>
<td>(0.5n + [1, 0.5]_n)</td>
</tr>
<tr>
<td>PB2</td>
<td>( n = 0 \land -n + 9 \geq 0 )</td>
<td>([n, 0.5n + 0.5], [n, 8])</td>
<td>(-0.5n + [8, 8.5]_n)</td>
</tr>
<tr>
<td>PB0</td>
<td>( n = 0 )</td>
<td>([0, 0], [0, 8])</td>
<td>9</td>
</tr>
<tr>
<td>PB3</td>
<td>( n = 10 )</td>
<td>([10, 0], [10, 8])</td>
<td>9</td>
</tr>
</tbody>
</table>

Table B.1: Program blocks and their Ehrhart Polynomials

In this way, Ehrhart Polynomials are used to calculate the number of consecutively executed iterations within a slice of the program block.
Control Flow Analysis: Glossary

Directed Graph: A directed graph, $G$, can be denoted by $G = (V, E)$ where $V$ is the set of vertices (nodes/blocks) and $E$ is the set of directed edges $\{(v_i, v_j), (v_k, v_l), \ldots\}$. Each directed edge is represented by an ordered pair $(v_i, v_j)$ of nodes which indicates that a directed edge goes from node $v_i$ to node $v_j$.

Successor Function: There exists a successor function $SF_G$, which maps $G$ into $SF_G$ such that $SF_G(v_i) = \{v_j | (v_i, v_j) \in E\}$. We call this set as a immediate successors of a node. It may be empty.

Predecessor Function: The inverse of the successor function $SF_G^{-1}$ gives the immediate predecessors of a node. It too may be empty. $SF_G^{-1}(v_j) = \{v_i | (v_i, v_j) \in E\}$.

Basic Block: A basic block (node) is a sequence of program instructions having one entry point and one exit point. It may of course have many predecessors and successors and may even be its own successor (self loop). A start block is the one which do not have any predecessors whereas end block is the one which do not have any successors.

Control Flow Graph: A control flow graph is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths.

Subgraph: A subgraph of a directed graph, $G = (V, E)$, is a directed graph $G' = (V', E')$ in which $V' \subseteq V$ and $E' \subseteq E$. Further more successors function $SF_{G'}$ defined for $V'$ must stay within $G'$; that is for $v'_i \in V'$, $SF_{G'}(v'_i) = \{v'_j | (v'_i, v'_j) \in E'\}$.

Path: A path in a directed graph is a directed subgraph, $P$, of ordered nodes and edges obtained by successive application of the successor function. It is expressed as a sequence of nodes $(v_1, v_2, v_3, v_4, \ldots v_n)$ where $v_{i+1} \in SF^n(v_i)$. The edges are implied: $(v_i, v_{i+1}) \in E$. 
**Successor & Predecessor:** A node $q$ is said to be a successor of a node $p$, if there exists a path $P = (v_1, v_2, \ldots, v_n)$ for which $v_1 = p$ and $v_n = q$. In this case, $p$ is said to be predecessor of $q$.

**Closed Path:** A closed path or circuit is a path in which $v_n = v_1$.

**Path Length:** The length of a path is the number of edges in the sequence. Formally, a distance function is defined such that for any path $P = (v_1, v_2, \ldots, v_n)$, $\delta(P) = n - 1$. Shortest path between two points $p$ and $q$ is defined as $\delta_{min}(p, q) = MIN(\delta(P1), \delta(P2), \ldots)$ for all $P_i = (p, \ldots, q)$.

**Dominator and Dominator Tree:** A node $v_i$ is said to dominate a node $v_k$, if $v_i$ is on every path from root (start) $v_0$ to $v_k$. Let $P = \{P \mid P = (v_0, \ldots, v_k)\}$. Then, the set of back dominators $BD(v_k)$ of $v_k$ consists of all of the nodes other than $v_k$ itself, which are on all the paths from $v_0$ to $v_k$. In other words $BD(v_k) = \{v_i \mid v_i \neq v_k \text{ and } v_i \in P\}$. The immediate back dominator of node $v_k$ is the back dominator which is “closest” to $v_k$. A dominator tree shows all the immediate dominator relationships.

**Back edge:** Edges whose heads dominates their tails are called back edges. For instance, $a \rightarrow b : b = \text{head}, a = \text{tail}$.

**Natural Loop:** Given a back edge $a \rightarrow b$, the natural loop of the edge is “b” plus the set of nodes that can reach “a” without going through the “b” where “b” is the header of the loop. Header is nothing bit a single entry point in the loop that dominates all nodes in the loop.

**Reducibility:** A control flow graph $G$ is reducible if and only if its edges can be partitioned into two disjoint groups, forward edges and back edges. Forward edges form a directed acyclic graph in which every node can be reached from the initial node of $G$. 


FIR Filter Codes

Listing D.1: Final optimized code for $PE(0, 0)$.

```
Instruction0 {  
  ADD: NOP  
  MUL: NOP  
  DPU0: MOV RD1 ID1  
  DPU1: MOV FD0 ID0  
  DPU2: MOV FD1 ID1  
  DPU3: NOP  
  DPU4: MOV RD0 ID0  
  CPU0: MOV OC0 IC0  
  BRU: NEXT
}

Instruction1 {  
  ADD: NOP  
  MUL: MUL RD2 RD0 RD1  
  DPU0: CONST FD1 0  
  DPU1: MOV FD0 ID0  
  DPU2: NOP  
  DPU3: CONST RD1 0  
  DPU4: MOV RD0 ID0  
  CPU0: MOV OC0 IC0  
  BRU: NEXT
}

Instruction2 {  
  ADD: ADDI FD2 RD2 0  
  MUL: MUL RD2 RD0 RD1  
  DPU0: CONST FD1 0  
  DPU1: MOV FD0 ID0  
  DPU2: NOP  
  DPU3: CONST RD1 0  
  DPU4: MOV RD0 ID0  
  CPU0: MOV OC0 IC0  
  BRU: NEXT
}

Instruction3 {  
  ADD: ADD FD2 FD2 RD2  
  MUL: MUL RD2 RD0 RD1  
  DPU0: CONST FD1 0  
  DPU1: MOV FD0 ID0  
  DPU2: NOP  
  DPU3: MOV FD1 ID1  
  DPU4: MOV RD0 ID0  
  CPU0: MOV OC0 IC0  
  BRU: IF IC0 JMP 4, 3
}

Instruction4 {  
  ADD: ADD FD2 FD2 RD2  
  MUL: MUL RD2 RD0 RD1  
  DPU0: CONST FD1 0  
  DPU1: MOV FD0 ID0  
  DPU2: CONST OD2 0  
  DPU3: CONST RD1 0  
  DPU4: MOV RD0 ID0  
  CPU0: MOV OC0 IC0  
  BRU: NEXT
}

Instruction5 {  
  ADD: ADD FD2 FD2 RD2  
  MUL: MUL RD2 RD0 RD1  
  DPU0: MOV RD1 ID1  
  DPU1: MOV FD0 ID0  
  DPU2: CONST OD2 0  
  DPU3: MOV RD0 FD0  
  DPU4: MOV RD0 ID0  
  CPU0: MOV OC0 IC0  
  BRU: NEXT
}

Instruction6 {  
  ADD: ADD OD3 FD2 RD2  
  MUL: MUL RD2 RD0 RD1  
  DPU0: MOV RD1 FD1  
  DPU1: MOV FD0 FD0  
  DPU2: NOP  
  DPU3: MOV RD0 FD0  
  DPU4: NOP  
  CPU0: MOV OC0 IC0
}

Instruction7 {  
  ADD: ADDI FD2 RD2 0  
  MUL: MUL RD2 RD0 RD1  
  DPU0: MOV RD1 FD1  
  DPU1: MOV FD0 FD0  
  DPU2: NOP  
  DPU3: MOV RD0 FD0  
  DPU4: NOP  
  CPU0: MOV OC0 IC0
}
```
BRU: NEXT

Instruction8 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV FD0 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: IF IC0 JMP 9, 8
}

Instruction9 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV FD0 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: IF IC0 JMP 10, 5
}

Instruction10 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 ID1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 ID1
CPU0: MOV OC0 IC0
BRU: NEXT
}

Instruction11 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 ID1
CPU0: MOV OC0 IC0
BRU: NEXT
}

Instruction12 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 ID1
CPU0: MOV OC0 IC0
BRU: NEXT
}

Instruction13 {
ADD: ADD FD2 FD2 0
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: NEXT
}

Instruction14 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: NEXT
}

Instruction15 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: NOP
DPU2: NOP
DPU3: NOP
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: NEXT
}

Instruction16 {
ADD: ADD FD2 FD2 RD2
MUL: NOP
DPU0: NOP
DPU1: NOP
DPU2: NOP
DPU3: NOP
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: NEXT
}

