Mocarabe: High-Performance Time-Multiplexed Overlays for FPGAs

by

Alireza Mellat

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This work is an extension of our work published in [43]. My colleagues Frederick Tombs, Ian Lang, and Srinirdheeshwar Kuttuva Prakash contributed to this work. Fred laid the foundation for our compiler and wrote the entire compiler flow for the torus (allocation, partitioning, placement, and scheduling), Ian proposed the ILP placer, and Srinirdheeshwar helped me through my research by giving me insights during our conversations. I am grateful for all their help.

Abstract

Coarse-grained reconfigurable array (CGRA) overlays can improve dataflow kernel throughput by an order of magnitude over Vivado HLS on Xilinx Alveo U280. This is possible with a combination of carefully floorplanned high-frequency (645 - 768 MHz Torus, 788 - 856 MHz Mesh, 583 - 746 MHz BFT) design and a scalable, communication-aware compiler. Our CGRA architecture supports configurable Processing Element (PE) functionality supported by a configurable number of communication channels to match application demands. Compared to recent FPGA overlays like 4×4 ADRES and HyCUBE implementations in CGRA-ME, our design operates at a faster clock frequency by up to 3.4×, while scaling to an orders-of-magnitude larger array size of 19×69 on Xilinx Alveo U280.

We propose a novel topology agnostic ILP placer that formulates the CGRA placement problem into an ILP problem. Our ILP placer optimizes placement regardless of topology and even for non-linear objective functions by using pre-computed placement costs as inputs to the ILP problem formulation. Using the ILP placer reduces placement quadratic wirelength up to 37% compared to the commonly used simulated annealing approach but increases runtime from less than a minute to hours.

Our communication-aware compiler targets HLS objectives such as initiation interval (II) and minimizes communication cost using an integer linear programming (ILP) formulation. Unlike SDC schedulers in FPGA HLS tools, we treat data movement as a first-class citizen by encoding the space and time resources of the communication network in the ILP formulation. Given the same constraints on operational resources as Vivado HLS, we can retain our target II and achieve up to 9.2× higher frequency. We compare Torus and Mesh topologies, and show Mesh has less latency per area compared to Torus for the same benchmarks.

Acknowledgements

I would like to thank my supervisor, Professor Nachiket Kapre, for all his support and help which enabled me to go through my Master's journey.

Next, I would like to thank everyone in the WatCAG lab: my friends and lab mates Frederick Tombs, Srinirdheeshwar Kuttuva Prakash, and Ian Lang, whose helps played an important role in this work.

The last couple of years have been difficult to pass, being far away from home with lockdowns during the pandemic. My dear friends Simin Asgari, Srinirdheeshwar Kuttuva Prakash, Sajjad Rashidiani, and Abtin Riasatian were the propelling force that helped me pass through these difficult times. I will always be grateful for their friendship.

Finally, I would like to thank my lovely family for their unconditional support and love. I would like to thank my father, who has always been supportive and has helped me with my difficulties, and my mother who has always been there for me, even through difficult times and when I was not at my best. My lovely brother has always been the source of joy in my life and has always been by my side. I would not have reached where I am now without their help and support, and will always be grateful for all they have done for me.

Dedication

This is dedicated to my grandfather, who was the guiding light through my life, and I will dearly miss him forever.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The slowdown in Moore's Law is limiting improvements to CPU performance scaling [42]. Specialized accelerators can increase throughput and efficiency of computations across various application domains. Field-Programmable Gate Arrays (FPGAs) are being adopted by cloud service providers such as Microsoft, IBM, and Intel [6] [13] [34], but low-level register-transfer level (RTL) development for these platforms is difficult and expensive. One solution is the use of High-Level Synthesis (HLS) to express the desired algorithmic behavior in high-level C/C++ rather than low-level RTL, but this approach can suffer from inefficiencies and overheads resulting in low frequency at scale (up to 9× away from peak, demonstrated in Section 5.6). We still need to complete FPGA place and route that can take hours or days of run-time for large cloud-scale designs. Furthermore, modern FPGAs like Xilinx Alveo U280 are composed of multiple FPGA dies connected together to form a large FPGA. Although the connected dies are equal to a large uniform FPGA in terms of available resources, they have limited connectivity to each other and can impose routing restrictions on the data crossing between them. Even though one may be able to write optimized RTL and HLS codes capable of efficiently using modern FPGAs, the code may not perform as expected as the implementation may result in die-crossing induced delays.

We can use structured coarse-grained FPGA overlays with careful floorplanning that trade-off FPGA flexibility in exchange for performance guarantees and faster application mapping times [37]. Coarse-grained reconfigurable arrays (CGRAs) are a class of programmable logic devices consisting of several coarse logic blocks such as adders and multipliers connected together with reconfigurable interconnect to provide high performance for specific classes of applications. Each logic block can have multiple inputs and outputs, a configurable ALU, and provide support for storage and data movement. In contrast to FPGAs that contain finer-grained

logic blocks (e.g., look-up-tables) supported by bit-level interconnect, CGRAs are less flexible and provide ALU-like building blocks with bus-oriented interconnect. One can naively overlay existing CGRAs on top of modern FPGAs [13] [34], but these overlays do not fully leverage the benefits of operating on an FPGA. They have small fixed array sizes (e.g. 4×4 [16] [46][39]), and operate at low frequencies (226–382 MHz [16][39]).

1.1 Objective

The aim of this thesis is to provide a framework that compiles structured high-level C code into high-speed FPGA overlays with efficient usage of FPGA resources. To that end, we introduce Mocarabe, a programmable FPGA overlay CGRA that supports variable array sizes and topologies (up to 19×69 Torus, 19×22 Mesh, and 256 PE BFT) and operates at high frequencies (645 – 768 MHz Torus, 788 – 856 MHz Mesh, and 583 - 746 MHz BFT) on the Xilinx Alveo U280 chip. We perform CGRA placement using our novel topology-agnostic Integer Linear Programming (ILP) placer, and reduce placement quadratic wirelength up to 37% compared to traditional simulated annealing. Mocarabe offers rich interconnect flexibility in the form of multiple Network on Chip topologies, multiple communication channels, and logic block I/Os that are essential for supporting communication-rich dataflow kernels. Our space-time compiler is an interconnect-aware ILP scheduling formulation which routes data movement in space and time while minimizing communication costs. To efficiently floorplan our CGRA, we use handcrafted FPGA placement scripts that ensure each CGRA block receives the resources necessary for high-frequency operation. Using our framework, one can give an input C code and receive a high-speed overlay programmed to deploy the given code, providing both high-level programmability and high performance.

1.2 Main contributions

The key contributions of this work are:

- Design of a CGRA architecture with pipelined interconnect for high-frequency operation with support for multiple topologies and communication channels.
- An interconnect-rich CGRA providing various CGRA multi-channel interconnect topologies (Torus, Mesh, BFT).
- An ILP based placer to find the optimum placement on CGRA PEs.

- A CGRA compiler with a communication-aware ILP scheduler formulation backbone that encodes interconnect resources in the ILP and generates a time-multiplexed hardware schedule for a given code.
- Comparison of ILP and simulated annealing placer that shows up to 37% reduction in quadratic wirelength.
- A carefully floorplanned implementation of 19×69 Torus, 19×22 Mesh, and 256 PE BFT Mocarabe configurations with up to 3 Torus or 2 Mesh channels on the Xilinx Alveo U280 for 645 – 768 MHz Torus, 788 – 856 MHz Mesh, and 583 – 746 MHz BFT operation frequency.
- Evaluation of Mocarabe as an HLS tool by comparing it with Vivado HLS across a range of datapath kernels with 4–49 I/Os, 2–35 adds, and 0–45 multiplies.

1.3 Thesis organization

The remainder of this thesis is organized as follows:

- **Chapter 2** gives detailed background information about CGRAs, ILP, Simulated Annealing (SA), dataflow computing, Network on Chips, the Alveo U280 board, and Vivado HLS.
- **Chapter 3** describes the hardware architecture of Mocarabe by explaining Processing Elements and Network on Chips components. Additionally, it presents CGRA implementation and floorplanning on Alveo U280, and gives details about how we managed to achieve a scalable high-frequency design.
- **Chapter 4** describes the Mocarabe compiler flow. It gives a detailed explanation about different parts of our compiler by explaining the allocation, partitioning, placement, and scheduling phases. It describes our ILP formulation of CGRA placement and ILP formulation for scheduling.
- **Chapter 5** explains our experiments and presents their results. We quantify the cost and performance of Mocarabe and compare it to other CGRAs and Vivado HLS. We show how many communication channels our scheduler needs to run each benchmark on Torus, Mesh, and BFT configurations, and compare ILP and SA placement in terms on runtime and quality of results.

- **Chapter 6** concludes the thesis by giving a brief overview of Mocarabe and presents future research ideas that could benefit Mocarabe compiler, hardware, and RTL overlay placement.

Chapter 2

Background

Developing applications for FPGA execution is a different experience from software development. Hardware design suffers from a lower productivity compared to software development as FPGA tools and programming methods have a steep learning curve. FPGA overlays are virtual reconfigurable architectures that are implemented on top of the FPGA fabric and act as an intermediate layer between FPGA and the application[37], as shown in Figure 2.1.

This chapter gives an overview of CGRAs and reviews notable recent CGRA frameworks. Then, we review Integer Linear Programming (ILP), Simulated Annealing (SA), dataflow computing, Network on Chips (NoCs), and the Alveo U280 board as they each play a vital role in this work.

2.1 CGRAs

Coarse-grained reconfigurable architectures (CGRAs) are a class of programmable logic devices where the processing elements (PEs) are large ALU-like logic blocks, and the interconnect fabric is bus-based. Figure 2.2 shows an abstract CGRA. This stands in contrast to Field Programmable Gate Arrays (FPGAs), which are configurable at the individual LUT level. CGRAs dedicate less area to flexibility, therefore requiring far fewer configuration bits than FPGAs. This simplifies their CAD complexity by reducing the number of decisions the tools need to make. Despite their reduced flexibility, CGRAs are ideal for applications where: 1) some flexibility is required, 2) software programmability is desired, and 3) compute / communication needs closely match with the CGRA capabilities. CGRAs can be realized as custom ASICs, or alternatively, implemented on FPGAs as overlays. CGRAs provide word-wide datapaths and more complex operators compared to FPGAs Theye also provide word-wide interconnect thereby reducing routing costs.

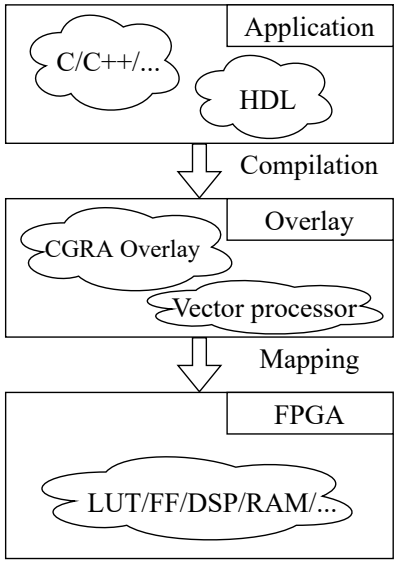

Figure 2.1. Overlay as a layer between FPGA and application.

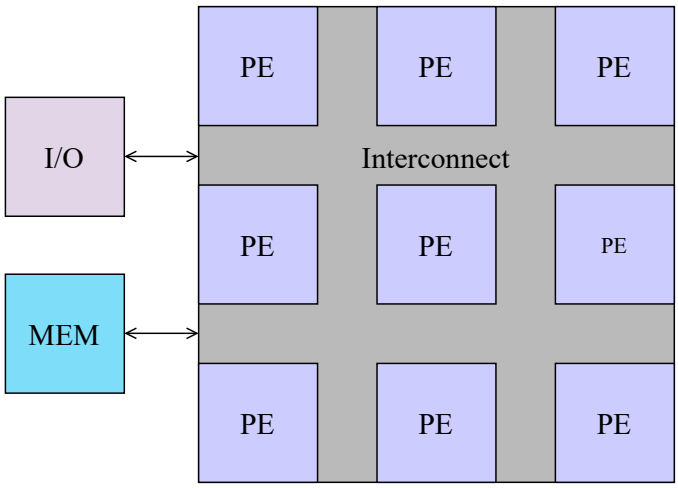

Figure 2.2. An abstract CGRA.

2.1.1 Differentiating factors for CGRAs

CGRAs can be classified into different groups based on their microarchitecures. Although many classifications are available, most CGRAs can be differentiated based on the following factors:

- **Logic functionality:** An important aspect of each CGRA is the types of computation in each block. Some CGRAs have blocks capable of more complex operations like branching, while others may only be capable of simple arithmetic operations per block. Mocarabe supports addition and multiplication per block.
- **Interconnect architecture:** Interconnect architecture and the data movement through the array influences cost and performance of the array. Some CGRAs use shared Network-On-Chips (NoCs) , while others use single hop or multi hop crossbar connections. Mocarabe supports NoCs with Torus, Mesh, and Butterfly Fat Tree (BFT) topologies.
- **Memory:** Memory can be contained inside the CGRA, be external, or a hybrid of both. Mocarabe has shift registers that store the incoming data at every block.
- **Dynamic reconfigurability:** Some CGRAs enable the user to reconfigure functional units and interconnects based on their application (closer to FPGAs). On the other hand, functional units and interconnect can be fixed and just perform differently for each application

(closer to ASICs). In Mocarabe, each block can be configured as either an adder or a multiplier, and interconnect can be Mesh, Torus, or BFT.

2.1.2 CGRAs vs ASICs and FPGAs

CGRAs provide coarse grained reconfigurability while maintaining high performance. Compared to ASICs that are fixed designs tailored towards specific applications with high performance, CGRAs provide some programmability, making them flexible for a wider range of applications. However, CGRAs have lower performance compared to ASICs due to the added overhead for programmability. Compared to FPGAs, CGRAs have shorter compile times as fewer, bigger components like PEs and switches need to be configured, in contrast to FPGAs that configure devices at the LUT and FF level. As a result, CGRAs can be placed between ASICs and FPGAs in terms of performance and flexibility, as shown in Figure 2.3, and are suitable for cases where high performance is needed with a degree of reconfigurability.

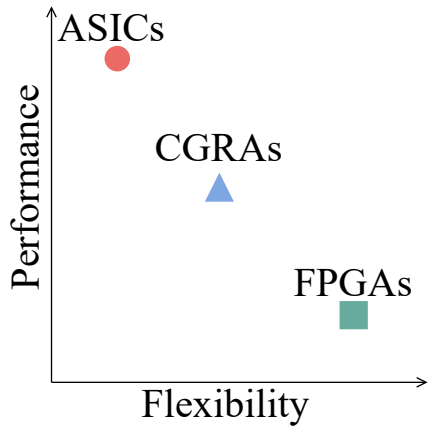

Figure 2.3. ASICs vs CGRAs vs FPGAs

2.2 Survey of CGRA frameworks

In this section we survey notable CGRA frameworks, and compare our work with some of the mentioned CGRAs in the later chapters.

2.2.1 ADRES

ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) [24] is a CGRA proposed in 2003 which couples a coarse grained reconfigurable matrix with a VLIW processor into a single architecture. The ADRES core has multiple functional units (FU) and Register Files (RF), with the FUs executing word level operations and RFs storing immediate data. The architecture has two views: a VLIW processor and a reconfigurable matrix. The VLIW has faster, more capable FUs connected to each other through a multi-port RF. The FUs have load/store operations available to access data on the RF.

The reconfigurable matrix is composed of the FUs and RFs shared with the VLIW and Reconfigurable cells (RC), each including FUs and RFs. Reconfigurable cell FUs are simpler and have less ports. They have a configuration RAM witch stores configurations locally to be used on a cycle by cycle basis. Figure 2.4 shows an ADRES core. The processor/coprocessor model means only one of the VLIW or reconfigurable matrix views are active at a time, enabling resource sharing between the views as they are different views of the same device. Additionally, the VLIW processor has higher performance than RISC cores, resulting in better speedups for the entire system. ADRES achieves an Instructions Per Cycle (IPC) of 28.7 for IDCT and 23.3 for FFT applications.