231
**Listing D.2: Final optimized code for PE(0, 1), and PE(0, 2).**

<table>
<thead>
<tr>
<th>Instruction0</th>
<th>Instruction1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: NOP</td>
<td>ADD: NOP</td>
</tr>
<tr>
<td>MUL: NOP</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: CONST RD1 0</td>
<td>DPU0: CONST RD1 0</td>
</tr>
<tr>
<td>DPU1: MOV FD0 ID0</td>
<td>DPU1: MOV FD0 ID0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: CONST FD1 0</td>
<td>DPU3: CONST FD1 0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID0</td>
<td>DPU4: MOV RD0 ID0</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 ID3 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: CONST RD1 0</td>
</tr>
<tr>
<td>DPU1: MOV FD0 ID0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: CONST FD1 0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID0</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: CONST RD1 0</td>
</tr>
<tr>
<td>DPU1: MOV FD0 ID0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: CONST FD1 0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID0</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: IF IC0 JMP 4, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: CONST RD1 0</td>
</tr>
<tr>
<td>DPU1: MOV FD0 ID0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: CONST FD1 0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID0</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: CONST RD1 0</td>
</tr>
<tr>
<td>DPU1: MOV FD0 ID0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: CONST FD1 0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID0</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction6</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD OD3 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction9</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
</tr>
</tbody>
</table>
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: IF IC0 JMP 9, 8
}
Instruction10 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV FD1 ID2
DPU1: MOV OD1 FD0
DPU2: MOV RD1 ID2
DPU3: MOV RD0 FD0
DPU4: MOV OD2 ID2
CPU0: MOV OC0 IC0
BRU: NEXT
}
Instruction12 {
ADD: ADD FD2 ID3 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: NEXT
}
Instruction14 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: NEXT
}
Instruction16 {
ADD: ADD OD3 FD2 RD2
MUL: NOP
DPU0: NOP
DPU1: NOP
DPU2: NOP
DPU3: NOP
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: NEXT
}
D. FIR Filter Codes

Listing D.3: Final optimized code for \( PE(0, 3) \).

<table>
<thead>
<tr>
<th>Instruction0</th>
<th>Instruction1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: NOP</td>
<td>ADD: NOP</td>
</tr>
<tr>
<td>MUL: NOP</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: CONST RD1 0</td>
<td>DPU0: CONST RD1 0</td>
</tr>
<tr>
<td>DPU1: MOV FD0 ID0</td>
<td>DPU1: MOV FD0 ID0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: CONST FD1 0</td>
<td>DPU3: CONST FD1 0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID0</td>
<td>DPU4: MOV RD0 ID0</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction2</th>
<th>Instruction3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 ID3 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: CONST RD1 0</td>
<td>DPU0: CONST RD1 0</td>
</tr>
<tr>
<td>DPU1: MOV FD0 ID0</td>
<td>DPU1: MOV FD0 ID0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: CONST FD1 0</td>
<td>DPU3: CONST FD1 0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID0</td>
<td>DPU4: MOV RD0 ID0</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: IF IC0 JMP 4, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction4</th>
<th>Instruction5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: CONST RD1 0</td>
<td>DPU0: MOV FD1 ID2</td>
</tr>
<tr>
<td>DPU1: MOV FD0 ID0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
<tr>
<td>DPU2: CONST OD2 0</td>
<td>DPU2: MOV RD1 ID2</td>
</tr>
<tr>
<td>DPU3: CONST FD1 0</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID0</td>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction6</th>
<th>Instruction7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD OD0 FD2 RD2</td>
<td>ADD: ADD FD2 ID3 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
<tr>
<td>DPU2: MOV FD1 FD1</td>
<td>DPU2: MOV FD1 FD1</td>
</tr>
<tr>
<td>DPU3: MOV RD0 FD0</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: NOP</td>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction8</th>
<th>Instruction9</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
</tbody>
</table>
DPU2: MOV FD1 FD1
DPU3: MOV RD0 FD0
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: IF IC0 JMP 9, 8
}
Instruction10 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV FD1 ID2
DPU1: MOV OD1 FD0
DPU2: MOV RD1 ID2
DPU3: MOV RD0 FD0
DPU4: MOV OD2 ID2
CPU0: MOV OC0 IC0
BRU: NEXT
}
Instruction11 {
ADD: ADD OD0 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: MOV FD1 FD1
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: IF IC0 JMP 10, 5
}
Instruction12 {
ADD: ADD FD2 ID3 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: MOV FD1 FD1
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: NEXT
}
Instruction13 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: MOV FD1 FD1
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: NEXT
}
Instruction14 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: MOV FD1 FD1
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: NEXT
}
Instruction15 {
ADD: ADD FD2 FD2 RD2
MUL: NOP
DPU0: NOP
DPU1: NOP
DPU2: NOP
DPU3: NOP
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: NEXT
}
Instruction16 {
ADD: ADD OD0 FD2 RD2
MUL: NOP
DPU0: NOP
DPU1: NOP
DPU2: NOP
DPU3: NOP
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: NEXT
D. FIR Filter Codes

### Listing D.4: Final optimized code for $PE(1, 0)$, and $PE(2, 0)$...

<table>
<thead>
<tr>
<th>Instruction0</th>
<th>Instruction1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: NOP</td>
<td>ADD: NOP</td>
</tr>
<tr>
<td>MUL: NOP</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 ID1</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: NOP</td>
<td>DPU1: MOV RD1 ID4</td>
</tr>
<tr>
<td>DPU2: MOV FD1 ID1</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV FD0 ID5</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: MOV RD0 ID5</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction2</td>
<td>Instruction3</td>
</tr>
<tr>
<td>ADD: ADDI FD2 RD2 0</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV FD1 ID4</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: MOV RD1 ID4</td>
<td>DPU1: MOV RD1 ID4</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV FD0 ID5</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: MOV RD0 ID5</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: IF IC0 JMP 4, 3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction4</td>
<td>Instruction5</td>
</tr>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV FD1 ID4</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: MOV OD2 ID4</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: MOV FD1 ID1</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction6</td>
<td>Instruction7</td>
</tr>
<tr>
<td>ADD: ADD OD3 FD2 RD2</td>
<td>ADD: ADDI FD2 RD2 0</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
<tr>
<td>DPU2: NOPU</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV RD0 FD0</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: NOP</td>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction8</td>
<td>Instruction9</td>
</tr>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
</tbody>
</table>
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: IF IC0 JMP 9, 8
}

Instruction10 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 ID1
  DPU1: MOV OD1 FD0
  DPU2: MOV FD1 ID1
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 ID1
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction11 {
  ADD: ADD OD3 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 FD1
  DPU1: MOV OD1 FD0
  DPU2: NOP
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 FD1
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction12 {
  ADD: ADDI FD2 RD2 0
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 FD1
  DPU1: MOV OD1 FD0
  DPU2: NOP
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 FD1
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction13 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 FD1
  DPU1: MOV OD1 FD0
  DPU2: NOP
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 FD1
  CPU0: MOV OC0 IC0
  BRU: IF IC0 JMP 14, 13
}

Instruction14 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 FD1
  DPU1: MOV OD1 FD0
  DPU2: NOP
  DPU3: NOP
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction15 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: NOP
  DPU1: NOP
  DPU2: NOP
  DPU3: NOP
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction16 {
  ADD: ADD OD3 FD2 RD2
  MUL: NOP
  DPU0: NOP
  DPU1: NOP
  DPU2: NOP
  DPU3: NOP
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}
D. FIR Filter Codes

Listing D.5: Final optimized code for \(PE(1, 1), PE(1, 2), PE(2, 1),\) and \(PE(2, 2).\)