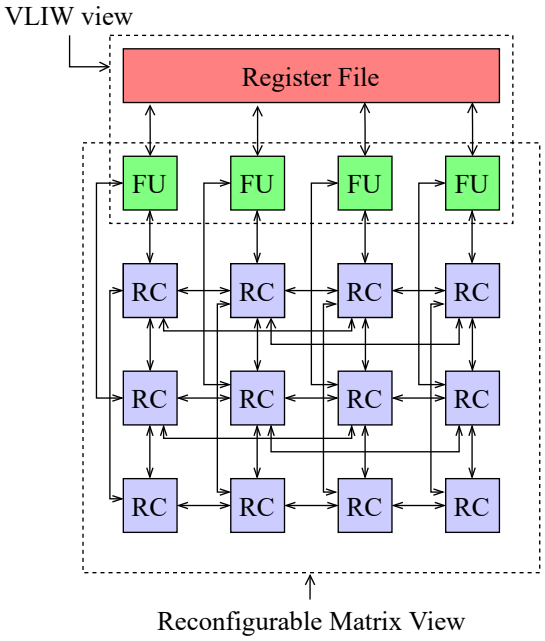

Figure 2.4. A 4×4 ADRES array.

2.2.2 ULP-SRP

Ultra Low Power Samsung Reconfigurable Processor (ULP-SRP) [16] is a CGRA proposed in 2014 focusing on biomedical applications. The SRP is a variation of ADRES that aims to satisfy both low energy consumption and high performance by having many FUs working on parallel to accelerate execution speed and lower execution time. Although having many FUs working in parallel increases power consumption, the reduced execution time has a bigger effect and leads

to a reduced total energy consumption. The SRP is composed of a 3×3 CGRA, used for accelerating loops with instruction level parallelism, and a dual issue VLIW which can process more complex functions and control flows outside the CGRA's capabilities. The CGRA supports two modes: a 2×2 low performance (LP), and a 3×3 high performance (HP) mode. Figure 2.5 shows the SRP structure.

In CGRA mode, the configuration is read from the Configuration Memory (CMEM) at every cycle. Furthermore, the CGRA can change performance mode at runtime using Dynamic CGRA Mode Changing (DCMC) which allows executing some loops in low performance mode while others execute at high performance mode. However, each application needs to be compiled once for each CGRA mode to enable using DCMC. To further increase energy efficiency, the SRP has multiple power domains and uses fine grained power gating to turn off FUs not used in a selected mode. ULP-SRP is implemented in 40nm ASIC and runs at 100MHz. It achieves an IPC of 2.35 in LP and 4.67 in HP for 256 point FFT applications.

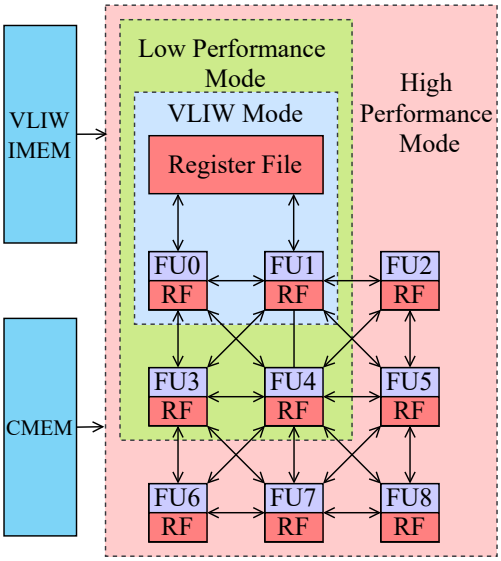

Figure 2.5. ULP-SRP structure.

2.2.3 SYSCORE

SYSCORE [29] is another CGRA similar to ADRES proposed in 2011. It focuses on low power, real-time processing of biomedical signal processing, and to the authors' claim, is the first CGRA aimed towards biosignal processing applications. SYSCORE architecture maps irregular algorithms by using RoundAbout Interconnect (RAI), which performs nearest neighbor data transfer without the power and area cost of dense interconnects. Furthermore, SYSCORE reduces energy usage by:

- Using reconfiguration to eliminate the fetch-decode steps of processors
- Using systolic data reuse to reduce the number of intermediate RAM accesses

- Using compact functional units to reduce logic switching
- Dynamically scaling Voltage and frequency

A 4×8 SYSCORE architecture is shown in Figure 2.6. The architecture has two main elements: Configurable Function Units (CFUs) and RAI. Direct Memory Access (DMA) units inject data into the architecture from west and north, and collect data from east. Each CFU has 4 input ports and 3 output ports and contains a computation unit tailored towards systolic algorithms capable of MUL-ADD, MULL-SUB, and compare (CMP). Each CFUs is connected to its nearest neighbor to the east and west. To reduce interconnect density, only Odd numbered columns have cross interconnections which are useful for non-systolic applications. A column of RAI is placed after every second columns of CFUs to allow data to go from any CFU on the left of the RAI column to any CFU on the right of the RAI column. The lack of a global interconnect further contributes to reducing chip area and power consumption. An 8×8 SYSCORE, built of two 4×8 arrays, is implemented using 90nm CMOS and operates at 100 MHz. Compared to DSP and Single Instruction Multiple (SIMD) designs, SYSCORE consumes less energy while providing up to $64\times$ speedup from DSP and $16\times$ speedup from SIMD.

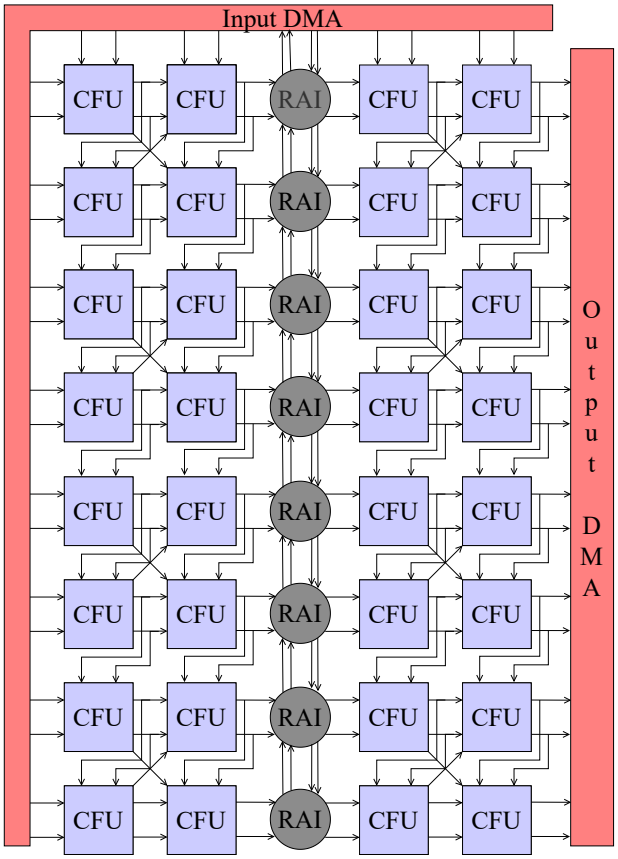

Figure 2.6. A 4×8 SYSCORE block.

2.2.4 Plasticine

Plasticine [33] is a large, 16×8 CGRA architecture focusing on parallel patterns proposed at 2017. At the highest abstraction level, it is a mesh of compute and memory units as shown in

Figure 2.7. Both the compute and memory units are programmable for every cycle of operation. Compute units contain ALUs and programmable state registers for controlling them, and support both Single Instruction Single Data (SISD) and SIMD Data type parallelism in addition to vector operations. The memory units contain scratchpad SRAMs and use a small set of ALUs cimbined with programmable logic to interface them. The mesh uses a set of address generators and coalescing units to interface outside memory. Plasticine is programmable using a custom language for dataflow computing computing called Spatial [17].

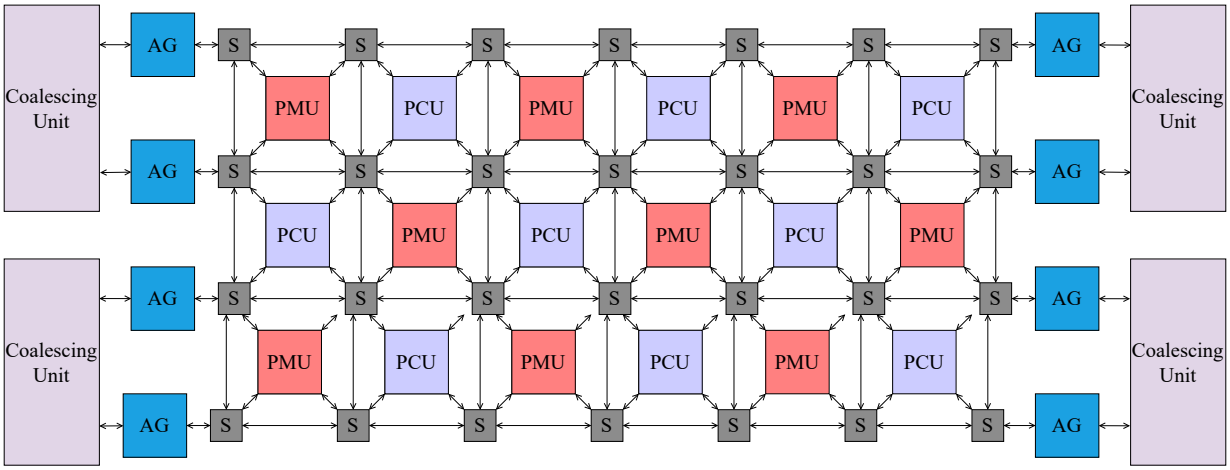

Figure 2.7. Plasticine array.

Plasticine is implemented as a 16×8 array using 28nm technology and runs at 1GHz. Compared to 28nm Altera Stratix V FPGA, Plasticine has a better performance to power ratio up to $76.9\times$.

2.2.5 Cascade

Cascade [46], proposed in 2019, aims at high-throughput data streaming by decoupling memory accesses from computations. Conventional designs generate data load/store memory addresses inside the array. However, generating addresses inside the arrays could use a noticeable amount of PEs which could otherwise perform computations. By moving the address generation outside the CGRA, Cascade reduces address generation overhead and makes the array focus on computations. Cascade uses specialized hardware units called Steam Engines (SE) that generate an address per cycle. Figure 2.8 shows Cascade architecture. Cascade compiler extracts computations and memory access patterns for a given C code and generates CGRA and SE con-

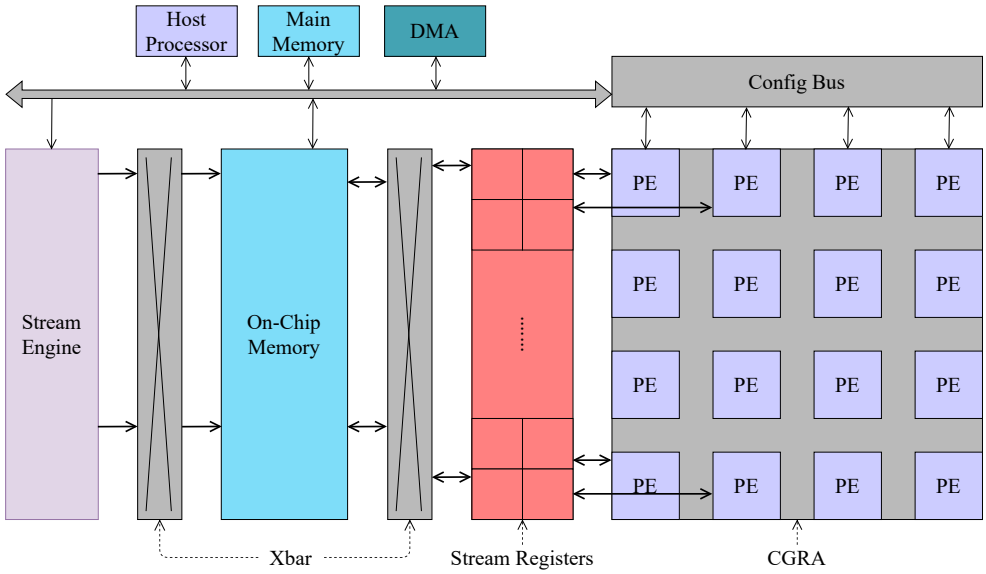

Figure 2.8. Cascade architecture.

figurations. Cascade is implemented as a 4×4 array in 40nm CMOS and reaches a maximum operating frequency of 510 MHz while resulting in up to $3\times$ improved performance.

2.2.6 4D-CGRA

4D-CGRA [15] proposed in 2019 introduces branch dimension to application mapping on CGRAs. Although statically scheduled modern CGRAs provide a good support for dataflow computing, static scheduling limits CGRAs' ability to accelerate loops with branches and complex control flows. 4D-CGRA uses predication for control divergence and maps both paths from a conditional divergence to the CGRA, while only one executes. When mapping conditional paths, duplicate allocation on unused resources limits performance and increases schedule length due to the wasted resources. To overcome this issue, splits each basic block (a single entry, single exit code sequence) into multiple subsets called shards, with all instructions inside a shard getting mapped to the same PE. Each shard is associated with a tag based on the the result of the last conditional branch. In 4D-CGRA, multiple shards get mapped to the same PE and only one of them gets selected for execution based on the branch outcomes. This leads to more efficient resource usage and reduced schedule lengths. Figure 2.9 compares 4D-CGRA to a generic CGRA for a given application. 4D-CGRA is implemented in 40nm technology, runs at 714 MHz, and improves throughput by up to $2.5\times$.

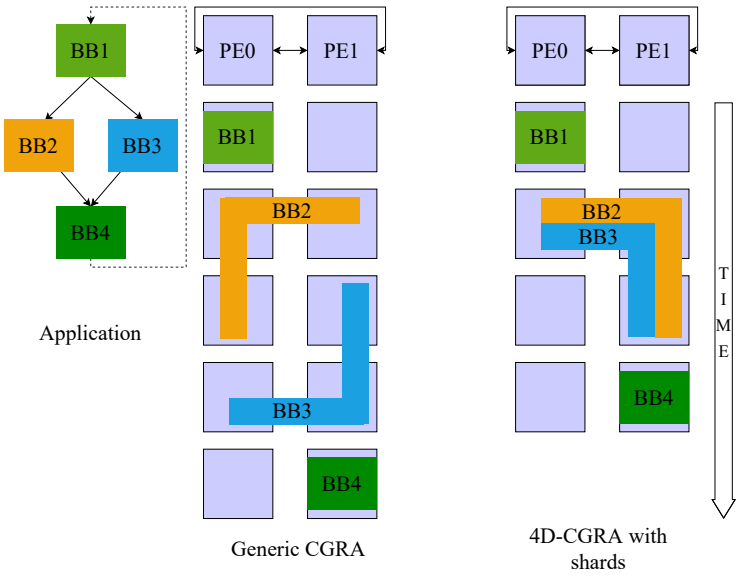

Figure 2.9. 4D-CGRA vs generic CGRA for a given application.

2.2.7 HyCube

HyCUBE [14] is a CGRA similar to ADRES with reconfigurable single-cycle multi-hop (combinational) interconnect proposed in 2017. HyCUBE's multi-hop interconnect benefits the CGRA by allowing for more flexible and efficient scheduling as distant PEs can exchange with minimum overheads, eliminating the need to use several intermediate PEs for communication. Unlike ADRES, HyCUBE loads data from the leftmost column of device, as opposed to ADRES which does so from the top row. Additionally, there are no toroid wrap around connections as data can go in any direction. Each PE consists of a FU and a crossbar switch. At each cycle. configuration is read from the configuration memory and the incming data can be registered or proceed directly to the crossbar. Figure 2.10 shows a 4×4 HyCUBE array and a PE's internal structure. HyCUBE is implemented as a 4×4 array on 28nm ASIC and operates at 704 MHz. HyCube achieves $1.5\times$ better performance-per-watt over a standard NoC.