<table>
<thead>
<tr>
<th>Instruction0</th>
<th>Instruction1</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ADD: NOP}</td>
<td>\texttt{ADD: NOP}</td>
</tr>
<tr>
<td>\texttt{MUL: NOP}</td>
<td>\texttt{MUL: MUL RD2 RD0 RD1}</td>
</tr>
<tr>
<td>\texttt{DPU0: MOV RD1 ID6}</td>
<td>\texttt{DPU0: MOV FD1 ID4}</td>
</tr>
<tr>
<td>\texttt{DPU1: NOP}</td>
<td>\texttt{DPU1: MOV RD1 ID4}</td>
</tr>
<tr>
<td>\texttt{DPU2: MOV FD1 ID6}</td>
<td>\texttt{DPU2: NOP}</td>
</tr>
<tr>
<td>\texttt{DPU3: MOV FD0 ID5}</td>
<td>\texttt{DPU3: MOV FD0 ID5}</td>
</tr>
<tr>
<td>\texttt{DPU4: MOV RD0 ID5}</td>
<td>\texttt{DPU4: MOV RD0 ID5}</td>
</tr>
<tr>
<td>\texttt{CPU0: MOV OC0 IC0}</td>
<td>\texttt{CPU0: MOV OC0 IC0}</td>
</tr>
<tr>
<td>\texttt{BRU: NEXT}</td>
<td>\texttt{BRU: NEXT}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction2</th>
<th>Instruction3</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ADD: ADD FD2 ID3 RD2}</td>
<td>\texttt{ADD: ADD FD2 FD2 RD2}</td>
</tr>
<tr>
<td>\texttt{MUL: MUL RD2 RD0 RD1}</td>
<td>\texttt{MUL: MUL RD2 RD0 RD1}</td>
</tr>
<tr>
<td>\texttt{DPU0: MOV FD1 ID4}</td>
<td>\texttt{DPU0: MOV FD1 ID4}</td>
</tr>
<tr>
<td>\texttt{DPU1: MOV RD1 ID4}</td>
<td>\texttt{DPU1: MOV RD1 ID4}</td>
</tr>
<tr>
<td>\texttt{DPU2: NOP}</td>
<td>\texttt{DPU2: NOP}</td>
</tr>
<tr>
<td>\texttt{DPU3: MOV FD0 ID5}</td>
<td>\texttt{DPU3: MOV FD0 ID5}</td>
</tr>
<tr>
<td>\texttt{DPU4: MOV RD0 ID5}</td>
<td>\texttt{DPU4: MOV RD0 ID5}</td>
</tr>
<tr>
<td>\texttt{CPU0: MOV OC0 IC0}</td>
<td>\texttt{CPU0: MOV OC0 IC0}</td>
</tr>
<tr>
<td>\texttt{BRU: NEXT}</td>
<td>\texttt{BRU: IF IC0 JMP 4, 3}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction4</th>
<th>Instruction5</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ADD: ADD FD2 FD2 RD2}</td>
<td>\texttt{ADD: ADD FD2 FD2 RD2}</td>
</tr>
<tr>
<td>\texttt{MUL: MUL RD2 RD0 RD1}</td>
<td>\texttt{MUL: MUL RD2 RD0 RD1}</td>
</tr>
<tr>
<td>\texttt{DPU0: MOV FD1 ID4}</td>
<td>\texttt{DPU0: MOV FD1 ID4}</td>
</tr>
<tr>
<td>\texttt{DPU1: MOV RD1 ID4}</td>
<td>\texttt{DPU1: MOV FD0 FD0}</td>
</tr>
<tr>
<td>\texttt{DPU2: MOV OD2 ID4}</td>
<td>\texttt{DPU2: MOV RD1 ID2}</td>
</tr>
<tr>
<td>\texttt{DPU3: MOV FD0 ID5}</td>
<td>\texttt{DPU3: MOV RD0 FD0}</td>
</tr>
<tr>
<td>\texttt{DPU4: MOV RD0 ID5}</td>
<td>\texttt{DPU4: NOP}</td>
</tr>
<tr>
<td>\texttt{CPU0: MOV OC0 IC0}</td>
<td>\texttt{CPU0: MOV OC0 IC0}</td>
</tr>
<tr>
<td>\texttt{BRU: NEXT}</td>
<td>\texttt{BRU: NEXT}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction6</th>
<th>Instruction7</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ADD: ADD OD3 FD2 RD2}</td>
<td>\texttt{ADD: ADD FD2 ID3 RD2}</td>
</tr>
<tr>
<td>\texttt{MUL: MUL RD2 RD0 RD1}</td>
<td>\texttt{MUL: MUL RD2 RD0 RD1}</td>
</tr>
<tr>
<td>\texttt{DPU0: MOV RD1 FD1}</td>
<td>\texttt{DPU0: MOV RD1 FD1}</td>
</tr>
<tr>
<td>\texttt{DPU1: MOV FD0 FD0}</td>
<td>\texttt{DPU1: MOV FD0 FD0}</td>
</tr>
<tr>
<td>\texttt{DPU2: NOP}</td>
<td>\texttt{DPU2: NOP}</td>
</tr>
<tr>
<td>\texttt{DPU3: MOV RD0 FD0}</td>
<td>\texttt{DPU3: MOV RD0 FD0}</td>
</tr>
<tr>
<td>\texttt{DPU4: NOP}</td>
<td>\texttt{DPU4: NOP}</td>
</tr>
<tr>
<td>\texttt{CPU0: MOV OC0 IC0}</td>
<td>\texttt{CPU0: MOV OC0 IC0}</td>
</tr>
<tr>
<td>\texttt{BRU: NEXT}</td>
<td>\texttt{BRU: NEXT}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction8</th>
<th>Instruction9</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ADD: ADD FD2 FD2 RD2}</td>
<td>\texttt{ADD: ADD FD2 FD2 RD2}</td>
</tr>
<tr>
<td>\texttt{MUL: MUL RD2 RD0 RD1}</td>
<td>\texttt{MUL: MUL RD2 RD0 RD1}</td>
</tr>
<tr>
<td>\texttt{DPU0: MOV RD1 FD1}</td>
<td>\texttt{DPU0: MOV RD1 FD1}</td>
</tr>
<tr>
<td>\texttt{DPU1: MOV FD0 FD0}</td>
<td>\texttt{DPU1: MOV FD0 FD0}</td>
</tr>
<tr>
<td>Instruction10</td>
<td>Instruction11</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
</tr>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD OD3 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV FD1 ID2</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV OD1 FD0</td>
<td>DPU1: MOV OD1 FD0</td>
</tr>
<tr>
<td>DPU2: MOV RD1 ID2</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV RD0 FD0</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: MOV OD2 ID2</td>
<td>DPU4: MOV OD2 FD1</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction12</th>
<th>Instruction13</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV OD1 FD0</td>
<td>DPU1: MOV OD1 FD0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV RD0 FD0</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: MOV OD2 FD1</td>
<td>DPU4: MOV OD2 FD1</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: IF IC0 JMP 14, 13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction14</th>
<th>Instruction15</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: NOP</td>
</tr>
<tr>
<td>DPU1: MOV OD1 FD0</td>
<td>DPU1: NOP</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV RD0 FD0</td>
<td>DPU3: NOP</td>
</tr>
<tr>
<td>DPU4: MOV OD2 FD1</td>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction16</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD OD3 FD2 RD2</td>
</tr>
<tr>
<td>MUL: NOP</td>
</tr>
<tr>
<td>DPU0: NOP</td>
</tr>
<tr>
<td>DPU1: NOP</td>
</tr>
<tr>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: NOP</td>
</tr>
<tr>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>
### D. FIR Filter Codes

#### Listing D.6: Final optimized code for \( PE(1,3) \), and \( PE(2,3) \).

<table>
<thead>
<tr>
<th>Instruction0 {</th>
<th>Instruction1 {</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: NOP</td>
<td>ADD: NOP</td>
</tr>
<tr>
<td>MUL: NOP</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 ID6</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: NOP</td>
<td>DPU1: MOV RD1 ID4</td>
</tr>
<tr>
<td>DPU2: MOV FD1 ID6</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV FD0 ID5</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: MOV RD0 ID5</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction2 {</th>
<th>Instruction3 {</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 ID3 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV FD1 ID4</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: MOV RD1 ID4</td>
<td>DPU1: MOV RD1 ID4</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV FD0 ID5</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: MOV RD0 ID5</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: IF IC0 JMP 4, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction4 {</th>
<th>Instruction5 {</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV FD1 ID4</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: MOV RD1 ID4</td>
<td>DPU1: MOV RD1 ID4</td>
</tr>
<tr>
<td>DPU2: MOV OD2 ID4</td>
<td>DPU2: MOV OD2 ID4</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV FD0 ID5</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: MOV RD0 ID5</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction6 {</th>
<th>Instruction7 {</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD OD0 FD2 FD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV RD0 FD0</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: NOP</td>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction8 {</th>
<th>Instruction9 {</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
</tbody>
</table>
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: IF IC0 JMP 9, 8
}

Instruction10 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV FD1 ID2
DPU1: MOV OD1 FD0
DPU2: MOV RD1 ID2
DPU3: MOV RD0 FD0
DPU4: MOV OD2 ID2
CPU0: MOV OC0 IC0
BRU: NEXT
}

Instruction11 {
ADD: ADD OD0 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: IF IC0 JMP 10, 5
}

Instruction12 {
ADD: ADD FD2 ID3 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: NEXT
}

Instruction13 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: IF IC0 JMP 14, 13
}

Instruction14 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: MOV RD1 FD1
DPU1: MOV OD1 FD0
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: MOV OD2 FD1
CPU0: MOV OC0 IC0
BRU: NEXT
}

Instruction15 {
ADD: ADD FD2 FD2 RD2
MUL: MUL RD2 RD0 RD1
DPU0: NOP
DPU1: NOP
DPU2: NOP
DPU3: NOP
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: NEXT
}

Instruction16 {
ADD: ADD OD0 FD2 RD2
MUL: NOP
DPU0: NOP
DPU1: NOP
DPU2: NOP
DPU3: NOP
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: NEXT
}
### D. FIR Filter Codes

<table>
<thead>
<tr>
<th>Instruction0</th>
<th>Instruction1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: NOP</td>
<td>ADD: NOP</td>
</tr>
<tr>
<td>MUL: NOP</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 ID1</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: NOP</td>
<td>DPU1: MOV RD1 ID4</td>
</tr>
<tr>
<td>DPU2: MOV FD1 ID1</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV FD0 ID5</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: MOV RD0 ID5</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction2</th>
<th>Instruction3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADDI FD2 RD2 0</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV FD1 ID4</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: MOV RD1 ID4</td>
<td>DPU1: MOV RD1 ID4</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV FD0 ID5</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: MOV RD0 ID5</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: IF IC0 JMP 4, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction4</th>
<th>Instruction5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV FD1 ID4</td>
<td>DPU0: MOV RD1 ID1</td>
</tr>
<tr>
<td>DPU1: MOV FD1 ID4</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
<tr>
<td>DPU2: MOV OD2 ID4</td>
<td>DPU2: MOV FD1 ID1</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction6</th>
<th>Instruction7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD OD3 FD2 RD2</td>
<td>ADD: ADDI FD2 RD2 0</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV RD0 FD0</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: NOP</td>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction8</th>
<th>Instruction9</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
</tbody>
</table>

242
Listing D.8: Final optimized code for PE(3, 1), and PE(3, 2).

Instruction0 {
  ADD: NOP
  MUL: NOP
  DPU0: MOV RD1 ID6
  DPU1: NOP
  DPU2: MOV FD1 ID6
  DPU3: MOV FD0 ID5
  DPU4: MOV RD0 ID5
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction1 {
  ADD: NOP
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 ID4
  DPU1: MOV RD1 ID4
  DPU2: NOP
  DPU3: MOV FD0 ID5
  DPU4: MOV RD0 ID5
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction2 {
  ADD: ADD FD2 ID3 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 ID4
  DPU1: MOV RD1 ID4
  DPU2: NOP
  DPU3: MOV FD0 ID5
  DPU4: MOV RD0 ID5
  CPU0: MOV OC0 IC0
  BRU: IF IC0 JMP 4, 3
}

Instruction3 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 ID4
  DPU1: MOV RD1 ID4
  DPU2: NOP
  DPU3: MOV FD0 ID5
  DPU4: MOV RD0 ID5
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction4 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 ID4
  DPU1: MOV RD1 ID4
  DPU2: NOP
  DPU3: MOV FD0 ID5
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction5 {
  ADD: ADD FD2 FD2 FD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 ID4
  DPU1: MOV FD0 FD0
  DPU2: MOV OD2 ID4
  DPU3: MOV RD0 FD0
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction6 {
  ADD: ADD OD3 FD2 FD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 FD1
  DPU1: MOV FD0 FD0
  DPU2: MOV FD1 FD1
  DPU3: MOV RD0 FD0
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction7 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 FD1
  DPU1: MOV FD0 FD0
  DPU2: MOV FD1 FD1
  DPU3: MOV RD0 FD0
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction8 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 FD1
  DPU1: MOV FD0 FD0
  DPU2: MOV FD1 FD1
  DPU3: MOV RD0 FD0
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction9 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 FD1
  DPU1: MOV FD0 FD0
  DPU2: MOV FD1 FD1
  DPU3: MOV RD0 FD0
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}
Instruction10 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 ID2
  DPU1: NOP
  DPU2: MOV RD1 FD1
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 ID2
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction11 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 ID2
  DPU1: NOP
  DPU2: MOV FD1 FD1
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 FD1
  CPU0: MOV OC0 IC0
  BRU: IF IC0 JMP 10, 5
}

Instruction12 {
  ADD: ADD FD2 ID3 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 ID2
  DPU1: NOP
  DPU2: MOV FD1 FD1
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 FD1
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction13 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 FD1
  DPU1: NOP
  DPU2: MOV FD1 FD1
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 FD1
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction14 {
  ADD: ADD OD3 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 FD1
  DPU1: NOP
  DPU2: MOV FD1 FD1
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 FD1
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction15 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 FD1
  DPU1: NOP
  DPU2: MOV FD1 FD1
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 FD1
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction16 {
  ADD: ADD OD3 FD2 RD2
  MUL: NOP
  DPU0: NOP
  DPU1: NOP
  DPU2: NOP
  DPU3: NOP
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}
Listing D.9: Final optimized code for $PE(3, 3)$.

<table>
<thead>
<tr>
<th>Instruction0</th>
<th>Instruction1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: NOP</td>
<td>ADD: NOP</td>
</tr>
<tr>
<td>MUL: NOP</td>
<td>MUL: MUL RD1 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 ID6</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: NOP</td>
<td>DPU1: MOV RD1 ID4</td>
</tr>
<tr>
<td>DPU2: MOV FD1 ID6</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV FD0 ID5</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: MOV RD0 ID5</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction2</th>
<th>Instruction3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 ID3 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV FD1 ID4</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: MOV RD1 ID4</td>
<td>DPU1: MOV RD1 ID4</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV FD0 ID5</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: MOV RD0 ID5</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: IF IC0 JMP 4, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction4</th>
<th>Instruction5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV FD1 ID4</td>
<td>DPU0: MOV FD1 ID4</td>
</tr>
<tr>
<td>DPU1: MOV RD1 ID4</td>
<td>DPU1: MOV RD1 ID4</td>
</tr>
<tr>
<td>DPU2: MOV OD2 ID4</td>
<td>DPU2: MOV RD1 ID2</td>
</tr>
<tr>
<td>DPU3: MOV FD0 ID5</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: MOV RD0 ID5</td>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction6</th>
<th>Instruction7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD OD2 FD2 RD2</td>
<td>ADD: ADD FD2 ID3 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
<tr>
<td>DPU2: NOP</td>
<td>DPU2: NOP</td>
</tr>
<tr>
<td>DPU3: MOV RD0 FD0</td>
<td>DPU3: MOV RD0 FD0</td>
</tr>
<tr>
<td>DPU4: NOP</td>
<td>DPU4: NOP</td>
</tr>
<tr>
<td>CPU0: MOV OC0 IC0</td>
<td>CPU0: MOV OC0 IC0</td>
</tr>
<tr>
<td>BRU: NEXT</td>
<td>BRU: NEXT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction8</th>
<th>Instruction9</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD: ADD FD2 FD2 RD2</td>
<td>ADD: ADD FD2 FD2 RD2</td>
</tr>
<tr>
<td>MUL: MUL RD2 RD0 RD1</td>
<td>MUL: MUL RD2 RD0 RD1</td>
</tr>
<tr>
<td>DPU0: MOV RD1 FD1</td>
<td>DPU0: MOV RD1 FD1</td>
</tr>
<tr>
<td>DPU1: MOV FD0 FD0</td>
<td>DPU1: MOV FD0 FD0</td>
</tr>
</tbody>
</table>
DPU2: NOP
DPU3: MOV RD0 FD0
DPU4: NOP
CPU0: MOV OC0 IC0
BRU: IF IC0 JMP 9, 8
)

Instruction10 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 ID2
  DPU1: NOP
  DPU2: MOV RD1 ID2
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 ID2
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction12 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV FD1 ID3 RD2
  DPU1: NOP
  DPU2: NOP
  DPU3: MOV RD0 FD0
  DPU4: MOV OD2 FD1
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction14 {
  ADD: ADD FD2 FD2 RD2
  MUL: MUL RD2 RD0 RD1
  DPU0: MOV RD1 FD1
  DPU1: NOP
  DPU2: NOP
  DPU3: NOP
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}

Instruction16 {
  ADD: ADD OD0 FD2 RD2
  MUL: NOP
  DPU0: NOP
  DPU1: NOP
  DPU2: NOP
  DPU3: NOP
  DPU4: NOP
  CPU0: MOV OC0 IC0
  BRU: NEXT
}
Global Controller: Implementation

This appendix focuses more on the hardware implementation of the Global Controller (GC). First, we describe the Iteration Space Scanner (ISS) followed by Control Signal Generator (CSG).

E.1 Iteration Space Scanner

In this section, detailed implementation of different sub blocks in the ISS is discussed. One of the important requirements of the hardware is that it should be programmable—depending on the algorithm—the number of iteration variables, stride vectors, stride condition spaces, and initial iteration of the tile will change which are programmed in the design. As this design is used in programmable accelerators, even though the design is synthesized for higher number of iteration variables or tile dimensions, at runtime the user should have the option to select the desired number of iteration variables or dimensions. Depending on the chosen number of dimensions, the design should also be reconfigured automatically for the required number of stride vectors and stride condition spaces. For instance, if the dimension of the tile is two, as shown in Figure 5.3, two stride vectors and for each stride vector a condition space is anticipated. In case of a three dimensional tile, three stride vectors and three stride condition spaces are required to scan the tile iteration space which should get adjusted automatically once the user selects the dimensions as per the application needs. Therefore, a maximum number of dimensions is specified by the parameter DIMENSION in the design at synthesis time. However, in the initializer block, by programming a specific register (more details in Appendix E.1.3) required number of dimensions can be selected at runtime. The required number of bits to represent each iteration variable width is specified using the parameter ITERATION_VARIABLE_WIDTH at synthesis time. In the following, we discuss stride determinator, stride selector, initializer, clock divider, reinitializer, and control Finite State Machine (FSM) blocks in detail.
E.1 Iteration Space Scanner

E.1.1 Stride Determinator

Given a current iteration, this block’s main functionality is to determine its stride condition space. Based on this information, this block will output a select signal for the stride selector block where the next iteration is computed from the current iteration by adding a stride vector which is selected based on the select signal from this block. In the following text, the brief functionality and conceptual architecture of this block is discussed using the architectural diagram shown in Figure E.1.