2.2.8 CGRA-ME

CGRA Modelling and Exploration (CGRA-ME) [8] is an open-source framework initially proposed at 2017 that allows describing arbitrary CGRA architecture and enables mapping, placement, and scheduling C benchmarks to the arbitrary CGRA. It generates Verilog code for the resulting design for simulation and synthesis. CGRA-ME also allows for area and performance

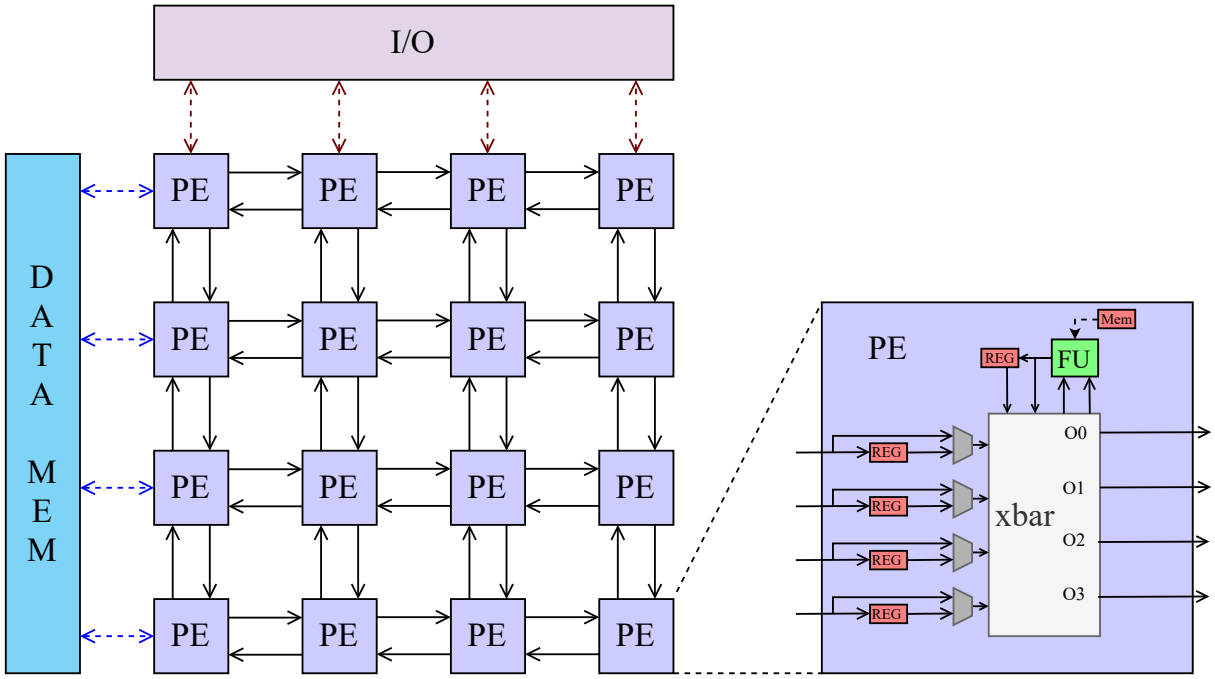

Figure 2.10. A 4×4 HyCUBE and enlarged view of PE.

modeling of CGRAs [26] by synthesizing commonly occurring primitives in isolation and adding

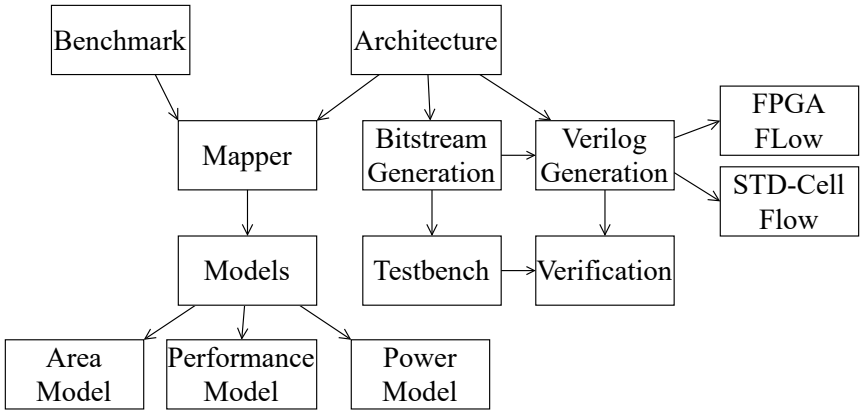

Figure 2.11. CGRA-ME framework overview.

component wise results together. Figure 2.11 [40] shows an overview of the CGRA-ME framework. ADRES and HyCUBE are implemented as FPGA overlays using CGRA-ME on Intel

Stratix 10 (S10) and Xilinx Ultrascale+ (US+) in [39]. After optimizations, the ADRES and HyCUBE overlays operate at up to 226 – 382MHz on S10 and US+.

2.2.9 Xilinx AI Engine

AI Engine [49] is a CGRA proposed by Xilinx in 2018 as a solution for higher compute density and lower power requirements of emerging machine learning and wireless applications. AI Engine provides $3-8\times$ compute capacity per silicon compared to programmable logic and reduces compute intensive power consumption by 50%. An AI Engine array can have between 30 AI Engines and 80k LUTs to 400 AI Engines and near 1 million LUTs, with each AI Engine including the following resources:

- Dedicated 16KB instruction memory
- 32KB RAM
- 32 bit RISC scalar processor
- 512 fixed-point and 512 bit floating-point vector processor

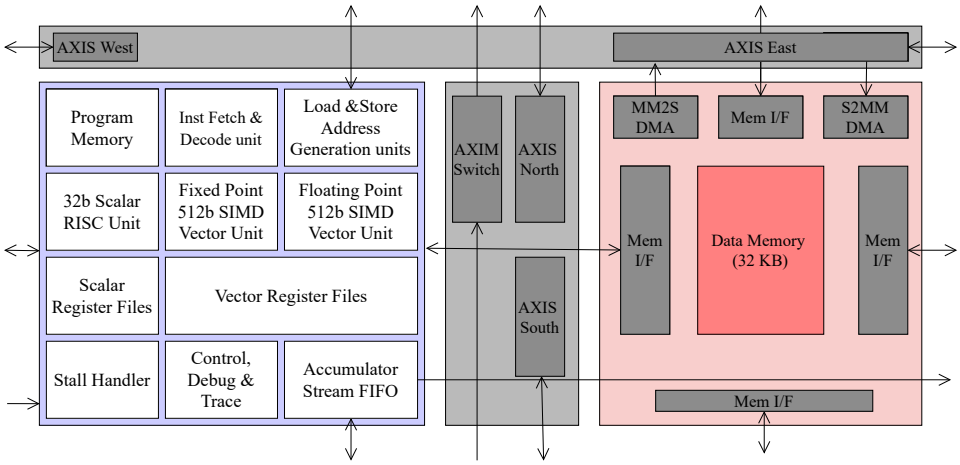

Figure 2.12. A Xilinx AI Engine tile.

Figure 2.12 shows details of the AI Engine tile. AI engine is implemented in 7nm and has a maximum frequency of 1GHz. It reaches 100 MHz for a 5-Channel LTE20 Wireless application.

2.2.10 Summary of Frameworks

To summarize the frameworks discussed so far, Table 2.1 shows the detail of each framework. As the discussed frameworks show, CGRAs are a capable candidate for accelerating applications and keep growing with new technologies, with more recent frameworks like Xilinx AI engine and Plasticine showcasing their potential. Furthermore, as frameworks like CASCADE and 4D-CGRA show, CGRAs can benefit greatly from compiler optimizations that make the most use of the available compute resources by reducing overheads and the number of IDLE resources.

TABLE 2.1
SUMMARY OF CGRA FRAMEWORKS

Name	Year	Tech	Size	Fmax	Performance
ADRES	2003	–	4×4	–	28.7 IPC (idct), 23.3 IPC (fft)
SYSCORE	2011	90 nm	8×8	100 MHz	Up to $64\times$ ($16\times$) speedup from DSP (SIMD)
ULP-SRP	2014	40nm	3×3	100 MHZ	2.35 IPC (LP), 4.67 IPC (HP) for 256 point FFT
HyCUBE	2017	28nm	4×4	704 MHZ	$1.5\times$ better performance-per-watt over standard NoC
Plasticine	2017	28nm	16×8	1GHz	Up to $76.9\times$ better performance to power ratio over Stratix V
AI Engine	2018	7nm	80 – 40	1GHz	100 MHz for a 5-Channel LTE20 Wireless application
CASCADE	2019	40nm	4×4	510 MHz	$3\times$ improved performance
4D-CGRA	2019	40nm	4×4	714MHZ	Up to $2.5\times$ increased throughput
CGRA-ME	2017	–	Variable	Variable	Variable

2.3 Integer Linear Programming (ILP)

Linear Programming (LP) is a method to compute optimal solutions for problems with linear objective functions and linear constraints. In Integer Linear Programming (ILP), the unknowns are limited to integers. ILP is NP-hard as the space of possible answers for variables is restricted to integers. Equation 2.1 shows an example ILP formulation which aims to minimize a value with respect to the given constraints. It can be seen as a scheduling problem with three tasks, called T_0, T_1, and T_2, with execution times of X_0, X_1, and X_2 respectively. T_1 relies on the output of T_0 and T_2 relies on the outputs of T_0 and T_1. As a result, assuming each task takes 1 timeslot to complete, their execution times can be formulated in the ILP formulation shown in Equation 2.1, which minimizes X_2 and forces a correct execution of tasks using constraints

($X_1 - X_0 \geq 1$ means X_1 should be executed at least 1 timeslot after X_0).

$$\begin{aligned} &\text{Variables: } X_0, X_1, X_2 \\ &\text{Constraints: } \begin{cases} X_0 > 0; X_1 > 0; X_2 > 0; \\ X_2 - X_0 \geq 1; \\ X_1 - X_0 \geq 1; \\ X_2 - X_1 \geq 1; \end{cases} \\ &\text{Objective: } Minimize X_2 \end{aligned} \tag{2.1}$$

2.3.1 Gurobi solver

Gurobi [12] is a commercial mathematical solver used for solving various optimization problems, like ILP. Gurobi provides a Python API and can be used to solve mathematical models in a Python program. In this work, we use Gurobi to solve ILP problems for partitioning, placement, and scheduling, which will be discussed in the upcoming chapters.

2.4 Simulated Annealing (SA)

Simulated annealing (SA) is a probabilistic technique for finding the global optimum of a given function. It is a metaheuristic particularly useful for finding global optimum in a large search space and is used when the search space is discrete. Although SA may not produce the optimal solution in finite time, it can find a solution close enough to the optimal solution faster than ILP given the right parameters. As shown in Equation 2.2, SA consists of multiple iterations of the annealing algorithm, in which every time a random variable is picked and given a new value at random. If the new value is an improvement with respect to a cost function, the new value is adopted for the variable. If the new value is not an improvement with respect to a cost function, it is adopted probabilistically based on a temperature parameter, T. With an initial value on N_0 and a new value of N_1, a cost function h(x) the new value will be accepted with a probability of $e^{-(h(N_1)-h(N_0))/T}$. Temperature can be reduced after each iteration. As a result, initial parameters

play an important role in SA's performance. SA is commonly used in CAD tools for placement.

$$
\begin{aligned}
&\text{Let } N_0 \text{ space be the initial assignment of values to all variables}\\
&\text{Let T be a high temperature}\\
&\text{repeat:}\begin{cases}
\text{Pick a random variable and assign it to a random value.}\\
\text{Call the new values of variables } N_1\\
\text{if } h(N_1) < h(N_0):\\
\quad N_0 = N_1\\
\text{else :}\\
\quad N_0 = N_1 \text{with probability} e^{-(h(N_1)-h(N_0))/T}\\
\text{reduce T}\\
\text{Stop if stopping criteria is reached}
\end{cases}
\end{aligned}
\tag{2.2}
$$

2.5 Dataflow computing

Dataflow computing models programs as graphs of data moving between operations. Unlike traditional programming where programs are modelled as series of operations executing in a specific sequential order, dataflow computing consists of programs each having multiple operations working in parallel, each processing as soon as their input operands are available.

2.5.1 Dataflow graph

A dataflow program can be described as a DataFlow Graph (DFG). Each node represents an operation and the edges between the nodes indicate data dependencies between operations. If an edge goes from node A to node B, it means node B relies on node A's output for one of its inputs, and as a result it should be executed after node A. Input nodes have no in going edges and output nodes have to no outgoing edges. Figure 2.13 shows a DFG for $z = ax^2 + bxy$, which has 4 input nodes and an output node. As $ax^2 + bxy$ can be written as $x(ax + by)$, we should first compute ax and by, and since they do not rely on each other, they can be computed in parallel. Then, we should add ax and by together and multiply the result by x to get $z = ax^2 + bxy$. The process to compute $z = ax^2 + bxy$ is encoded in the DFG in Figure 2.13, with the first two multiplications happening in parallel and using their results to make the rest of the computations. We can use ILP to formulate and schedule DFG nodes, giving each node an execution time in such a way that all data dependencies are maintained.

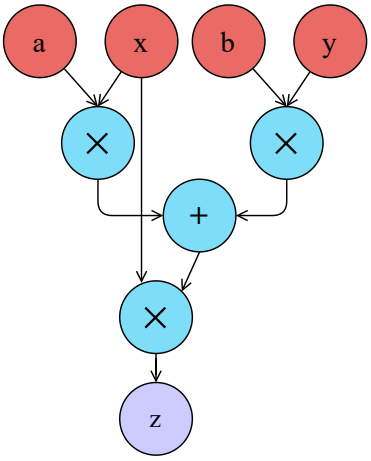

Figure 2.13. An example DFG for $z = ax^2 + bxy$.

2.6 Network on Chips

FPGAs have long supported statically configured routing which gets programmed at compile time and does not change during runtime. Although the statically configured routing suits circuit-like applications well, there is still a need for runtime-configured communications to route the traffic overlays or IP cores generate. Network on Chips (NoCs) provide chip-wide communication and support routing multiple individual packets simultaneously using NoC switches, making them a suitable candidate for managing chip-wide exchange of data and control. NoCs could have multiple topologies and configurations like Mesh [20] [25], Torus [10], and Butterfly fat trees [1] [27]. NoCs are usually either time-multiplexed or packet-switched. In packet-switched NoCs, data arrives at switches in packets containing their destination address, and routing decisions get made at packet arrival based on each packet's destination. In time-multiplexed routing, routing decisions at each switch are based on a fixed, predetermined schedule that instructs each switch on how to route the incoming data at each cycle. In Mocarabe, we use statically-scheduled time-multiplexed NoCs and store routing decisions in RAM blocks called context memories.

2.6.1 Mesh

In Mesh topology, switches are connected in a 2D array structure as shown in Figure 2.14. Data moves bidirectionally and each switch is connected to its 4 neighboring switches at north, south, east, and west, with boundary switches having less than 4 connections.

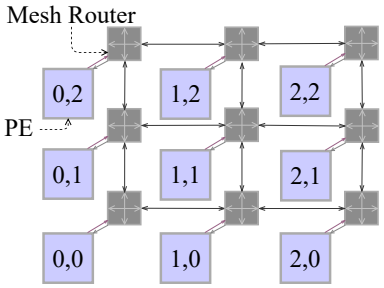

Figure 2.14. A 3 × 3 Mesh.

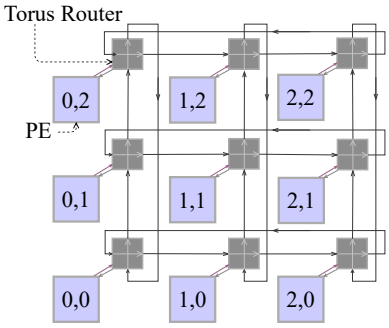

Figure 2.15. A 3 × 3 Torus.

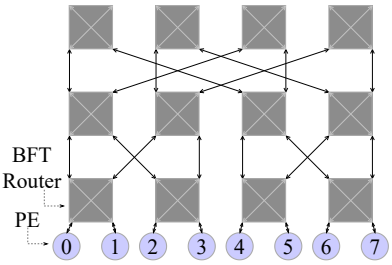

Figure 2.16. BFT topology.

2.6.2 Torus

In Torus topology, switches are connected in a 2D array structure as shown in Figure 2.15. Each switch is connected to its 4 neighboring switches at the top, bottom, left, and right, with boundary switches having wrap-around connections that go through the entire array and connect them to other boundary switches and the other end of the array. Data moves directionally to east and north. Although data only moves in one direction, having the wrap-around connections enables data to reach any destination on the array.

2.6.3 Butterfly Fat Tree

In Buttefly Fat Trees (BFTs) switches are connected together in multiple levels, organized in a folded-tree-like structure, with PEs connected to the lowest level, leaf switches. To present various structures, BFTs usually have two types of switches: t switches with two inputs from the lower level and an output to the higher level, and *pi* switches with two inputs from the lower level and two outputs to the higher. Figure 2.16 shows a BFT composed of only *pi* switches, connecting 8 PEs located at the bottom together.

2.7 Alveo U280 board

Xilinx Alveo U280 [50] is a datacenter acceleration card produced by Xilinx. It is a custom-built UltraScale+ FPGA that runs on the alveo architecture and uses the XCU280 FPGA, which in composed of three FPGA dies, called Super Logic Regions (SLRs), connected together using Xilinx Stacked Silicon Interconnect (SSI). Table 2.2 showsshows available resources on the Alveo U280 board.