This block contains few registers that need to be programmed which are also shown in the block diagram. These registers store the stride condition spaces and iteration variable select registers for the min/max comparators, detail to follow next. For instance, in Example 5.1 there are two stride condition spaces one for each stride vectors $I_1^{scs} = \{0 \leq j_1 \leq 6\}$ and $I_2^{scs} = \{0 \leq i_1 \leq 6 \land j_1 = 7\}$. The current iteration is to be checked against these two stride condition spaces to find out its stride condition space. This checking can be done in parallel using the min/max comparator blocks as shown in the block diagram where each row of the comparator blocks implement a stride condition space. Each comparator block in a row implements a portion of the stride condition space, for instance, $I_2^{scs} = \{0 \leq i_1 \leq 6 \land j_1 = 7\}$ is implemented by second row of the min/max comparators block in which first comparator may programmed with $0 \leq i_1 \leq 6$ and the second comparator may be programmed with $j = 7$. Now, from the current iteration $i_1$ is connected to the first comparator and $j_1$ is connected to

![Figure E.1: A functional architectural block diagram of the stride determinator.](image-url)
the second comparator and output of these row comparators is ANDed together. If the current iteration belongs to this stride condition space, its corresponding stride select bit value will go high.

The functionality of the min/max comparator block is shown in Figure E.2, where the current iteration value is checked whether it is in the min and max bounds of the corresponding iteration variable bounds from a stride condition space. As mentioned before, programmable registers in this block hold the min \((\text{ivar\_min\_reg})\) and max \((\text{ivar\_max\_reg})\) bounds for an iteration variable in the stride condition space and an iteration variable select register \((\text{ivar\_sel\_reg})\) selects that iteration variable. A quick look at the stride condition space reveals that in each stride condition space an extra iteration variable is introduced. For instance, for the first stride condition space \(\mathcal{I}_{scs} = \{0 \leq j_1 \leq 6\}\) in which only iteration variable \(j_1\) is considered irrespective of the value of the \(i_1\), see Figure 5.3. For the next stride condition space \(\mathcal{I}^2_{scs} = \{0 \leq i_1 \leq 6 \land j_1 = 7\}\) bounds on both iteration variables need to be checked. In fact, this is similar to executing iterations in any nested for loop where innermost loop iterations are executed first and once the bound is reached on this innermost loop, next level of the loop iteration variable is considered. In this example, the scanning order of the iteration variables is \(j_1\) followed by \(i_1\). In other words, innermost loop is \(j_1\) and outermost loop is \(i_1\). Therefore, in the block diagram shown in Figure E.1 min/max comparators arranged in a lower triangular matrix fashion—where in each row—an additional comparator is added to check the bounds on the extra iteration variable of a stride condition space compared to its previous row. In this way, hardware cost of required comparators is reduced and unused registers that are declared in the stride determinator block are pruned away during the synthesis. It is to be noted that there are still some optimizations possible in the stride determinator which might reduce the comparators further.

In stride determinator block, as mentioned before, at runtime, the user can select the required number of dimensions using the output selector signal and depending on that the appropriate number of rows of comparator matrix will be active. Furthermore, only the required number of registers in this block are programmed, one after the other, in a serial fashion through the configuration bus \((\text{conf\_bus})\) and configuration clock \((\text{conf\_clk})\) interface. Once this sub block is programmed, a configuration acknowledgment \((\text{conf\_ack})\) is also raised. The maximum number of bits to configure this is block, denoted by \(N^{SD}_{bits}\), depends on the number of min/max comparators instantiated in the design and the number of bits required to configure a min/max comparator as given by Equation (E.1).

\[
N^{SD}_{bits} = \sum_{i=1}^{\text{DIMENSION}} i \cdot (2 \cdot \text{ITERATION VARIABLE WIDTH} + \text{DIMENSION} + 1) \quad \text{(E.1)}
\]
The maximum number of cycles to configure the registers in this block, denoted by $N_{SD\text{cycles}}$ and given in Equation (E.2), also depends on the number of comparators and the number of cycles to configure a comparator, which is four cycles as there are four registers to be configured.

$$N_{SD\text{cycles}} = 4 \cdot \sum_{i=1}^{DIMENSION} i$$

(E.2)

Further details on the configuration scheme and its hardware are given in Section 5.4 and Appendix E.3.

**Figure E.2:** A structural view of the min/max comparator architecture.

### E.1.2 Stride Selector

In this block, the current iteration is added to a stride vector to determine the next iteration. All the path strides are stored in registers in this block. Depending on the stride select signal from the stride determinator block, an appropriate stride vector is selected and added to the current iteration to calculate the next iteration as shown in the conceptual architecture block diagram of Figure E.3. In this block, only programmable registers are path stride matrix registers, which get programmed through the configuration bus (conf_bus) and configuration clock (conf_clk) interface. Once this block is programmed, a configuration acknowledgment (conf_ack) is also raised. The maximum number of bits to configure this is block, denoted by $N_{SS\text{bits}}$, depends on the number of stride vectors which in turn depends on the dimension of the tile as given in Equation (E.3).

$$N_{SS\text{bits}} = DIMENSION \times DIMENSION \times \text{ITERATION VARIABLE WIDTH}$$

(E.3)

The maximum number of cycles to configure the registers in this block, denoted by $N_{SS\text{cycles}}$, can be calculated using the Equation (E.4).

$$N_{SS \text{cycles}} = DIMENSION \times DIMENSION$$

(E.4)
Figure E.3: Conceptual architecture block diagram of the stride selector block.

Further details on the configuration scheme and its hardware are given in Section 5.4. This block is also designed in a way that at runtime depending on the tile dimensions, user can select the number of stride vectors in the path stride matrix. For instance, assuming the dimension of the tile is two, a $2 \times 2$ path stride matrix is selected and gets programmed by using the output selector (output_selector) signal even though the hardware might have synthesized for more than two dimensions.

E.1.3 Initializer

In this initializer block, few programmable registers are present that drive few important signals in the whole design. In order to start scanning the iteration space, according to Algorithm 5.1, an initial iteration is required which is stored in the init_iivar register in this block as shown in Figure E.4. During the synthesis of the design, one can select as many dimensions as needed, which denote the number of iteration variables of the tile, using the parameter DIMENSION. However, at runtime, by programming the output_selector register appropriate number of dimensions can be selected as per the current application requirements. Once the ISS reaches its final iteration, i.e., tile execution is finished, the ISS can return to its first iteration and re-scan the whole iteration space again or it can stop. This behavior can be controlled by programming the restart_mode, a single bit register in this sub block. If this register bit is programmed with a true value then ISS restarts after reaching the final iteration and if it is programmed with a false value, it will stop after the final iteration. In the CSG block (see Appendix E.2) where the actual control signals to orches-
E.1 Iteration Space Scanner

Figure E.4: Block diagram of the initializer showing different programmable registers.

The register values and control signals needed to determine the program execution are generated, one can select the desired number of program blocks by programming the pb_selector register even though during the synthesis time maximum number of program blocks are specified by the parameter MAX_NO_OF_PROGRAM_BLOCKS. Furthermore, the maximum number of control signals during synthesis time are specified by the parameter NO_OF_IC_SIGNALS. Depending on the application, at runtime, an appropriate number of control signals are selected by programming the ic_selector register whereas the initial values of these control signals are specified by programming the init_ic register, see Figure E.4. All the registers in this block are also programmed using the configuration bus and configuration clock interface similar to all the other blocks. The maximum number of bits to configure this is block, denoted by $N_{bits}^{IN}$, is given by Equation (E.5).

$$N_{bits}^{IN} = \text{DIMENSION} \times \text{ITERATION_VARIABLE_WIDTH} + \text{DIMENSION} + 2 \times \text{NO_OF_IC_SIGNALS} + \text{MAX_NO_OF_PROGRAM_BLOCKS} + 1 \quad (E.5)$$

The maximum number of cycles to configure the registers in this block, denoted by $N_{cycles}^{IN}$, can be calculated using the Equation (E.6).

$$N_{cycles}^{IN} = \text{DIMENSION} + \left\lceil \frac{\text{MAX_NO_OF_PROGRAM_BLOCKS}}{\text{CONFIG_BUS_WIDTH}} \right\rceil + 4 \quad (E.6)$$

E.1.4 Clock Divider

In this section, we explain the procedure of generating the clock that depends on the iteration interval using the block diagram shown in Figure E.5. A programmable register named divide register (div_reg) in this block is programmed
with the configuration bus and configuration clock interface with an appropriate value of the iteration interval. This is the only single register which gets programmed in this block. Once the programming is done, a configuration acknowledgment is also raised to notify the configuration logic. The maximum number of bits required to configure the registers in this block is denoted by \( N_{\text{bits}}^{CD} \), given by Equation (E.7).