TABLE 2.2
ALVEO U280 AVAILABLE RESOURCES

Resource	Capacity
PCIe interface	Gen3 $\times 3$, Gen4 $\times 8$, CCIX
HBM2 total capacity	8 GB
HBM2 total bandwidth	460 GB/s
Look-up tables (LUTs)	1304K
Registers	2607K
DSP slices	9024
Block RAMs	960
Ultra RAMs	960
DDR total capacity	32GB
DDR maximum data rate	2400 MT/s

2.7.1 Laguna registers and Super Long Lines (SLLs)

In order to communicate between different SLRs in multi-die Xilinx FPGAs, like Alveo u280, Laguna registers and Super Long Lines (SLLs) are used to transmit data fast and efficiently between different SLRs. SLLs are long vertical wires spanning FPGA chips that connect different SLRs together. Laguna registers are special registers in Xilinx FPGAs designed to connect to SLLs efficiently to communicate between SLRs. In order to route data between SLRs without imposing heavy delays, the SLR crossing data should cross between a pair of Laguna registers (each in a separate SLR) using SLLs without having any logic on the path as having logic on the path interrupts the Laguna–SLL–Laguna path and imposes heavy delays on the design. Alveo U280 has 23040 Laguna register pairs SLR0 and SLR1 communication and 23040 Laguna register pairs for SLR1 and SLR2 communication. Figure 2.17 shows an overview of SLR connections using SLLs and Laguna registers [5].

2.8 Vivado HLS

Vivado High Level Synthesis (HLS) converts higher-level languages like C, C++ into RTL like Verilog, which enables higher level language programs to directly target and deploy on Xilinx devices. HLS greatly boosts development speed and creativity by eliminating the need to manually create RTL designs. However, HLS designs usually suffer from lower operating frequencies and higher resource usage, and need various compiler directives to work properly. In the remain-

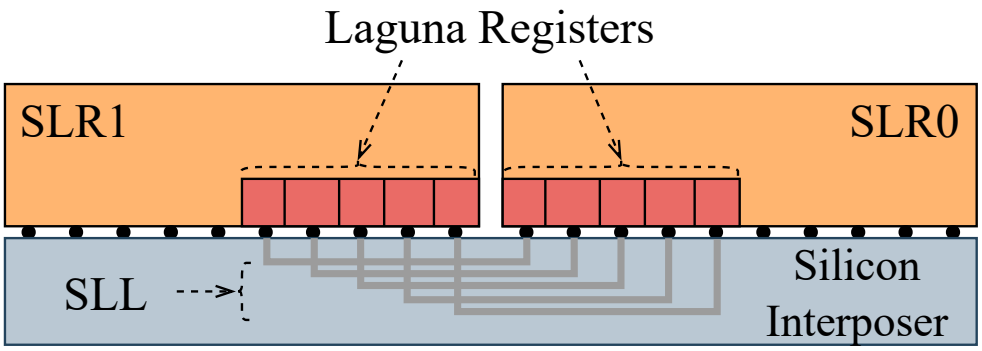

Figure 2.17. SLR connections using SLLs and Laguna registers.

der of this section, we go through running a simple example on Vivado HLS and observe the effect of compiler directives on the resulting designs. We use Vivado HLS to implement poly function, which implements $ax^2 + bx + c$, on a Xilinx 'xc7z020clg400' device. The function code is shown in Code block 2.1.

```
1 void poly(int x, int a, int b, int c, int* y)
2 {
3   (*y)=a*x*x+b*x+c;
4 }
```

Listing 2.1: Poly Function

First, we direct Vivado to pipeline the function for an Initiation Interval (II) of 1, accepting inputs every cycle, and aim for a clock period of 2 ns. Running Vivado to Synthesize the design results in a pipelined design with an II of 1, but fails to achieve a 2 ns clock period and gets 2.79 ns instead. The generated design has a latency of 42 cycles and requires 6 DSP48s, 1532 FFs, and 374 LUTs. The 42-cycle latency is due to pipelining, which breaks operations into smaller chunks for speeding up the design. Changing the target clock period to 2 ns results in a design with a latency of 4 cycles, as the 2 ns clock period is long enough to accommodate each operation in the code. The resulting design consumes fewer resources and needs 6 DSP48s, 292 FFs, and 154 LUTs. Next, we modify the experiment by targeting II=2 and a clock period of 2 ns. Doing so does not improve previous timing and results in a clock period of 2.79 ns and latency of 42 cycles. Increasing II also has a negligible effect on resource usage and the resulting design uses 6 DSP48s, 1513 FFs, and 374 LUTs.

None of the experiments so far imposed any constraints on the computing resources available. To see the effect of putting resource constraints, we run an experiment targeting a clock period

TABLE 2.3
SUMMARY OF VIVADO HLS EXPERIMENTS

Function	Target Clock (ns)	Target II	Resource Limit	Achieved Clock (ns)	Achieved II	Resource Usage			Latency (cycles)
						DSP	FF	LUT	
Poly	2	1	-	2.79	1	6	1532	374	42
Poly	10	1	-	10	1	6	292	154	4
Poly	2	2	-	2.79	2	6	1513	374	42
Poly	10	1	1adder, 1 multiplier	10	2	3	197	141	5
poly_loop	2	1	-	2.79	1	6	1530	450	1068
poly_loop (unroll)	2	1	-	2.79	1	48	10837	2398	172

of 10 ns and limiting the design to only use 1 adder block and 1 multiplier block. The resulting design meets timing and has a latency of 5 cycles. However, it can no longer support II = 1 and needs at least an II of 2 to operate. As expected, the resulting design has a lower resource usage and only uses 3 DSP48s, 197 FFs, and 141 LUTs.

```
void poly_loop(int x[N], int a, int b, int c, int y[N])
{
  loopA: for(int i=0; i<N; i++) {
    y[i]=a*x[i]*x[i]+b*x[i]+c;
  }
}
```

Listing 2.2: Loop-Based Poly Function

To see the effect of loop unrolling and partitioning, we use a modified loop-based version of the code shown in Code block 2.2. Synthesizing the code with N=1024, a target clock period of 2 ns, and a target II of 1 results in a design with a clock period of 2.79 ns, II of 1, and latency of 1068 cycles, which is close to 1024. The resulting design uses 6 DSP48s, 1530 FFs, and 450 LUTs. We are now able to use loop unrolling to create multiple copies of the datapath and divide the work between them. Since the loop-based code has single-port memory interfaces for accessing x and y, we need to use partitioning in conjunction with unrolling to create parallel ports for accessing x and y, enabling efficient loop unrolling. Synthesizing 'poly_loop' with N = 1024, a target clock period of 2 ns, target II of 1, and an unroll factor of 8 results in a design with a clock period of 2.79 ns, II of 1, and latency of 172 cycles, which is close to 1024/8. As unrolling makes multiple copies of the datapath, the unrolled design needs more resources and uses 48 DSP48s, 10837 FFs, and 2398 LUTs.

During the experiments above, we observed how target clock period, target II, available resource limitations, loop-based code, and unrolling affect performance and resource usage. Table 2.3 summarizes HLS experiments and their resulting designs.

Chapter 3

Hardware Architecture

The Mocarabe architecture consists of PEs connected by a torus, mesh, or Butterfly Fat Tree (BFT) NoC, where PEs execute operations on incoming data and NoC routers control data movement. This chapter will explain the architecture of Mocarabe PEs and NoC switches and show how each Mocarabe NoC topology is formed. Then, we will discuss floorplanning and implementation of Mocarabe overlay on the Alveo U280 FPGA, and show how each NoC topology gets floorplanned on the FPGA fabric.

3.1 Processing Element (PE)

A PE can be configured as either an operator (multiply or add) or a data input/output. PEs store incoming operands in shift registers and select the relevant stored operands as inputs to their ALU at each cycle, as shown in Figure 3.1. If $C > 1$, a multiplexer is used at each PE input to choose incoming data from the relevant channel. Operand and channel selection at each cycle are extracted from the compiler output and put into the context memories (labeled as 'CTX' in Figure 3.1) which act as the select lines for the multiplexers and shift registers. For a given Initiation Interval (II), Mocarabe accepts new inputs, and each element will perform the same task once per II cycles. As a result, each context memory has the size of 'II' configuration words. PE output goes to all NoC channels, as at each channel the switch decides whether to take and route PE output.

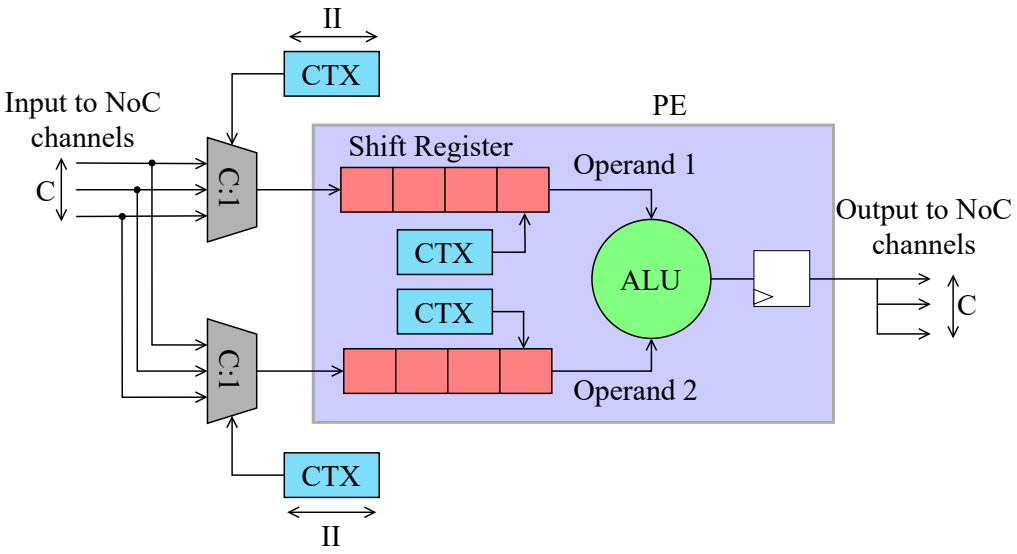

Figure 3.1. Internals of a PE.

3.2 Network Architecture

Mocarabe supports Torus, Mesh, and BFT NoCs as the interconnect network to route data between PEs. A key feature of our architecture is the variable number of parallel physical communication channels [44]. If the number of communication channels is greater than one, PE inputs are fanned in from each channel to both shift registers.

3.2.1 Torus

A Torus-based Mocarabe is a 2D array of building blocks connected by a directional Torus NoC. Each block contains both a PE to execute operations on incoming data and a set of NoC routers to control data movement. Figure 3.2 shows a 2×2 Torus Mocarabe array with C communication channels and I-input PEs.

3.2.2 Mesh

A mesh-based Mocarabe is a 2D array of building blocks connected by a bi-directional mesh NoC. Each block contains both a PE to execute operations on incoming data and a set of NoC

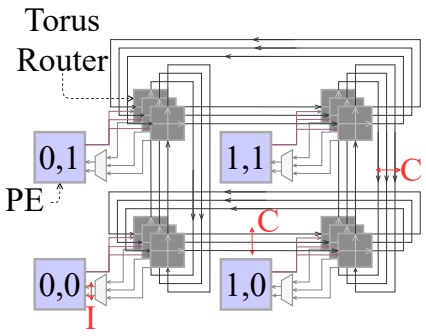

Figure 3.2. A 2 × 2 Mocarabe array with I-input PEs and a C-channel Torus NoC.

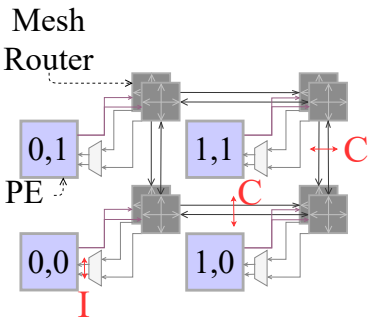

Figure 3.3. A 2 × 2 Mocarabe array with I-input PEs and a C-channel Mesh NoC.

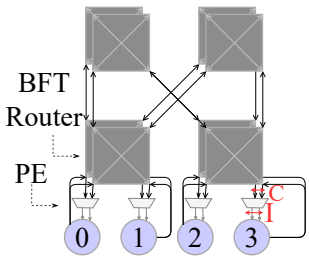

Figure 3.4. A Mocarabe array with 4 I-input PEs and a C-channel BFT NoC.

routers to control data movement. Figure 3.3 shows a 2 × 2 Mocarabe array with with C communication channels and I-input PEs.

3.2.3 Butterfly Fat Tree

A BFT-based Mocarabe is an array of PEs connected by a BFT NoC. Each PE executes operations on incoming data and communicates with other PEs using a BFT NoC, which is composed of *pi* switches. Each *pi* switch is connected to two switches at the next level and two switches or PEs(at level 0 switches) at the previous level. PEs are located at the leaf nodes and are connected to the level 0 switches. Figure 3.4 shows a 4-PE Mocarabe array with C communication channels and I-input PEs.

3.3 NoC Switch

Each communication channel is composed of multiple connected NoC switches exchanging data. Every switch accepts inputs from the local PE (in Torus or Mesh) and the neighbors on the same channel and sends outputs to the local PE (in Torus or Mesh) and the neighbors on the same channel. As Mocarabe is statically scheduled, NoC switches have RAMs (in the form of context memories) that hold the schedule and control their routing.

3.3.1 Torus Switch

Each Torus NoC switch takes data from the PE inside its block or its western and southern neighbors and sends data to the PE or its northern and eastern neighbors. Figure 3.5 shows the internal components of a Torus switch. Context memories are connected to output multiplexers and choose the data going to each output port at every cycle.

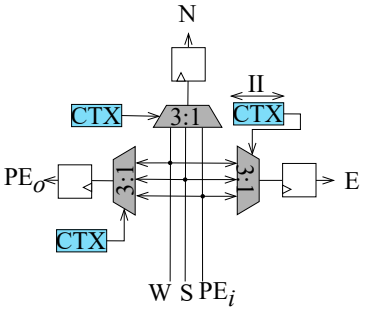

Figure 3.5. Internals of a Torus switch.

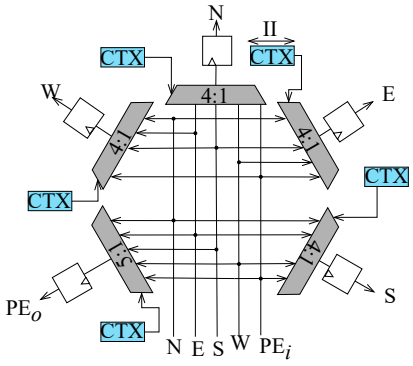

Figure 3.6. Internals of a mesh switch.

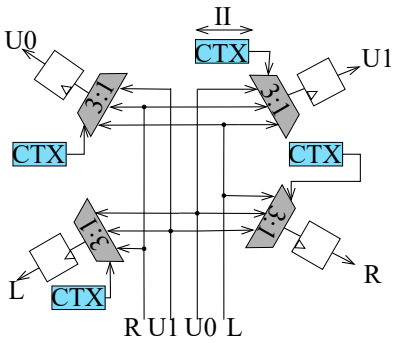

Figure 3.7. Internals of a BFT switch.

3.3.2 Mesh Switch

Each mesh NoC switch takes data from the PE inside the block or its western, eastern, northern, or southern neighbors and sends data to the PE or its western, eastern, northern, or southern neighbors. Figure 3.6 shows the internal components of a mesh switch. Each output port multiplexer is controlled by context memories extracted from compiler output.

3.3.3 BFT Switch

Each BFT NoC switch takes and sends data data to two higher level switches (labeled U0 and U1) and two lower level switches (labeled L and R). Figure 3.7 shows the internal components of a BFT switch. Each output port multiplexer is controlled by context memories extracted from compiler output.

Unlike other CGRAs [24][16][14], which have fixed array sizes, an application can be mapped over a subset of all available PEs and unrolled (repeated) by tiling over the full array. Table 3.1 shows the average resource consumption and maximum operating frequency of each PE and

router on Alveo U280. The high operating frequency of our modules is due to careful floorplanning and pipelining, which will be discussed in the upcoming sections.

TABLE 3.1
PE AND ROUTER RESOURCE USAGE

	LUT (logic)	LUT (memory)	FF	DSP blocks	Fmax
Adder PE	40	80	146	0	1GHz
Multiplier PE	9	64	148	3	986 MHz
Torus router	99	0	290	0	1 GHz
Mesh router	130	0	514	0	1 GHz
BFT router	129	0	386	0	1 GHz

3.4 Context memories

The entire Mocarabe architecture is designed for statically-scheduled, time-multiplexed operation. Every routing and functional resource will repeat the same task, accept inputs, and drive outputs in a repeating phase of II cycles. II is thus also the number of operations mapped to a resource which can enable larger applications to be mapped to fewer blocks at the cost of more LUTs to drive multiplexer select lines. II is the number of cycles in the modulo schedule found by the compiler. Operation execution and data movement are statically scheduled and encoded as multiplexer select or shift register address line memories. Statically-scheduled operation allows for simpler NoC router designs as routers do not need to make routing decisions and usual routing difficulties like back pressure will not occur. Furthermore, a static schedule will lessen data overheads like packet address, and allows for a much more deterministic approach. We use the compiler output to generate context memories that determine how each CGRA element operates at every cycle.