\[
N_{\text{bits}}^{CD} = \text{iteration\_variable\_width}
\]  

(E.7)

Only one configuration cycle is enough to configure this register as given in Equation (E.8).

\[
N_{\text{cycle}}^{CD} = 1
\]  

(E.8)

Based on this divide register value, an appropriate clock needs to be generated at runtime. For this purpose, we decided to use the Digital Clock Manager (DCM) which is already available in the Field-Programmable Gate Array (FPGA) and generate one base clock for the Tightly Coupled Processor Array (TCPA) and at the same time generate the GC clock based on the divide register value. In general, DCMs are used to generate the fixed clocks in all the FPGA-based emulation platforms and in our case, we need to generate a GC clock at runtime depending on the iteration interval. For this purpose, DCM already provides a Dynamic Reconfiguration Port (DRP) by which few registers in the DCM are programmed at runtime to generate a clock that is either multiplied clock or divided clock of the input clock. This DRP has specific D* signals as shown in Figure E.5, which can be used to write internal registers in the DCM at specific addresses. It is to be noted that any programmable functional block in an FPGA has a memory, out of which some address locations are read only and some other address locations are write only. However, total content of this memory determines the configuration of the functional block. This DRP is synchronous parallel memory port, with separate read and write buses. Synchronous timing for this interface is provided by the DCLK input and all the other input signals are registered in the DCM on the rising edge of DCLK. The data to be written is presented simultaneously with the write address DADDR, DWE and DEN signals prior to the next positive edge of DCLK. The interface acknowledges for one clock cycle when it is ready to accept more data. The timing requirements relative to DCLK for all other signals are the same. The timing relationship between the different signals of the interface for a write operation are shown in Figure E.7. In case of reading from DCM internal registers, data is available after few cycles following the cycle that DEN and DADDR are asserted. It is to be noted that all the operations on this interface are enabled by DEN and for write operations DWE is active high whereas for read operations DWE is active low. Once the read operation is issued, the availability of the output data is indicated by the assertion of DRDY and the data itself is available on the
E.1 Iteration Space Scanner

Figure E.5: Architectural block diagram of clock divider showing the Digital Clock Manager (DCM), Dynamic Reconfiguration Port (DRP) and the DRP_FSM which controls the reads/writes into the internal registers of the Digital Clock Manager (DCM).

DO bus. The timing relationship between the different signals of the interface for a read operation are shown in Figure E.6. DCM has multiply and divide (M/D) values which can be programmed using DRP, as mentioned before, by reading from and writing to hex address 50h. One must read from address 50h first if either the M or the D value is changed. The M value must be masked into the most significant byte and the D value must be masked into the least significant byte of the 16-bit word. The entire word is then written back—even though both values are changed at the same time, only a single write cycle is required. DCM must be held in reset state by activating the input reset while changing the M/D values. If the M/D values are changed dynamically, few bits of the address 41h and 51h must be set or reset depending on the high frequency mode or low frequency mode. Once all the read/writes are finished, input reset signal is released, signal locked will go high indicating that the clock output of
the DCM is stable. For further details on this read/write sequence, we refer to [Xil12].

In order to implement these read/write sequences, a FSM called DRP_FSM is implemented as shown in Figure E.8. In the state S0:RESET of the FSM, reset input to the DCM is triggered high to force it into the reset state to program appropriate M/D values. Meanwhile, the divide register in this block gets programmed with an appropriate iteration interval through the configuration bus interface. Once it gets programmed, the configuration acknowledgment (conf_ack) goes high—making the state transition to S1:READ_PREPARE
E.1 Iteration Space Scanner

from S0:RESET. It is to be noted that the state transitions in this FSM are synchronous with the DCLK which is relatively a slower clock. With respect to the discussion above, there are three address locations in total that need to be read first and then written back with a modified data. In the DRP_FSM, S1 $\rightarrow$ S5 states performs a read followed by a write; thus, states S1 $\rightarrow$ S5 are traversed in total three times, each time with a distinct address controlled by the count signal. For instance, a zero value for the count indicates read/write to the address 50h. In S1:READ state, a read is issued from address 50h as the initial value of the count is zero. DEN goes active high whereas the D WE signal is active low since this is a read operation. Once the read is issued, state transition to S2:READ_WAIT state in the next clock cycle where acknowledgment of high DRDY signal indicates that the read is complete and data is available on the DO bus. A read operation might take some time which introduces the wait states as shown in the read timing diagram, see Figure E.6. Acknowledgment of read by DRDY signal causes the state transition from S2:READ_WAIT state to S3:PROCESS_DATA where appropriate data is prepared to write in the next cycle.

For the first write operation, multiply and divide (M/D) values are selected based on iteration interval programmed in divide register as shown in Table E.1. M and D values are concatenated together and put on the DI bus. In the next cycle, in state S4:WRITE, a write operation is issued by asserting DEN and DWE along with DADDR of 50h. Once the write is issued, in the next cycle DRP_FSM transitions from S4:WRITE to S4:WRITE_WAIT where acknowledgment of high DRDY signal indicates that the write is complete. In this way, a read followed by write is completed for an address and then the count value is increased by one to select a different address for the next read followed by write. A write operation also may take some time which introduces the wait states as shown in the write timing diagram, see Figure E.7. Acknowledgment of write by DRDY signal causes the state transition to S1:READ to carry out the next read followed by write operation which follows the same procedure as explained above. In the next two read followed by write operations, address 41h and 51h are read and written back with a modified data as shown in Figure E.8. In these two write operations, only few bits of the whole word are modified as per the functional block description of DCM [Xil12].

Finally, when all the three address locations are read followed by a write with modified data or the count value is three, state transition from S1:READ state to S6:DRP_DONE with reset being released so that the DCM is locked after some time where the clock is stable. Once it is locked, state transitions to S7:LOCKED state and a signal named locked goes high notifying the control FSM hardware block to be discussed in Appendix E.1.6. To summarize, by using the DRP_FSM and with DRP of the DCM an appropriate GC clock is generated that depends on the II.
Figure E.8: State diagram of DRP_FSM which performs a read followed by a write to specific addresses in the internal registers of the DCM using the DRP.
**Table E.1**: Multiply (M) and divide (D) register values depending on the iteration interval.

<table>
<thead>
<tr>
<th>Iteration interval</th>
<th>Multiply (M)</th>
<th>Divide (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01h</td>
<td>01h</td>
</tr>
<tr>
<td>2</td>
<td>01h</td>
<td>03h</td>
</tr>
<tr>
<td>3</td>
<td>01h</td>
<td>05h</td>
</tr>
<tr>
<td>4</td>
<td>01h</td>
<td>07h</td>
</tr>
<tr>
<td>5</td>
<td>01h</td>
<td>09h</td>
</tr>
<tr>
<td>6</td>
<td>01h</td>
<td>0Bh</td>
</tr>
<tr>
<td>7</td>
<td>01h</td>
<td>0Dh</td>
</tr>
<tr>
<td>8</td>
<td>01h</td>
<td>0Fh</td>
</tr>
</tbody>
</table>

The clock divider logic can also be implemented with a counter logic whose output value is compared with the iteration interval value programmed in the divided register to generate an enable signal whenever both values are equal. This enable signal can be used further in the required hardware blocks to register inputs which essentially implemented as clock gating logic after the synthesis is performed. Later on, this clock gating cells are replaced appropriate glitch free integrated clock gating cells in the same synthesis step. This solution is also straight forward, however, when closing the timing, worst case scenario has to be considered where the iteration interval is one. The same clock generated here in the GC is also driving the address generators and reconfigurable buffers surrounding the array, which are implementing complex arithmetic operations while calculating the addresses and address enable signals. In those hardware blocks, timing might be critical. In case of timing violations, the design must be run with a higher iteration interval and most importantly multi-cycle path timing constraints should be introduced and adapted according to the iteration interval so that static timing analysis does not show any violations. In case of the design with the DCM, there is no need to update the multi-cycle path timing constraints and default setup and hold checks are sufficient to close the timing as the clock itself adapts according to the II. As we are using the FPGAs for demonstrating our designs, we ended up in the design with the DCM otherwise there is no particular reason for preferring this solution over the other. This solution looks bit complex at a first look, however, once the DRP is understood, implementing the DRP_FSM for reading and writing to specific address locations is very straightforward.
E.1.5 Reinitializer

The main functionality of this sub block is to hold the final iteration point and compare it against the current iteration value. Once the final iteration is reached, this block raises the `reinitialize` signal notifying the rest of the hardware blocks. In this block, registers holding the final iteration point are programmed through the configuration bus and configuration clock interface, similar to all the other blocks. Once the programming of this block is done, a configuration acknowledgment is also raised to notify the configuration logic. The maximum number of bits required to configure the registers in this block, denoted by $N_{RI\text{bits}}$, are given by Equation (E.9). Similarly, the number of cycles to program all registers in this block are calculated using Equation (E.10).

\[
N_{RI\text{bits}} = \text{DIMENSION}-\text{ITERATION}\_\text{VARIABLE}\_\text{WIDTH} \quad (E.9)
\]

\[
N_{RI\text{cycle}} = \text{DIMENSION} \quad (E.10)
\]

A conceptual architecture block diagram is shown in Figure E.9, which contains an equality comparator for each iteration variable. The number of required comparators depend on the dimension of the tile which is already specified as a parameter in this block. Furthermore, at runtime, appropriate number of dimensions can be selected using the output selector signal even though the design is synthesized for higher number of dimensions at synthesis time.

![Reinitializer Block Diagram](image)

**Figure E.9:** A conceptual architecture block diagram of the reinitializer sub block for a three dimensional tile.
E.1.6 Control FSM

In this section, details on the control FSM whose states are controlled by the GC clock which in turn depends on the II is discussed in detail. A functional block diagram of this control FSM is shown in Figure E.10. In the following, the functionality of this control FSM is briefly explained. This FSM has four states called S0:CONFIGURE, S1:INITIAL, S2:RUN, and S3:F I N A L as shown in the state diagram. Once the design is reset, the FSM is in the CONFIGURE state and it issues the configuration enable (conf_en) signal for the configuration scheme (to be discussed in Appendix E.3) so that all different blocks in the GC design get programmed. Control stays in this state until the configuration is done and DCM discussed in the last section gets locked to the desired iteration interval clock. In the whole TCPA architecture, there are different programmable modules such as address generators, global controller, reconfigurable buffers, and Processing Element (PE)s (programs and interconnects needs to be configured), which have to be programmed one after the other in a sequential

![Control FSM Diagram](image)

**Figure E.10**: Functional state diagram of the control FSM hardware block.
manner before starting any useful computation. Once all these modules are programmed, a glue logic, which checks for the configuration acknowledgment from each of these modules, issues a *start* signal so that every module is working in a lock step mode. Furthermore, in case of any buffer is empty or it has to be drained, the whole computation in the TCPA must be stopped and even the ISS must be stopped at the current iteration itself. In accordance with this behavior, a *stop* signal is issued by the buffer logic to notify the GC to stop computing the next iteration further, unless it goes high again. The control FSM will wait in the S0:CONFIGURE state for appropriate start and stop signal values.

Once the start signal goes high and the stop signal is active low, it will transition to the S1:INITIAL state, where the current iteration is initialized with the initial iteration. In this cycle, stride determinator and stride selector block are used to calculate the next iteration variable using this current iteration. In the next cycle, immediately the FSM will switch to the S2:RUN state if the stop signal is active low, where the next iteration calculated in the last cycle becomes the current iteration—by which next iteration is calculated with the help of stride determinator and stride selector blocks. The control stays in this state as long as the *reinitialize* signal is low or the stop signal is active low, i.e., in case of any buffer events. If the final iteration point is reached, reinitialize signal will be active high (as discussed in Appendix E.1.5) and if the iteration space scanner is programmed in the restart mode as mentioned before, control goes from S2:RUN state to S1:INITIAL state which makes the scanning of the iteration space to begin again from the initial iteration. Once the final iteration point is reached and if the restart mode is false, from S2:RUN state control goes to S3:FINAL state. In this state, one can restart the scanning of the tile iteration space again by issuing the external restart (*restart_ext*) signal which makes the state transition from S3:FINAL state to S1:INITIAL state.

It is to be noted that only in the configure state start signal is respected and in the rest of all the states only stop signal is respected which halts the computation of the next iteration in case of any buffer events. Furthermore, control signals, which depend on the current iteration are calculated in the CSG discussed in Section 5.3, are given as inputs to this control FSM block and they are also emitted as an output vector along with the current iteration as shown in Figure E.10. For instance, in the S0:INITIAL state initial values of the control signals from the initializer block are sent out as the control signals whereas in the all other states control signals (ic_in) calculated in the CSG block are sent as output signals as shown. To summarize, this FSM is one of the core blocks of the global controller design which is driving all other combinatorial blocks with the necessary inputs and getting the results back and halting the computation if needed.
E.2 Control Signal Generator

In the following, design of the control signal generator is explained in detail. The conceptual architecture block diagram of the control signal generator is shown in Figure E.11. This block contains a lot of registers to program, which hold the programming data for the Program Block Selector (PBS) as well as IC Signal Generator (ISG). These registers are programmed by the configuration bus and configuration clock interface similar to all the other blocks as mentioned before. Once the configuration is done, a configuration acknowledgment signal is also raised to notify the configuration logic. The maximum number of bits required to configure registers in this block, denoted by $N_{\text{bits}}^{\text{CSG}}$, are given by Equation (E.11). Similarly, the required number of configuration clock cycles to configure this block are calculated using Equation (E.12).

$$N_{\text{bits}}^{\text{CSG}} = \max_{\text{NO\_OF\_PROGRAM\_BLOCKS}}\left(\text{DIMENSION} \times (2 \times \text{ITERATION\_VARIABLE\_WIDTH} + \text{DIMENSION} + 1) + \text{NO\_OF\_IC\_SIGNS} \times (2 \times \text{ITERATION\_VARIABLE\_WIDTH} + \text{DIMENSION} + 1)\right)$$  \hspace{1cm} (E.11)

$$N_{\text{cycle}}^{\text{CSG}} = \max_{\text{NO\_OF\_PROGRAM\_BLOCKS}}\left(\text{DIMENSION} \times 4 + \text{NO\_OF\_IC\_SIGNS} \times 4\right)$$  \hspace{1cm} (E.12)

The PBS logic, shown in the block diagram, determines the current iteration's program block. Here, depending on the number of program blocks these PBS logic blocks can be increased. However, for the sake of better visualization in the block diagram, only three such blocks are shown. For each PBS logic, depending on the number of iteration variables, i.e., on the tile dimensions, comparators to check the min/max bounds on the iteration variables are instantiated. For each such comparator, min and max bounds of the iteration variables are provided by programmed registers and appropriate iteration variable is selected by iteration variable selector register. Outputs of the comparators in a PBS logic are ANDed together—a true value indicates that the current iteration belongs to that program block. It is to be noted that even though at synthesis time the number of dimensions and the number of program blocks are specified by respective parameters, at runtime, desired number of dimensions and program block are selected using output selector (output_selector) and program block selector (pb_selector) signals, respectively. Once the program block is identified, one needs to generate the control signals in that program block based on the subindexspaces of the transfer condition spaces as mentioned in Section 5.3.
Figure E.11: Architectural block diagram of control signal generator.
E.3 Configuration Scheme

Depending on the number of control signals, such logic blocks can be instantiated multiple times for each program block where each of them generates a control signal. However, to visualize better in the block diagram, for each program block only one ISG block is shown. The comparators in these logic blocks are also driven by min/max bounds of iteration variables (corresponding to the subindexspaces of transfer condition spaces determining the control signal values in the Reduced Order Binary Decision Diagram (ROBDD) as discussed in Section 5.3) provided by programmed registers and an appropriate iteration variable is selected for each comparator by an iteration variable selector register. At runtime, number of comparators in each ISG logic block are selected based on the output selector signal even though at synthesis time higher number of comparators selected by parameter DIMENSION. Similarly, for each program block, the number of ISG logic blocks, which drive a distinct tristated control signal, is selected by ic selector signal (ic_selector) at runtime even though at synthesis time a higher number of ic signal generator blocks is specified with a parameter NO_OF_IC_SIGNALS. In this way, a fully programmable control signal generator hardware block is designed to generate the control signals.

E.3 Configuration Scheme

In this section, we explain the configuration scheme for the entire global controller block as well as a corresponding hardware scheme. All the discussed hardware blocks have the configuration bus and configuration clock interface which are used in programming various registers in these blocks. All these hardware blocks are programmed one after the other using a small FSM called loader FSM as shown in Figure E.12, whose states are synchronous with the configuration clock. Once the control FSM of the global controller is reset, configuration enable is activated indicating that global controller is going to be configured which makes the loader FSM to come out of the reset state. Later on, in each state of the loader FSM, a hardware block is configured by selecting that block with an appropriate select signal and programming the registers in that block with the data coming from the configuration bus. In each hardware block, depending on the output selector signal/ic selector signal/program block selector signals at appropriate time a configuration acknowledgement is raised after configuring only the required number of registers. The configuration acknowledgement from a hardware block causes the loader FSM to transition from one state to the other, i.e., configuring one block after the other. Different hardware blocks are programmed in the following order: clock divider, initializer, stride selector, stride determinator, control signal generator, and reinitializer. Once the configuration of all the blocks is over, configuration done (config_done) signal is raised notifying the control FSM logic so that once the DCM is also
Figure E.12: State diagram of loader FSM that issues the select signals for configuring different hardware blocks in the global controller.

locked and start signal is available, it will start calculating the next iteration and control signals. Now, the important question that needs to be addressed is *From where and how the configuration data comes onto the configuration bus?* The answer to this question is explained by the configuration hardware scheme shown in Figure E.13. Assume that the required global controller configuration data is programmed through a LEON RISC core into the global controller configuration memory attached to the APB bus. In this memory, second bit of the first word is reserved for the global controller reset and the first bit is for programming done signal (pdone), which indicates that the configuration memory is populated with the configuration data. From the second word onwards, actual configuration data for the different hardware blocks is present. While populating the memory through the LEON core, the global controller is always forced into reset state and the programming done signal is set to active low. Once the full configuration memory is populated—only the first word is re-written—this time the reset of the global controller is released and programming done signal is forced to high. Release of the reset will result in the configuration enable
signal from the control FSM and the loader FSM is ready to read data from the configuration memory once the configuration enable signal is high.