3.5 CGRA implementation and floorplanning

We implement the Mocarabe overlay using parametric Verilog for PEs and switches. We use Xilinx Vivado 2020.1 to synthesize, place, and route the design on a Xilinx Alveo U280 card for analysis. We use Vivado in command-line mode and use TCL scripts to control the flow. We use the XDC flow to supply frequency and placement constraints.

Floorplanning allows choosing a suitable grouping and connectivity of logic in a design and manually placing logic blocks in an FPGA. It aims to make a place and routed design meet timing by increasing density, routability, or performance. During floorplanning, a Physical block (Pblock) spans a part of the FPGA and contains various resources like LUTs, FF, etc. When a logic block is assigned to a Pblock, Vivado tries to use the resources inside the Pblock for placing the assigned logic block. We design hand-crafted placement scripts to effectively map the design and make use of FPGA resources while keeping the operation frequency high.

3.5.1 Torus implementation

Each logical block containing PE and switches is assigned to a physical block (Pblock) on the chip. We define an arbitrary estimate for each Pblock's size as the number of logic slices it contains, with each slice containing Look-Up Tables (LUTs) and flip flops. For instance, a 10×10 Pblock can span the chip from slice X0Y0 to slice X9Y9, creating a rectangular area over the device that contains 100 slices. We use interleaved Torus placement to eliminate the long toroidal critical path between different array ends. During our floorplanning experiments we notice that the unused portions at each Pblock add to routing delay. We place and route a 10×10 array with multipliers on the first column and the rest of the blocks being adders for varying Pblock sizes up to 100 slices shown in Figure 3.8. Compact Pblocks generally deliver higher frequency with the 8×10 PE achieving the highest frequency of 980MHz. To scale up the CGRA and span

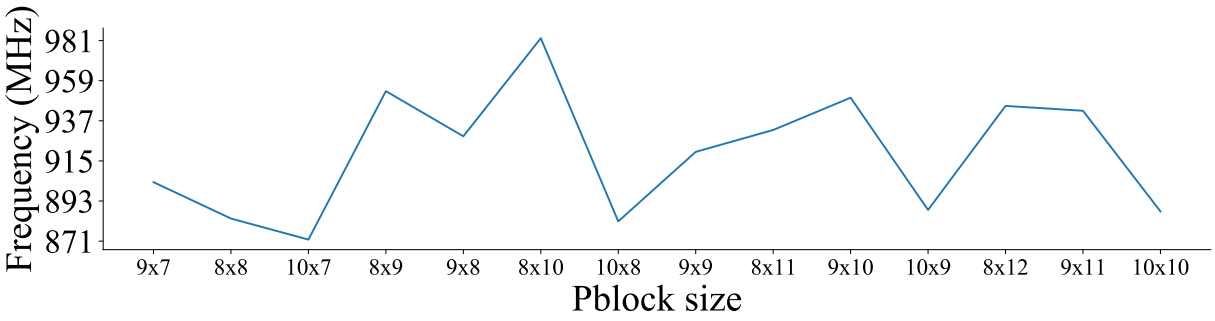

Figure 3.8. Frequency as a function of block size for a 10×10 Torus Mocarabe array.

the FPGA, we set the array size to 20×60, with the same configuration of having multipliers on the first column and the rest of the blocks being adders. Spanning the entire FPGA required communication between different SLRs which lead to large routing delays between SLRs. We performed multiple tweaks to the placement and RTL to maintain high operating frequencies at large sizes. We fixed Pblock size to 10×10 in order to avoid having single Pblocks spread

among multiple SLRs. Furthermore, we added extra pipeline registers to each router's output to the nearby routers and registered the incoming data. We forced Vivado to map the router registers that had incoming or outgoing SLR crossing nets to SLR crossing Laguna registers, which are connected together by Super Long Lines (SLLs), inside the router's corresponding Pblock. We use Vivado's "phys_opt_design" compiler option to include various physical optimizations (e.g. an SLR-crossing optimization), once after placement and once after routing to enable the 20×60 array to run at 650 MHz+.

Although the 20×60 array operates at 650 MHz, the operation frequency is not stable between different operator configurations. For instance, having all blocks performing multiplication results in a noticeable frequency drop to 400 MHz. The drop in frequency is because some PEs use DSPs placed far away outside their Pblocks. To overcome this issue, we modify the placement scripts to force each PE to only use the DSPs located inside its Pblock and added paddings to array placement on the chip to make sure each Pblock contained at least the amount of DSP blocks needed by each PE. The 10×10 size for each Pblock is a suitable option here as well to make sure all Pblocks receive the resources they need. By using the updated placement scripts, we are able to scale the array size to 19×69, spanning the entire chip and operate at 640 MHz for an array composed only from multiplier PEs. To further increase the operation frequency we pipelined the router outputs to PEs, which improved operation frequency, taking the all-multiplier 19×69 array frequency up from 640 – 740MHz. Since some applications at a given II require higher channel counts, we support 1–3 channel, two-input PE configurations. Figure 3.9 shows a dual-channel 19×69 array's device view in Vivado.

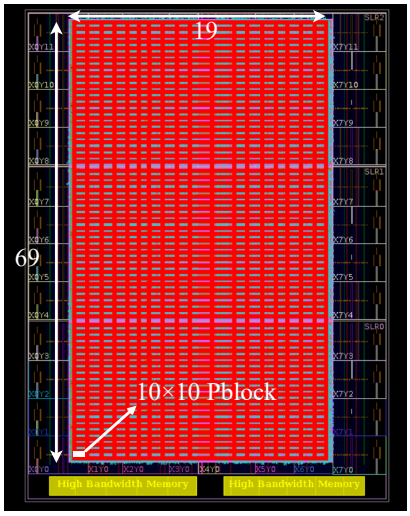

Figure 3.9. A 19×69 Torus Mocarabe device view.

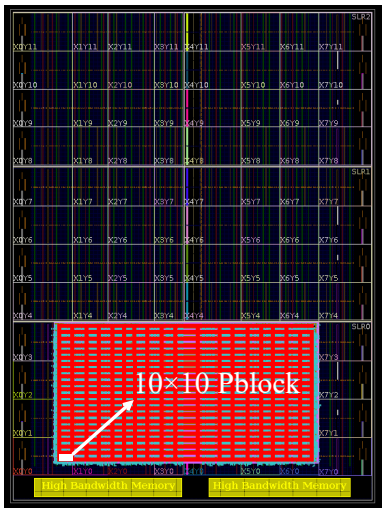

Figure 3.10. A 19×22 Mesh Mocarabe device view.

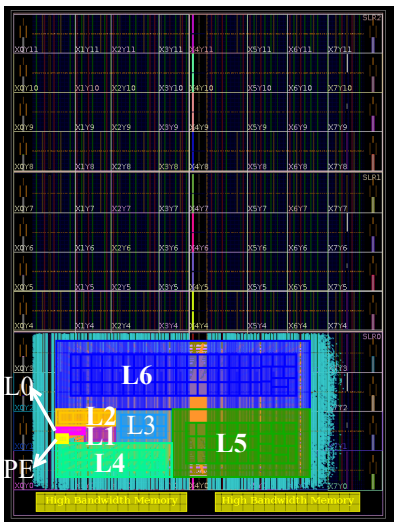

Figure 3.11. A 256 PE BFT Mocarabe device view.

3.5.2 Mesh implementation

Mesh implementation is similar to Torus. Each logic block containing PE and switches is assigned to a Pblock on the chip, and we use 10×10 sized Pblocks as it gives enough resources to each logical block and prevents Pblocks from spilling between different SLRs. Each logic block is forced to to only use the resources (LUT, DSP, etc.) available in its assigned Pblock. We implement a 19×22 mesh Mocarabe and provide 1–2 channels to satisfy applications' needs. Like Torus, we pipeline computations and communications and use various Vivado compiler directives like "phys_opt_design" to perform various physical optimizations once after placement and once after routing. The 19×22 mesh array runs at 788 – 856 MHz. Figure 3.10 shows a 19×22 mesh Mocarabe device view in Vivado.

3.5.3 BFT implementation

Unlike mesh and Torus, we assign PEs and routers to separate Pblocks to implement BFT. This is because instead of having logic blocks connected together in a 2D array, BFT is composed of a set of PEs communicating with each other using a multi-layered network of NoC routers. As a result, we assign PEs and each layer of routers to different sets of Pblocks and place different layers on top of each other using Vivado's nested Pblocks. Nested Pblocks allows having multiple layers of Pblocks placed on top of each other if each lower level Pblock is covered by only one higher level Pblock at each level above and do not cross between multiple Pblocks at each higher level. As an example, if Pblock A is placed on top of Pblock B, Pblock A must cover Pblock B entirely and there must be no parts of Pblock B spilling out of Pblock A.

To implement BFT Mocarabe, first, we assign PE Pblocks each accommodating a single PE. Then, we assign Pblocks for each layer of NoC switches. At layer 0, each NoC Pblock spans 2 PE Pblocks and contains a NoC switch that connects the underlaying PEs together. At layer 1, each NoC Pblock spans 4 PE Pblocks and contains 2 NoC switches that connect the level 0 switches together. This way, we place each layer's Pblocks on top of the previous layer's Pblocks until all layers have their Pblocks assigned, with each layer l having 2^l switches assigned to a Pblock and each Pblock spanning 2^{l+1} PEs. Assigning more switches to each Pblock at higher levels makes placement easier and gives vivado more freedom to achieve an optimal placement. Figure 3.12 shows Pblock placements for an 8 PE BFT and how each layer of Switches gets placed on top of the previous layer. Figure 3.11 shows a 256 PE BFT Mocarabe's device view in Vivado highlighting a PE Pblock and a Pblock for each NoC level (except level 7 which covers the entire array), showing a pattern similar to Figure 3.12 (each level of switches covers lower level ones). Furthermore, like Torus and Mesh, we force each PE to use DSP blocks inside its

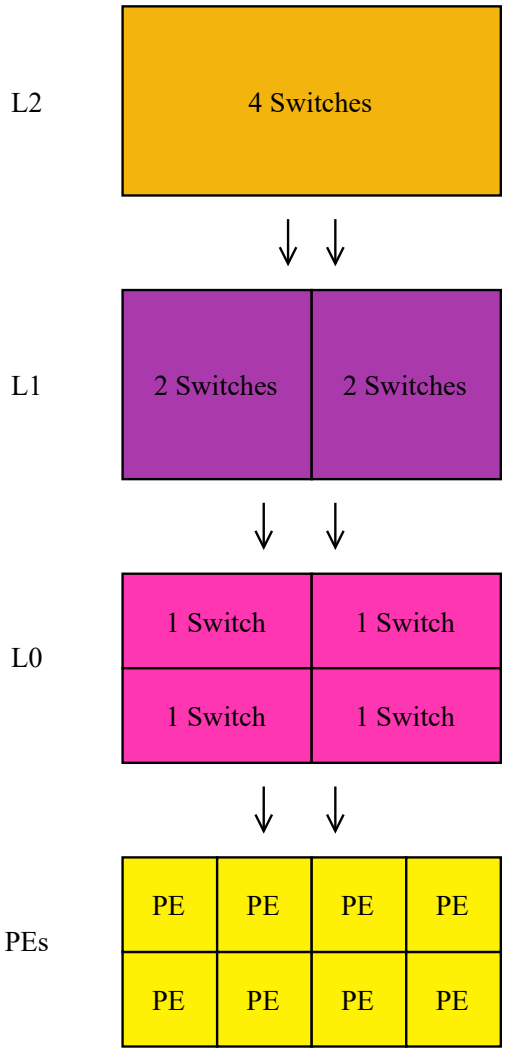

Figure 3.12. The multilayered implementation of an 8 PE BFT.

assigned Pblock and use Vivado compiler directives and optimizations to maximize performance. The 256 PE BFT array runs at 583 – 746 MHz.

Chapter 4

Compiler

In this chapter, we present Mocarabe compiler flow. We give a detailed explanation about different parts of our compiler by explaining allocation, partitioning, placement, and scheduling.

4.1 Framework overview

The Mocarabe compiler framework extracts the data-flow graph (DFG) from a C kernel (with gcc and GIMPLE) and generates an architecture configuration and a schedule to coordinate data movement. The compiler flow, shown in Figure 4.1, consists of four phases: ① operator allocation, ② DFG partitioning, ③ placement, and ④ scheduling; the objective is to find a feasible schedule with a minimum number of channels. We use simple linear search for channels, and most of our benchmarks need 2–3 torus channels or 1–2 BFT or mesh channels when scheduled with ILP.

Compared to SDC scheduling [9], data dependencies are handled differently in our scheduler. We schedule each DFG edge in isolation, but use rotating registers to ensure correct alignment of data dependency. Frequency constraints ('cycle time') found in SDC schedulers are not found in the compiler, as the overlay is guaranteed to run close to the FPGA fabric f_{max} (645–768 MHz torus, 788–856 MHz mesh, 583–746 MHz BFT).

Unlike the CGRA-ME ILP scheduler [7] that unifies partitioning, placement, and scheduling into a monolithic ILP formulation, we split these tasks into separate disjoint phases to ensure feasible computational runtime for large problem sizes. The CGRA-ME scheduler routinely times out after 24 hours for a benchmark with an operation count under 30. When using mesh or torus, our compiler can tackle 80 operations in less than 30 minutes, with most benchmarks

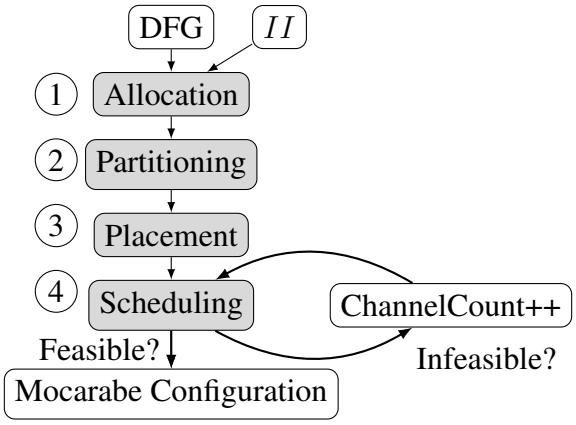

Figure 4.1. Mocarabe compiler flow.

taking less than a minute. Due to the increased number of BFT switches, it takes longer to perform scheduling for BFT, although most benchmarks take less than 12 hours to complete.

4.1.1 Formal Description

The DFG is encoded in a hypergraph format [35] to retain multi-fanout attributes. A DFG is comprised of a set of nodes, Ops, which represent operations, inputs, and outputs, while edges, $Vals$, represent the data dependencies between Ops. The kernel in Figure 4.2 is mapped to the torus architecture as a motivating example, with $II = 2$ (dual-context).

4.2 Operator Allocation and Partitioning

Given a context count II, the compiler allocates K PEs. DFG Ops are partitioned among the K PEs, each PE holding at most II Ops to allow for time-multiplexed operator sharing. We use an ILP formulation for hypergraph partitioning [19] that is solved using the Gurobi solver [12].

Variables: The formulation has two binary variables.

- $ValInPartition_{j,k} = 1$ indicates that DFG edge j is entirely in partition k.
- $OpInPartition_{op,k} = 1$ indicates that DFG node op is in partition k.

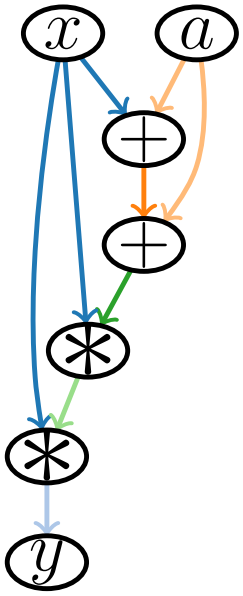

Figure 4.2. Example Dataflow for $y = (2a + x) \times x^2$.

Constraints:
Every operator must be in exactly one partition.

$$\sum_{k}^{K} OpInPartition_{op,k} = 1, \forall op \in Ops \tag{4.1}$$

No more than II operators can be mapped to one partition.

$$\sum_{op}^{Ops} OpInPartition_{op,k} \leq II, \forall k \in K \tag{4.2}$$

Graph dependency is encoded with the following constraint. $Op(j)$ denotes all DFG nodes incident on edge j.

$$\begin{gathered} OpInPartition_{op,k} >= ValInPartition_{j,k}, \\ \forall j \in Vals, \forall op \in Op(j), \forall k \in K \end{gathered} \tag{4.3}$$

There is a fixed number of partitions for each operator type. $K(op)$ denotes the partitions re-

served for op's operator type (e.g. '*').

$$\sum_{k}^{K(op)} OpInPartition_{op,k} = 1, \forall op \in Ops \tag{4.4}$$

Objective Function:
The objective function is set to minimize the sum of cut nets ($Vals$), which is encoded as the maximization of uncut nets.

$$Maximize \sum_{j}^{Vals} \sum_{k}^{K} ValInPartition_{j,k} \tag{4.5}$$

A partition may only group operations of the same type into groups no larger than II. The motivating example has four operator types (input, add, multiply, output) and the resulting partitioning is shown in Figure 4.3. Here, the sum is two uncut nets, one between both adds and another between both multiplies. Note that in general, operations need not be neighbours to be in the same partition.

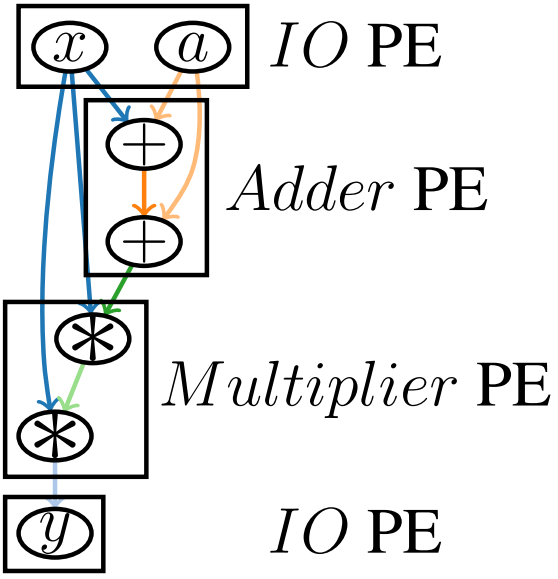

Figure 4.3. DFG Partitioned into CGRA PEs.

4.3 Placement

Every DFG operator is mapped to one of K partitions, which must then be placed in a specific (x, y) (torus,mesh) or n (BFT) location. We use Simulated Annealing (SA) [31], an approach which has been used in other CGRA compilers [8] as a fast placement option and propose an topology-agnostic ILP-based placer to find the optimal placement, given enough time.

4.3.1 SA placer

Any PE location can be fixed to be any type of operator. The placer's objective is to minimize the minimal torus, mesh, or BFT distance a net must travel, as in (4.6). The placer and the scheduler are decoupled, but the aim is to provide the next stage with a placement that will enable it to find a feasible solution using the fewest parallel channels. Moves from one placement state to another are unrestricted. The cost of a certain placement is the sum, across every source-destination pairs, of the squared distance each net would have to travel through the interconnect. The result is a mapping from every DFG operator to its PE, which is used to create a netlist to be scheduled in the next step.

$$Minimize \sum_{j}^{Vals} (MinDistance(source(j), dest) \quad \forall dest \in dests(j)). \tag{4.6}$$

4.3.2 ILP placer

To achieve the optimal placement, we propose a topology-agnostic ILP placer as an alternative to the SA placer. The ILP placer formulates placement as an ILP problem and solves it using Gurobi solver. When mapping M operators to N PEs ($N \geq M$), we define the following binary and continuous variables to encode the placement:

- $Z_{m,n}$: Operation m is placed on PE n.
- w_{op_a,op_b,n_a,n_b}: For connected operators op_a and op_b where op_a produces an input for op_b, op_a is placed on PE n_a and op_b is placed on PE n_b. As a result, $w_{op_a,op_b,n_a,n_b} = Z_{op_a,n_a} \wedge Z_{op_a,n_a}$.
- d_{op_a,op_b}: The distance between two connected operators. d_{op_a,op_b} is calculated as follows:

$$d_{op_a,op_b} = \sum_{n_a=0}^{N-1} \sum_{n_b=0=}^{N-1} w_{op_a,op_b,n_a,n_b}.D_{n_a,n_b} \tag{4.7}$$

Where D_{n_a,n_b} is the distance between PEs n_a and n_b in the given topology and can be calculated using any procedure.

ILP Constraints and Objective Function: We present the following constraints for placement ILP formulation:

- **PE exclusivity:** Each PE may be used by at most one operator.

$$\forall n \sum_{m=0}^{N-1} Z_{m,n} \leq 1 \tag{4.8}$$

- **Operator exclusivity:** Each operator is mapped to exactly one PE.

$$\forall m \sum_{m=0}^{M-1} Z_{m,n} = 1 \tag{4.9}$$

- **AND constraint:** To encode the "$\wedge$" operator in $w_{op_a,op_b,n_a,n_b} = Z_{op_a,n_a} \wedge Z_{op_a,n_a}$ into the ILP formulation, we can use the following constraints [3]:

$$\begin{aligned} w_{op_a,op_b,n_a,n_b} &\leq Z_{op_a,n_a} \\ w_{op_a,op_b,n_a,n_b} &\leq Z_{op_b,n_b} \\ w_{op_a,op_b,n_a,n_b} + 1 &\geq Z_{op_a,n_a} \wedge Z_{op_a,n_a} \end{aligned} \tag{4.10}$$

- **Objective function:** Our objective function minimizes the sum of all distances between all connected operators.

$$Minimize \sum_{op_a}^{M} \sum_{op_b}^{M} d_{op_a,op_b} \tag{4.11}$$

As we pre-compute all distances between all locations, our ILP placer is topology-agnostic and can place to any topology as the distances are given as input to the placer. Furthermore, pre-computing the distances allows us to minimize placement cost even for some non-linear optimization objectives like quadratic wirelength, as the distances are pre computed and are given as numbers to the ILP placer

4.4 Scheduling

SDC schedulers encode the execution cycle of an operation as an integer. Our scheduling problem, formulated as an ILP and solved with the Gurobi solver [12], encodes resource occupancy in space and time as boolean unknowns. The resulting ILP is larger, but we make it feasible by limiting schedule length (the time dimension) to II and realigning dependencies after scheduling. The input is a netlist of sources and destinations on the array with the same number of Ops and $Vals$ as the input DFG, but with decoupled dataflow dependencies (which will be realigned after scheduling). The resulting modulo schedule has up to II cycles. We now show how to set up this formulation for each NoC topology.

4.4.1 Torus scheduling

ILP Variables: We define six sets of binary variables, grouped into pairs. The connectivity between some of these is illustrated in Figure 4.4.

- $R^h_{(x,y,t,c),j}$ and $R^v_{(x,y,t,c),j}$: routing resource at (x, y) on channel c is used by value j in cycle t. Horizontal and vertical routing resources are denoted by ^{h} and ^{v}, respectively. $R^{h,v}$ indicates that a constraint applies to both types of resources independently.
- $EnterRouting_{(x,y,t),j}$: At cycle t, value j leaves PE (x, y). $EnterChannel_{(x,y,t,c),j}$ specifies which channel to use.
- $ExitRouting_{(x,y,t),j}$: At cycle t, value j enters PE (x, y). $ExitChannel_{(x,y,t,c),j}$ specifies which channel to use.

For the sake of brevity, we denote the tuple (x, y, t) as "i". For example, $R^h_{(i,c),j}$ represents value j's use of the horizontal routing resource at (x, y, t) on channel c. A denotes $M \times N \times II$, the cube over the 2D array and the schedule length II.

ILP Constraints and Objective Function: We present a number of constraints for the ILP formulation. The first six sets are somewhat trivial, while the last is the core of how the formulation encodes data movement.

Source/Destination Mapping: A value must leave its source exactly once, but can do so any time. For all values j in $Vals$, with source PE $src_{x,y}(j)$,

$$\sum_{t=0}^{T} EnterChannel_{(src_{x,y}(j),t,c),j} = 1. \tag{4.12}$$

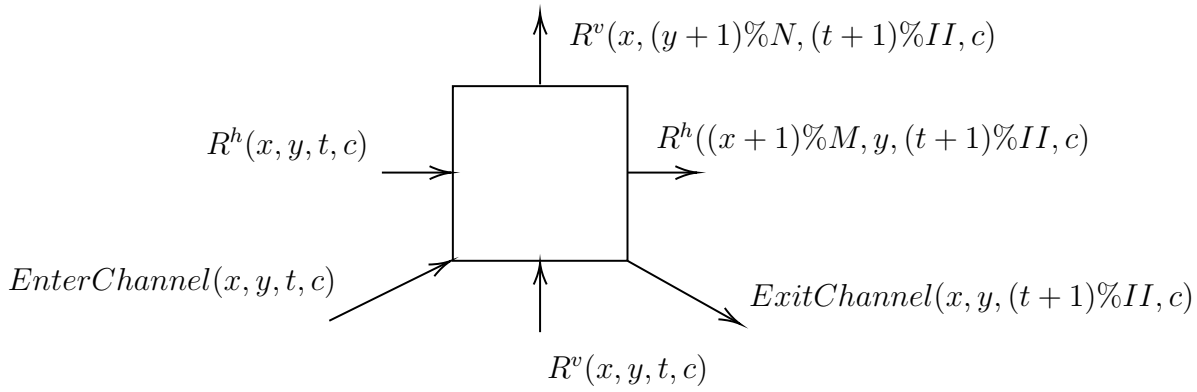

Figure 4.4. ILP Variables for Scheduling on Torus.

A value j must enter all of its destinations exactly once. For every j in $Vals$ with destination $dest_{x,y}(j)$,and for all $t \in II$,

$$\sum_{t=0}^{T} ExitRouting_{(dest_{x,y}(j),t),j} = 1. \tag{4.13}$$

A value cannot enter or exit any PE that is not its source or one of its destinations. For every j in $Vals$, and every location which is not one of j's destinations, $notDest_{x,y}(j)$, and for all $t \in II$,

$$\sum_{t=0}^{T} ExitRouting_{(notDest_{x,y}(j),t),j} = 0. \tag{4.14}$$

Routing Resource Exclusivity: Each routing resource may be used by at most one value in each cycle.

$$\sum_{j}^{Vals} R^{h,v}_{(i,c),j} \leq 1, \forall i \in A, \forall c \in C. \tag{4.15}$$

$$\begin{aligned} \sum_{j}^{Vals} EnterChannel_{(i,c),j} \leq 1, \forall i \in A, \forall c \in C \\ \sum_{j}^{Vals} EnterRouting_{(i),j} \leq 1, \forall i \in A. \end{aligned} \tag{4.16}$$

Single-Channel Entry: When a value enters the NoC, it must choose a single channel. For all $i \in A, c \in C, j \in Vals$,

$$\sum_{c}^{C} EnterChannel_{(i,c),j} \geq EnterRouting_{(i),j} \tag{4.17}$$

$$EnterChannel_{(i,c),j} \leq EnterRouting_{(i),j}. \tag{4.18}$$

PE Input: A PE can absorb at most 2 values from the NoC each cycle.

$$\sum_{j}^{Vals} ExitChannel_{(i,c),j} \leq 2, \forall i \in A, \forall c \in C. \tag{4.19}$$

Single-Channel Exit: Similarly, when a value exits a PE, it must choose a single channel. For all $i \in A, c \in C, j \in Vals$

$$\sum_{c}^{C} ExitChannel_{(i,c),j} \geq ExitRouting_{(i),j} \tag{4.20}$$

$$ExitChannel_{(i,c),j} \leq ExitRouting_{(i),j}. \tag{4.21}$$

Value Propagation: These core constraints illustrate the interconnect's torus connectivity and modulo scheduling.

This constraint ensures that information cannot be created from nothing. This is encoded for horizontal routing (4.22), vertical routing (4.23), and for leaving the NoC (4.41). For all $(x, y, t) \in A, c \in C, j \in Vals$,

$$R^{h}_{(x,y,t,c),j} + R^{v}_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq R^{h}_{((x+1)\%M,y,c,(t+1)\%II),j}, \tag{4.22}$$

$$R^{h}_{(x,y,t,c),j} + R^{v}_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq R^{v}_{(x,(y+1)\%N,c,(t+1)\%II),j}, \tag{4.23}$$

$$R^{h}_{(x,y,t,c),j} + R^{v}_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq ExitChannel_{(x,y,c,(t+1)\%II),j}. \tag{4.24}$$

A value must fan-out to at least one routing resource, i.e. information cannot be destroyed. For all $\forall i \in A, \forall c \in C, \forall j \in Vals$

$$\begin{aligned} R^h_{((x+1)\%Nx,y,c,(t+1)\%II),j} + R^v_{(x,(y+1)\%Ny,c,(t+1)\%II),j} + \\ ExitChannel_{(x,y,c,(t+1)\%II),j} \\ \geq R^h_{(i,c),j} + R^v_{(i,c),j} + EnterChannel_{(x,y,t,c),j}. \end{aligned} \tag{4.25}$$

Objective Function: Our objective function minimizes the sum of all routing resources used, across every PE and every location.

$$\sum_{i}^{A} \sum_{c}^{C} \sum_{j}^{Vals} R^h_{(i,c),j} + R^v_{(i,c),j} \tag{4.26}$$

4.4.2 Mesh scheduling

ILP Variables: We define six sets of binary variables. The connectivity between some of these is illustrated in Figure 4.5.

- $R^s_{(x,y,t,c),j}$, $R^n_{(x,y,t,c),j}$, $R^e_{(x,y,t,c),j}$, and $R^w_{(x,y,t,c),j}$: routing resource at (x, y) on channel c is used by value j in cycle t. Northbound, southbound, eastbound, and westbound routing resources are denoted by ^{n}, ^{s}, ^{e}, and ^{w}, respectively. $R^{n,s,w,e}$ indicates that a constraint applies to multiple types of resources independently.
- $EnterRouting_{(x,y,t),j}$: At cycle t, value j leaves PE (x, y). $EnterChannel_{(x,y,t,c),j}$ specifies which channel to use.
- $ExitRouting_{(x,y,t),j}$: At cycle t, value j enters PE (x, y).
  $ExitChannel_{(x,y,t,c),j}$ specifies which channel to use.

For the sake of brevity, we denote the tuple (x, y, t) as "i". For example, $R^n_{(i,c),j}$ represents value j's use of the north bound routing resource at (x, y, t) on channel c. A denotes $M \times N \times II$, the cube over the 2D array and the schedule length II.

ILP Constraints and Objective Function: We present a number of constraints for the ILP formulation. The first six sets are somewhat trivial, while the last is the core of how the formulation encodes data movement.

Source/Destination Mapping: A value must leave its source exactly once, but can do so any

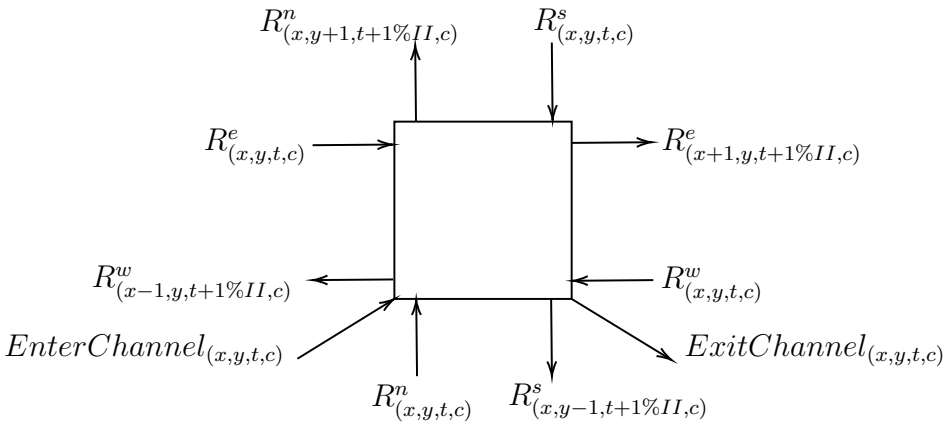

Figure 4.5. ILP Variables for Scheduling on Mesh.

time. For all values j in $Vals$, with source PE $src_{x,y}(j)$,

$$\sum_{t=0}^{T} EnterChannel_{(src_{x,y}(j),t,c),j} = 1. \tag{4.27}$$

A value j must enter all of its destinations exactly once. For every j in $Vals$ with destination $dest_{x,y}(j)$,and for all $t \in II$,

$$\sum_{t=0}^{T} ExitRouting_{(dest_{x,y}(j),t),j} = 1. \tag{4.28}$$

A value cannot enter or exit any PE that is not its source or one of its destinations. For every j in $Vals$, and every location which is not one of j's destinations, $notDest_{x,y}(j)$, and for all $t \in II$,

$$\sum_{t=0}^{T} ExitRouting_{(notDest_{x,y}(j),t),j} = 0. \tag{4.29}$$

Routing Resource Exclusivity: Each routing resource may be used by at most one value in each cycle.

$$\sum_{j}^{Vals} R^{n,s,e,w}_{(i,c),j} \leq 1, \forall i \in A, \forall c \in C. \tag{4.30}$$

PE Output: A PE can emit at most one value per cycle, and cannot enter multiple channels simultaneously.

$$\sum_{j}^{Vals} EnterChannel_{(i,c),j} \leq 1, \forall i \in A, \forall c \in C$$
$$\sum_{j}^{Vals} EnterRouting_{(i),j} \leq 1, \forall i \in A. \tag{4.31}$$

Single-Channel Entry: When a value enters the NoC, it must choose a single channel. For all $i \in A, c \in C, j \in Vals$,

$$\sum_{c}^{C} EnterChannel_{(i,c),j} \geq EnterRouting_{(i),j} \tag{4.32}$$

$$EnterChannel_{(i,c),j} \leq EnterRouting_{(i),j}. \tag{4.33}$$

PE Input: A PE can absorb at most 2 values from the NoC each cycle.

$$\sum_{j}^{Vals} ExitChannel_{(i,c),j} \leq 2, \forall i \in A, \forall c \in C. \tag{4.34}$$

Single-Channel Exit: Similarly, when a value exits a PE, it must choose a single channel. For all $i \in A, c \in C, j \in Vals$

$$\sum_{c}^{C} ExitChannel_{(i,c),j} \geq ExitRouting_{(i),j} \tag{4.35}$$

$$ExitChannel_{(i,c),j} \leq ExitRouting_{(i),j}. \tag{4.36}$$

Value Propagation: These core constraints illustrate the interconnect's mesh connectivity and modulo scheduling.