In order to read word by word from configuration memory a small four state MEM_READ FSM is attached with the configuration memory. States of this FSM are controlled by the ahb_clk itself and this FSM reset corresponds to LEON3 reset, which is a active low signal. Once the configuration is enabled and the configuration memory is populated, MEM_READ FSM enters into a READ state where it will read word by word from the configuration memory and puts it onto the configuration bus. Then, the loader FSM will configure the hardware blocks in the global controller one after the other as mentioned above. However, whenever there is a transition between the states of the loader FSM, one cycle is needed for the next block to accept data coming from the configuration bus. Therefore, a read not ready signal is generated and given as input to MEM_READ FSM which introduces a single cycle wait state while reading from configuration memory. In other words, MEM_READ FSM switches from READ state to WAIT state for one cycle and switches back to the READ state. Once all the hardware blocks are programmed, i.e., in the CONFIG_DONE state of loader FSM, a configuration done signal is raised which causes the MEM_READ FSM to stop further reading from the global controller configuration memory and it switches from READ state to READ_DONE state. MEM_READ FSM stays in the READ_DONE state as long as global controller is not reset. It is worth noticing that the global controller reset is decoupled from the RISC core reset—enabling the global controller to be reset independently and to re-program it—a recommended methodology to follow [Cha08]. Furthermore, the configuration clock which is the same as ahb_clk and the global controller design clock are two completely independent clocks—enabling the configuration logic to run much faster, which also reduces the overall load on the configuration clock. Depending on the design complexity, global controller can run at a different frequency compared to the LEON core since the clocks are decoupled from each other.

### E.4 Configuration Data and Time

The total amount of configuration data required by the global controller is nothing but the sum of all the configuration data required by its hardware sub blocks. It is denoted by $N_{GC \text{ bits}}$ and given by the following equation:

$$N_{GC \text{ bits}} = N_{CD \text{ bits}} + N_{IN \text{ bits}} + N_{SS \text{ bits}} + N_{SD \text{ bits}} + N_{CSG \text{ bits}} + N_{RI \text{ bits}} \quad (E.13)$$

In Figure E.14, the amount of configuration data required is shown with respect to varying number of tile dimensions and program blocks for a single control signal.
**Figure E.13:** Hardware for the configuration scheme used in the configuration of the global controller showing how to program the configuration memory through a LEON core and read from it word by word using a MEM_READ FSM.
E.4 Configuration Data and Time

The configuration data increases with increasing number of tile dimensions, program blocks, and control signals as the number of registers to configure also increases with any of the above mentioned parameters.

Similarly, the global controller configuration time is measured in clock cycles and is given by the sum of all the configuration cycles required by its hardware sub blocks. It is denoted by $N_{GC}$ cycles and given by the following equation:

$$N_{GC}^{cycles} = N_{CD}^{cycle} + N_{IN}^{cycle} + N_{SS}^{cycle} + N_{SD}^{cycle} + N_{CSG}^{cycle} + N_{RI}^{cycle} + 5$$  \hspace{0.5cm} (E.14)

In Equation (E.14), the extra offset of 5 cycles accounts for the unused configuration cycles when configuring different hardware sub blocks one after the other in the loader FSM as mentioned in Appendix E.3.

Figure E.15 shows the configuration cycles required with respect to varying number of tile dimensions and program blocks for a single control signal. The configuration time also increases with increasing number of tile dimensions, program blocks, and control signals as the number of registers to configure also increases with any of the above mentioned parameters.

Figure E.14: Required configuration data of the global controller with respect to varying number of tile dimensions, program blocks for one control signal.
Figure E.15: Required configuration time of the global controller with respect to varying number of tile dimensions, program blocks for one control signal.
German Part

Codegenerierung für eng gekoppelte Prozessorfelder
Zusammenfassung

Bibliography


[DSRV02] Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frédéric Vivien. Constructing and Exploiting Linear Schedules


International Workshop on Reconfigurable Communication Centric System-on-Chips (ReCoSoC), pages 31–37, July 2006.


[KOD+91] Kazukuni Kitagaki, Takayoshi Oto, Tatsuhiko Demura, Yoshitsugu Araki, and Tomoji Takada. New Address Generation Unit


285
Bibliography


[MBF+12] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis...


Bibliography


Bibliography


Bibliography


Author’s Own Publications


Author’s Own Publications


<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Address Condition</td>
</tr>
<tr>
<td>ADL</td>
<td>Architectural Description Language</td>
</tr>
<tr>
<td>AE</td>
<td>Address Equation</td>
</tr>
<tr>
<td>AG</td>
<td>Address Generator</td>
</tr>
<tr>
<td>AHB</td>
<td>Advanced High-performance Bus</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>APU</td>
<td>Accelerated Processing Unit</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>BDD</td>
<td>Binary Decision Diagram</td>
</tr>
<tr>
<td>BRAM</td>
<td>Block Random Access Memory</td>
</tr>
<tr>
<td>CCP</td>
<td>Configuration and Communication Processor</td>
</tr>
<tr>
<td>CGRA</td>
<td>Coarse-Grained Reconfigurable Array</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSG</td>
<td>Control Signal Generator</td>
</tr>
<tr>
<td>CSR</td>
<td>Configurable Shift Register</td>
</tr>
<tr>
<td>DCM</td>
<td>Digital Clock Manager</td>
</tr>
<tr>
<td>DLP</td>
<td>Data Level Parallelism</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DPLA</td>
<td>Dynamic Piecewise Linear Algorithm</td>
</tr>
<tr>
<td>DRP</td>
<td>Dynamic Reconfiguration Port</td>
</tr>
<tr>
<td>DSM</td>
<td>Deep Sub Micron</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage and Frequency Scaling</td>
</tr>
<tr>
<td>FOC</td>
<td>Final Optimized Code</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>FU</td>
<td>Functional Unit</td>
</tr>
<tr>
<td>GC</td>
<td>Global Controller</td>
</tr>
<tr>
<td>GL</td>
<td>Global Latency</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Computation On Graphics Processing Unit</td>
</tr>
</tbody>
</table>
**Acronyms**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HLS</td>
<td>High-Level Synthesis</td>
</tr>
<tr>
<td>II</td>
<td>Iteration Interval</td>
</tr>
<tr>
<td>ILP</td>
<td>Instruction-Level Parallelism</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>ISG</td>
<td>IC Signal Generator</td>
</tr>
<tr>
<td>ISS</td>
<td>Iteration Space Scanner</td>
</tr>
<tr>
<td>LL</td>
<td>Local Latency</td>
</tr>
<tr>
<td>LLP</td>
<td>Loop-Level Parallelism</td>
</tr>
<tr>
<td>LPGS</td>
<td>Locally Parallel Globally Sequential</td>
</tr>
<tr>
<td>LSGP</td>
<td>Locally Sequential Globally Parallel</td>
</tr>
<tr>
<td>LT</td>
<td>Life Time</td>
</tr>
<tr>
<td>LTI</td>
<td>Life Time Interval</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiplier Accumulator</td>
</tr>
<tr>
<td>MILP</td>
<td>Mixed Integer Linear Programming</td>
</tr>
<tr>
<td>MOSFET</td>
<td>Metal-Oxide-Semiconductor Field-Effect Transistor</td>
</tr>
<tr>
<td>MPPA</td>
<td>Massively Parallel Processor Array</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multi-Processor System-on-Chip</td>
</tr>
<tr>
<td>MTMDD</td>
<td>Multi-Terminal Multi-Valued Decision Diagram</td>
</tr>
<tr>
<td>NoC</td>
<td>Network on Chip</td>
</tr>
<tr>
<td>PBS</td>
<td>Program Block Selector</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PLA</td>
<td>Piecewise Linear Algorithm</td>
</tr>
<tr>
<td>PRA</td>
<td>Piecewise Regular Algorithm</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RB</td>
<td>Reconfigurable Buffer</td>
</tr>
<tr>
<td>RDG</td>
<td>Reduced Dependence Graph</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>ROBDD</td>
<td>Reduced Order Binary Decision Diagram</td>
</tr>
<tr>
<td>RTL</td>
<td>Register-Transfer Level</td>
</tr>
<tr>
<td>SII</td>
<td>Super Iteration Interval</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SSA</td>
<td>Single Static Assignment</td>
</tr>
<tr>
<td>TCPA</td>
<td>Tightly Coupled Processor Array</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread Level Parallelism</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
<tr>
<td>ZOLB</td>
<td>Zero Overhead Loop Buffer</td>
</tr>
<tr>
<td>ZOLC</td>
<td>Zero Overhead Loop Controller</td>
</tr>
</tbody>
</table>