This constraint ensures that information cannot be created from nothing. This is encoded for southbound routing (4.37), northbound routing (4.38), westbound routing (4.39), eastbound routing (4.40), and for leaving the NoC (4.41). For all $(x, y, t) \in A, c \in C, j \in Vals$,

$$R^{n}_{(x,y,t,c),j} + R^{w}_{(x,y,t,c),j} + R^{e}_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq R^{s}_{(x,y-1,c,(t+1)\%II),j} \tag{4.37}$$

$$R^{s}_{(x,y,t,c),j} + R^{w}_{(x,y,t,c),j} + R^{e}_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq R^{n}_{(x,y+1,c,(t+1)\%II),j} \tag{4.38}$$

$$R^{n}_{(x,y,t,c),j} + R^{s}_{(x,y,t,c),j} + R^{e}_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq R^{w}_{(x-1,y,c,(t+1)\%II),j} \tag{4.39}$$

$$R^{n}_{(x,y,t,c),j} + R^{s}_{(x,y,t,c),j} + R^{w}_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq R^{e}_{(x+1,y,c,(t+1)\%II),j} \tag{4.40}$$

$$R^{n}_{(x,y,t,c),j} + R^{s}_{(x,y,t,c),j} + R^{w}_{(x,y,t,c),j} + R^{e}_{(x,y,t,c),j} + EnterChannel_{(x,y,t,c),j} \geq ExitChannel_{(x,y,c,(t+1)\%II),j} \tag{4.41}$$

A value must fan-out to at least one routing resource, i.e. information cannot be destroyed. For all $\forall i \in A, \forall c \in C, \forall j \in Vals$

$$R^{n}_{(x,y+1,c,(t+1)\%II),j} + R^{s}_{(x,y-1,c,(t+1)\%II),j} + R^{w}_{(x-1,y,c,(t+1)\%II),j} + R^{e}_{(x+1,y,c,(t+1)\%II),j} + ExitChannel_{(x,y,c,(t+1)\%II),j} \geq R^{n}_{(i,c),j} + R^{s}_{(i,c),j} R^{w}_{(i,c),j} + R^{e}_{(i,c),j} + EnterChannel_{(x,y,t,c),j} \tag{4.42}$$

The blocks located at the boundaries take less inputs and do not use the routing resources that would come from outside the array. For instance, a block located at x = 0 will not take input from the eastbound channel as no data enters the block from right.

Objective Function: Our objective function minimizes the sum of all routing resources used, across every PE and every location.

$$Minimize \sum_{i}^{A} \sum_{c}^{C} \sum_{j}^{Vals} R^{n}_{(i,c),j} + R^{s}_{(i,c),j} + R^{w}_{(i,c),j} + R^{e}_{(i,c),j} \tag{4.43}$$

4.4.3 BFT scheduling

We define six sets of binary variables. The connectivity between some of these is illustrated in Figures 4.6 and 4.7.

- $R^{u0}_{(l,loc,t,c),j}$, $R^{u1}_{(l,loc,t,c),j}$, $R^{d0}_{(l,loc,t,c),j}$, and $R^{d1}_{(l,loc,t,c),j}$: routing resource at level *l* location *loc* on channel c is used by value j in cycle t. Upward straight, upward crossing, downward

straight, and downward crossing routing resources are denoted by ^{u0}, ^{u1}, ^{d0}, and ^{d1}, respectively. As based on BFT topology, routing resources can either go straight up or down, or cross into $2^{0,1,\cdots}$ locations away based on the level. $R^{u0,u1,d0,d1}$ indicates that a constraint applies to multiple types of resources independently.

- $EnterRouting_{(loc,t),j}$: At cycle t, value j leaves PE loc. $EnterChannel_{(loc,t,c),j}$ specifies which channel to use.
- $ExitRouting_{(loc,t),j}$: At cycle t, value j enters PE loc.
  $ExitChannel_{(loc,t,c),j}$ specifies which channel to use.

For N PEs, we would have $log(N)$ levels each with $N/2$ switches. For the sake of brevity, we denote the tuple (l, loc, t) as "i". For example, $R^{d0}_{(i,c),j}$ represents value j's use of the down going straight routing resource at (l, loc, t) on channel c. A denotes $N/2 \times log(n) \times II$, the cube over the array and the schedule length II.

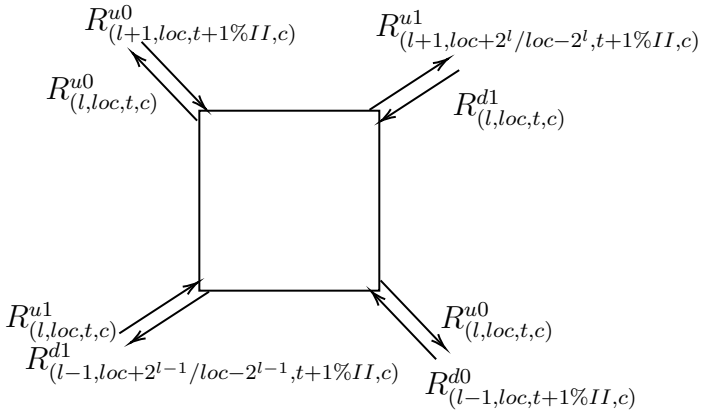

Figure 4.6. ILP Variables for Scheduling on BFT (Interconnect Tree Node).

ILP Constraints and Objective Function: We present a number of constraints for the ILP formulation. The first six sets are somewhat trivial, while the last is the core of how the formulation encodes data movement.

Source/Destination Mapping: A value must leave its source exactly once, but can do so any time. For all values j in $Vals$, with source PE $src_{loc}(j)$,

$$\sum_{t=0}^{T} EnterChannel_{(src_{loc}(j),t,c),j} = 1 \tag{4.44}$$

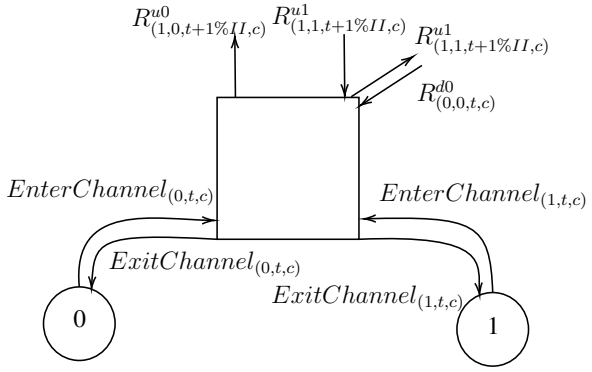

Figure 4.7. ILP Variables for Scheduling on BFT (BFT PE-Switch Connection).

A value j must enter all of its destinations exactly once. For every j in $Vals$ with destination $dest_{loc}(j)$,and for all $t \in II$,

$$\sum_{t=0}^{T} ExitRouting_{(dest_{loc}(j),t),j} = 1 \tag{4.45}$$

A value cannot enter or exit any PE that is not its source or one of its destinations. For every j in $Vals$, and every location which is not one of j's destinations, $notDest_{loc}(j)$, and for all $t \in II$,

$$\sum_{t=0}^{T} ExitRouting_{(notDest_{loc}(j),t),j} = 0 \tag{4.46}$$

Routing Resource Exclusivity: Each routing resource may be used by at most one value in each cycle.

$$\sum_{j}^{Vals} R^{u0,u1,d0,d1}_{(i,c),j} \leq 1, \forall i \in A, \forall c \in C \tag{4.47}$$

PE Output: A PE can emit at most one value per cycle, and cannot enter multiple channels simultaneously.

$$\begin{aligned} \sum_{j}^{Vals} EnterChannel_{(i,c),j} \leq 1, \forall i \in A, \forall c \in C \\ \sum_{j}^{Vals} EnterRouting_{(i),j} \leq 1, \forall i \in A \end{aligned} \tag{4.48}$$

Single-Channel Entry: When a value enters the NoC, it must choose a single channel. For all $i \in A, c \in C, j \in Vals$,

$$\sum_{c}^{C} EnterChannel_{(i,c),j} \geq EnterRouting_{(i),j} \tag{4.49}$$

$$EnterChannel_{(i,c),j} \leq EnterRouting_{(i),j} \tag{4.50}$$

PE Input: A PE can absorb at most 2 values from the NoC each cycle.

$$\sum_{j}^{Vals} ExitChannel_{(i,c),j} \leq 2, \forall i \in A, \forall c \in C \tag{4.51}$$

Single-Channel Exit: Similarly, when a value exits a PE, it must choose a single channel. For all $i \in A, c \in C, j \in Vals$

$$\sum_{c}^{C} ExitChannel_{(i,c),j} \geq ExitRouting_{(i),j} \tag{4.52}$$

$$ExitChannel_{(i,c),j} \leq ExitRouting_{(i),j} \tag{4.53}$$

Value Propagation: These core constraints illustrate the interconnect's BFT connectivity and modulo scheduling.

This constraint ensures that information cannot be created from nothing. This is encoded for upward straight routing (4.54), upward crossing routing (4.55), downward straight routing (4.56), and downward crossing routing (4.57). For all $(l, loc, t) \in A, c \in C, j \in Vals$,

$$R^{u0}_{(l,loc,t,c),j} + R^{u1}_{(l,loc,t,c),j} + R^{d1}_{(l,loc,t,c),j} \geq R^{u0}_{(l+1,loc,(t+1)\%II,c),j} \tag{4.54}$$

$$\begin{aligned}
&\textbf{if } loc\%2^{l+1} < 2^l: \\
&\quad R^{u0}_{(l,loc,t,c),j} + R^{u1}_{(l,loc,t,c),j} + R^{d0}_{(l,loc,t,c),j} \geq R^{u1}_{(l+1,loc+2^l,(t+1)\%II,c),j} \\
&\textbf{else:} \\
&\quad R^{u0}_{(l,loc,t,c),j} + R^{u1}_{(l,loc,t,c),j} + R^{d0}_{(l,loc,t,c),j} \geq R^{u1}_{(l+1,loc-2^l,(t+1)\%II,c),j},
\end{aligned} \tag{4.55}$$

$$R^{u1}_{(l,loc,t,c),j} + R^{d0}_{(l,loc,t,c),j} + R^{d1}_{(l,loc,t,c),j} + \geq R^{d0}_{(l-1,loc,(t+1)\%II,c),j} \tag{4.56}$$

$$\begin{aligned}
&\textbf{if } loc\%2^l < 2^{l-1} : \\
&\quad R^{u0}_{(l,loc,t,c),j} + R^{d0}_{(l,loc,t,c),j} + R^{d1}_{(l,loc,t,c),j} + \geq R^{d1}_{(l-1,loc+2^{l-1},(t+1)\%II,c),j} \\
&\textbf{else:} \\
&\quad R^{u0}_{(l,loc,t,c),j} + R^{d0}_{(l,loc,t,c),j} + R^{d1}_{(l,loc,t,c),j} + \geq R^{d1}_{(l-1,loc-2^{l-1},(t+1)\%II,c),j}
\end{aligned} \tag{4.57}$$

As level 0 switches communicate with PEs, the enter and exit channel variables define the data going between PEs and level 0 switches, as shown in figure 4.7. As a result, the value propagation constraints for level 0 switches are shown in equations 4.58, 4.59, 4.60, and 4.61.

$$R^{d0}_{(0,loc,t,c),j} + R^{d1}_{(0,loc,t,c),j} + EnterChannel_{(loc*2,c,t)} \geq ExitChannel_{(loc*2+1,c,(t+1)\%II)} \tag{4.58}$$

$$R^{d0}_{(0,loc,t,c),j} + R^{d1}_{(0,loc,t,c),j} + EnterChannel_{(loc*2+1,c,t)} \geq ExitChannel_{(loc*2,c,(t+1)\%II)} \tag{4.59}$$

$$\begin{aligned}
&\textbf{if } loc\%2 = 0 : \\
&\quad R^{d0}_{(0,loc,t,c),j} + EnterChannel_{(loc*2,c,t)} + EnterChannel_{(loc*2+1,c,t)} \geq R^{u1}_{(1,loc+1,(t+1)\%II,c),j} \\
&\textbf{else:} \\
&\quad R^{d0}_{(0,loc,t,c),j} + EnterChannel_{(loc*2,c,t)} + EnterChannel_{(loc*2+1,c,t)} \geq R^{u1}_{(1,loc-1,(t+1)\%II,c),j}
\end{aligned} \tag{4.60}$$

$$R^{d1}_{(0,loc,t,c),j} + EnterChannel_{(loc*2,c,t)} + EnterChannel_{(loc*2+1,c,t)} \geq R^{u0}_{(1,loc,(t+1)\%II,c),j} \tag{4.61}$$

A value must fan-out to at least one routing resource, i.e. information cannot be destroyed. For all $\forall i \in A, \forall c \in C, \forall j \in Vals$ (for $l > 1$),

$$\begin{aligned}
R^{u0}_{(l+1,loc,c,(t+1)\%II),j} + R^{u1}_{(l+1,loc+2^l,c,(t+1)\%II),j} + R^{d0}_{(l-1,loc,c,(t+1)\%II),j} + R^{d1}_{(l-1,loc-2^{l-1},c,(t+1)\%II),j} \\
\geq R^{u0}_{(l,loc,c,t,j)} + R^{u1}_{(l,loc,c,t,j)} + R^{d0}_{(l,loc,c,t,j)} + R^{d1}_{(l,loc,c,t,j)}
\end{aligned} \tag{4.62}$$

Similarly, For level 0, the following equation holds:

$$
\begin{aligned}
&\textbf{if } loc\%2 = 0: \\
&\quad R^{u0}_{(1,loc,c,(t+1)\%II),j} + ExitChannel_{(loc*2,(t+1)\%II,c),j} + \\
&\quad R^{u1}_{(1,loc+1,c,(t+1)\%II),j} + ExitChannel_{(loc*2+1,(t+1)\%II,c),j} \\
&\qquad \geq R^{u0}_{(0,loc,c,t,j)} + EnterChannel_{(loc*2,t,c),j} + \\
&\qquad R^{u1}_{(0,loc,c,t,j)} + EnterChannel_{(loc*2+1,t,c),j} \\
&\textbf{else:} \\
&\quad R^{u0}_{(1,loc,c,(t+1)\%II),j} + ExitChannel_{(loc*2,(t+1)\%II,c),j} + \\
&\quad R^{u1}_{(1,loc-1,c,(t+1)\%II),j} + ExitChannel_{(loc*2+1,(t+1)\%II,c),j} \\
&\qquad \geq R^{u0}_{(0,loc,c,t,j)} + EnterChannel_{(loc*2,t,c),j} \\
&\qquad + R^{u1}_{(0,loc,c,t,j)} + EnterChannel_{(loc*2+1,t,c),j}
\end{aligned}
\tag{4.63}
$$

The blocks located at the highest level take less inputs and do not use the routing resources that would come from outside the array (output nothing and input nothing from above).

Objective Function: Our objective function minimizes the sum of all routing resources used, across every PE and every location.

$$
Minimize \sum_{i}^{A} \sum_{c}^{C} \sum_{j}^{Vals} R^{u0}_{(i,c),j} + R^{u1}_{(i,c),j} + R^{d0}_{(i,c),j} + R^{d1}_{(i,c),j} \tag{4.64}
$$

Scheduling concludes the generation of a repeatable schedule for a fixed size array, which can then be replicated over a chip. Figure 4.8 illustrates a schedule of the motivating example over a torus NoC. Inputs leave the IO PE at $(0, 1)$ and both use the same router and the same channel in different contexts (cycles). Both x and a propagate to the adder PE at $(0, 0)$ and, in x's case, to the multiplier PE at $(1, 1)$, and so on until the final multiply is propagated to the IO PE at $(1, 1)$. This dual-context run ($II = 2$, represented by two close parallel lines) was mapped with $C = 2$ channels which is required because of PE self-communication (e.g. the result of one add is the input to another, and there is one adder).

Table 4.1 compares scheduling complexity over different topologies for a DFG with 15 nodes (Caprasse benchmark) by showing the number of ILP variables, ILP constraints, and runtime. Due to its simpler connectivity, torus has the smallest ILP problem size and runtime, and BFT is hte most complex one due to more complex routing and a larger number of interconnect elements (the multi-layered interconnect tree).

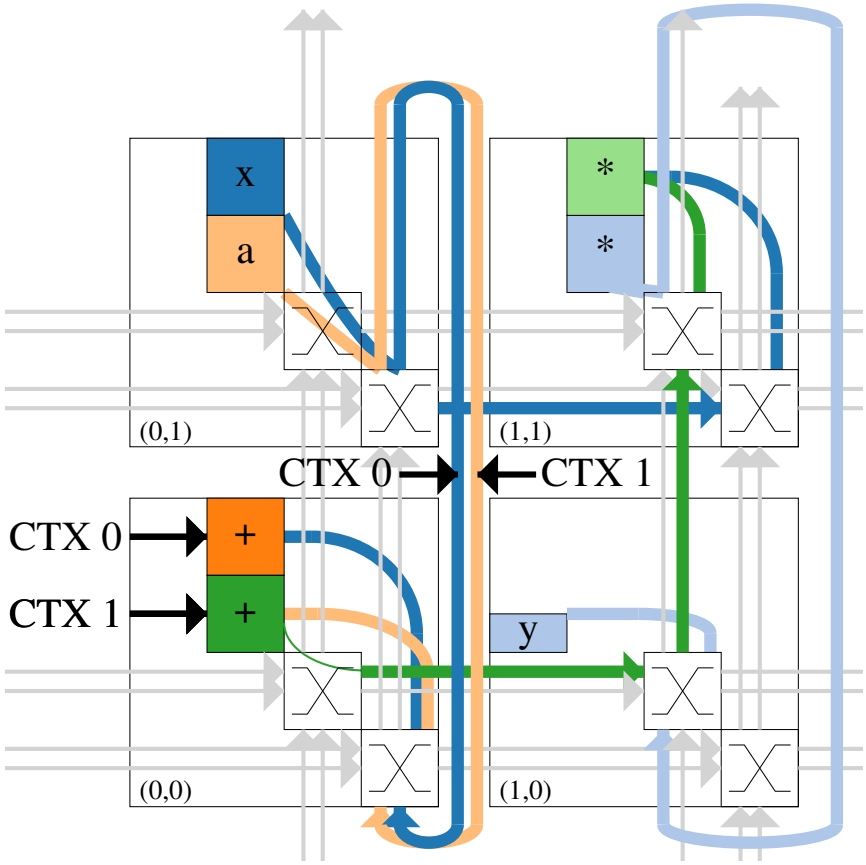

Figure 4.8. CGRA placement and schedule.

TABLE 4.1
SCHEDULING ILP PROBLEM SIZE AND RUNTIME COMPARISON OF DIFFERENT TOPOLOGIES FOR A DFG WITH 15 NODES.

Topology	ILP variables	ILP constraints	Runtime
Torus	2100	4470	0.05s
Mesh	2940	6270	0.05s
BFT	4928	11904	0.13s

4.4.4 Configuration

After an appropriate schedule is computed, we generate a hardware configuration with the final channel count, II, and PE operator arrangement. The static schedule is processed to generate

context memories for all routers, PE input multiplexers, and operand rotating register addresses, which are then fed into the synthesis tool. Here, the array can be unrolled to the greatest extent allowed by the chip-specific implementation (e.g. a 3x19 array can be unrolled 23 times to utilize the entire 19×69 torus overlay).

Chapter 5

Experimental results

In this chapter, we quantify the cost and performance of Mocarabe and compare it with Vivado HLS and other CGRAs. First, we show how many communication channels (C) our scheduler needs to run each benchmark on our Torus, Mesh, and BFT configurations. We then compare CGRA placement results with the Integer Linear Programming (ILP) and Simulated Annealing (SA) formulation in terms of runtime and quality of results. We compare the final scheduled results for different overlay configurations based on performance and resource usage. We also compare our work with contemporary CGRA overlays. We compare our chip-spanning torus overlays with Vivado HLS and show how Mocarabe performs compared to Vivado HLS in terms of performance and resource usage.

5.1 Tools

In this work, We use GNU parallel [38] to run our experiments. We use Vivado 2020.1 and Vivado HLS 2020.1 for hardware synthesis, and use Gurobi solver [12] to solve ILP problems. We use a modified version of the open-source gcc-python plugin to convert a given C code to a DFG representing the code. Gcc-python-plugin is usually used for modifying the gcc GIMPLE backend which allows accessing the internal representations of compiler and performing various analysis. Our modified framework can be found at: https://git.uwaterloo.ca/watcag-public/gcc-python-plugin.

5.2 Benchmarks

To evaluate our work, we use a set of benchmarks from BitGPU benchmarks [53]. Our benchmarks are feedforward dataflow kernels that result in DataFlow Graphs (DFGs) ranging from 10 to more than 100 nodes. Table 5.1 shows an overview of every benchmark's DFG size and the number of I/O, adder, and multiplier nodes they have.

TABLE 5.1
OVERVIEW OF BENCHMARKS.

Benchmark	DFG Size		
	I/Os	Adds	Multiplies
fig3	4	2	1
adder_chain	5	3	0
level1_linear	5	2	2
poly_quadratic	5	2	2
poly3	5	2	3
bellido	4	5	3
approx1	6	4	3
poly4	6	3	4
level1_saturation	7	5	2
caprasse	6	3	6
poly6	8	5	6
poly8	10	7	7
sobel	11	11	4
rgb	15	3	9
poly10	12	9	9
gaussian	19	8	9
poly20	21	19	9
iir8	31	7	14
dct	23	32	22
deriche	49	35	45

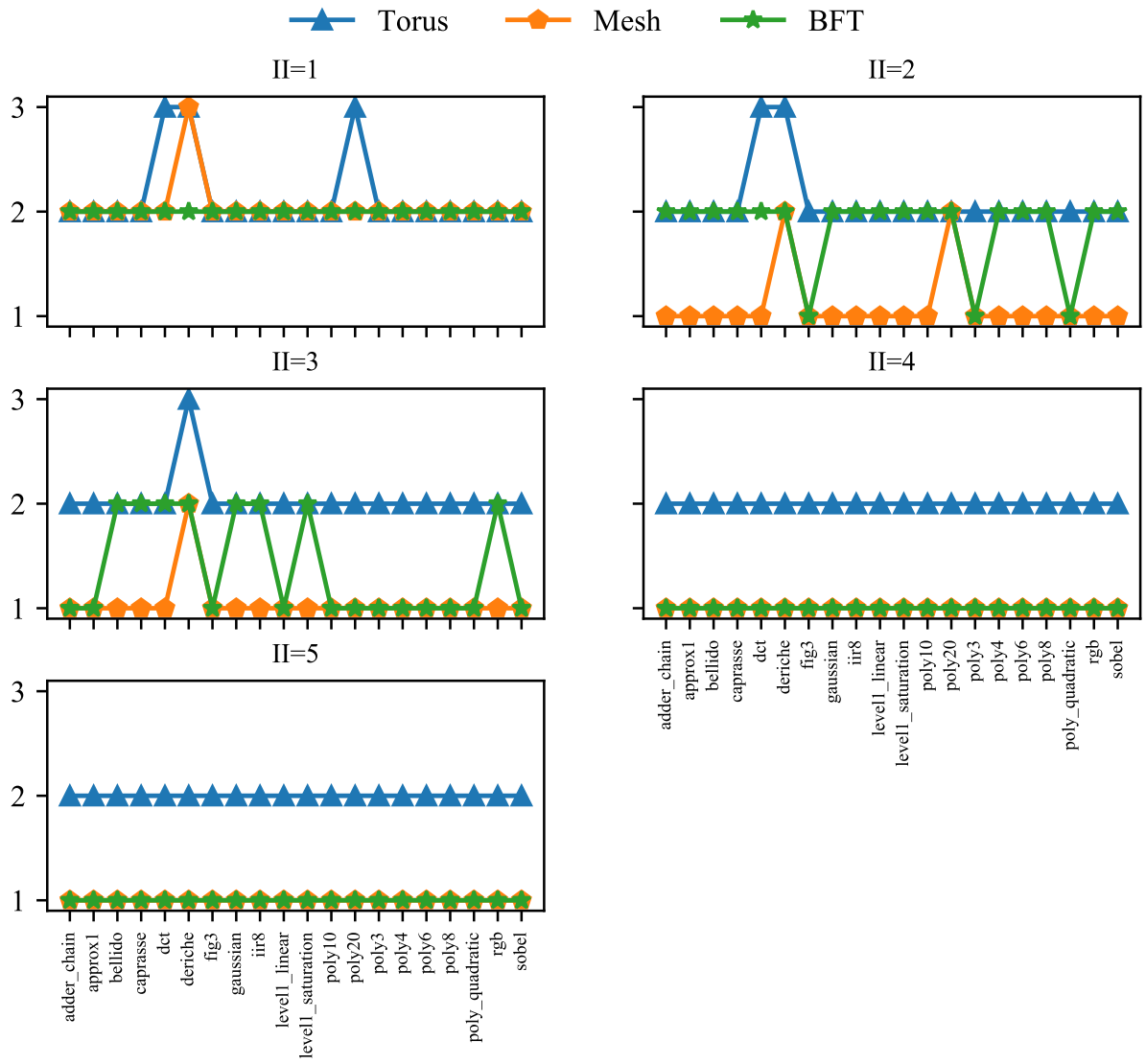

Figure 5.1. Required channels per benchmark for II = 1 to 5.

5.3 Scheduler results

To find out how many communication channels each benchmark needs for running on each NoC topology, we used our compiler to generate overlay schedules for all benchmarks with II = 1 to II = 5 for Torus, Mesh, and BFT topologies. The resulting schedules then can be used to generate

context memories for RTL overlays to run the desired benchmarks. Figure 5.1 shows the required number of communication channels per benchmark for each topology for II = 1 to II = 5. Figure 5.2 shows the mean required channels per II for each topology. As seen in Figures 5.1 and 5.2, as II increases, the number of required channels always decreases since more II means more time and freedom to route data, resulting in fewer required communication channels. Furthermore, Torus always needs more communication channels than BFT or Mesh to run benchmarks. This is due to the more limited routing Torus offers, as data can only go up or right over the array. On the other hand, as Mesh and BFT provide higher degrees of freedom for data movement, they need less communication channels.

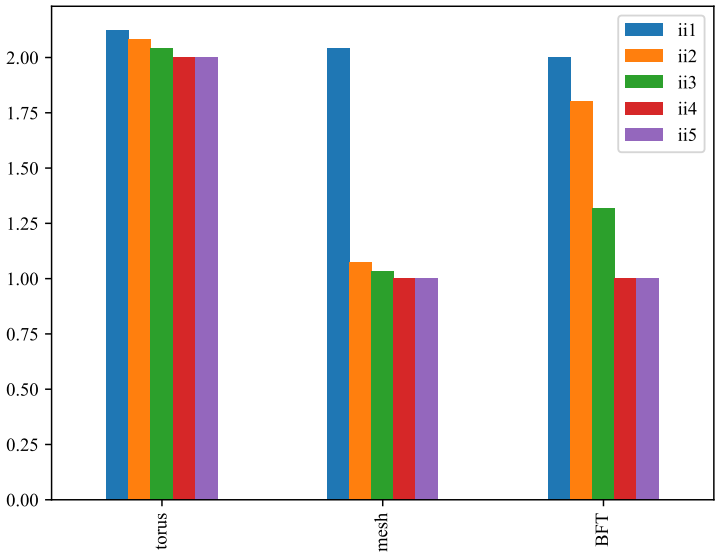

Figure 5.2. Mean required channels per benchmark for II = 1 to 5.

5.4 Comparing ILP and SA placer

Although our topology-agnostic ILP placer finds an optimal placement, it strictly performs equal to or better than SA placer,but can take longer than SA to finish. Moreover, SA could find placements very close to the optimal version. To compare our SA and ILP placers and find out if the ILP placer is always the best choice, we used our compiler to perform placement using SA and ILP placers for all benchmarks over mesh, torus, and BFT topologies. Table 5.2 compares ILP and SA placers by providing ILP vs SA placement cost (quadratic wirelength) ratio and ILP placer runtime. SA finds placement for all benchmarks in less than 30 seconds, and we

give ILP placer 13 hours for each benchmark, which times out for `sobel, deriche, dct, poly20,` and `iir8` benchmarks. As shown in the table, ILP can improve placement quadratic wirelength up to 37%, which makes it a good candidate for applications where a high quality solution is preferred over runtime.

TABLE 5.2
ILP VS SA PLACEMENT QUALITY AND RUNTIME
(SA FINDS PLACEMENT FOR ALL BENCHMARKS IN LESS THAN 30 SECONDS).

Benchmark	ILP vs SA W^2 ratio			Average ILP runtime
	torus	mesh	BFT	
fig3	1	1	1	0.81s
adder_chain	1	1	1	0.57s
level1_lin	1	1	0.93	1.7s
poly_quad	1	1	1	1.57s
poly3	1	1	1	2.25s
bellido	1	1	0.93	9.66s
approx1	1	1	0.86	5.6s
poly4	1	1	0.94	7.03s
level1_sat	1	0.97	0.94	0.46s
caprasse	0.92	0.9	1	22s
poly6	0.92	0.84	1	11.43s
poly8	0.88	0.76	0.93	13.67s
rgb	0.8	0.9	0.86	1h10m
poly10	0.87	0.80	0.91	5s
gaussian	0.63	0.71	0.85	12h29m

5.5 CGRA overlay floorplanning Fmax

To compare the cost and performance of different Mocarabe overlay configurations, and compare our work with recent CGRA overlays, we implemented Mocarabe on a Xilinx Alveo U280 using the floorplanning described in chapter 3 to carefully optimize the design. Mocarabe is available in seven different configurations, which are shown in table 5.3 (the 4×4 torus is for comparison with other CGRA overlays).

To demonstrate high operation frequencies across a spectrum of operator configurations (multipliers and adders), each CGRA configuration was tested with 10–100% of PEs as multipliers

TABLE 5.3
OVERLAY CONFIGURATIONS AND THEIR AVERAGE MAXIMUM FREQUENCY

Topology	C	PE inputs	Array Size	Average Fmax (MHz)
Torus	1	1	19×69	748
Torus	2	2	19×69	717
Torus	3	2	19×69	675
Torus	1	1	4×4	864
Mesh	1	1	19×22	801
Mesh	2	2	19×22	809
BFT	1	1	256 PEs	670

(in steps of 10%), randomly distributed across the array, with the remaining PEs configured as adders. Figure 5.3 shows the operation frequencies of different configurations and figure 5.4 compares different topologies in term of frequency per LUT and FF usage (as both the mesh and BFT overlays only span SLR0, we divided torus resource usage by 3 to have a sensible comparison as torus spans all 3 available SLRs and uses almost the same amount of resources on each SLR). There are small frequency variations between different PE configurations, and Mocarabe maintains high operating frequencies regardless of the PE configurations because of Pblock sizing and adequate pipelining.

TABLE 5.4
CGRA SIZES AND FREQUENCIES

CGRA	C	Frequency (MHz)	Array size
Torus Mocarabe	1	711–768	19×69
Torus Mocarabe	2	691–750	19×69
Torus Mocarabe	3	645–693	19×69
Torus Mocarabe	1	813–921	4×4
ADRES (Ultrascale+)	1	382	4×4
ADRES (Stratix10)	1	260	4×4
HyCUBE (Ultrascale+)	1	307	4×4
HyCUBE (Stratix 10)	1	226	4×4

We compare the chip-spanning torus Mocarabe to recent CGRA-ME [39] overlay implementations of ADRES [24] and HyCUBE [14]. In [39], the authors implement 4×4 ADRES and HyCUBE CGRAs as overlays on the Xilinx Ultrascale+ XCVU3P and the Intel Stratix 10

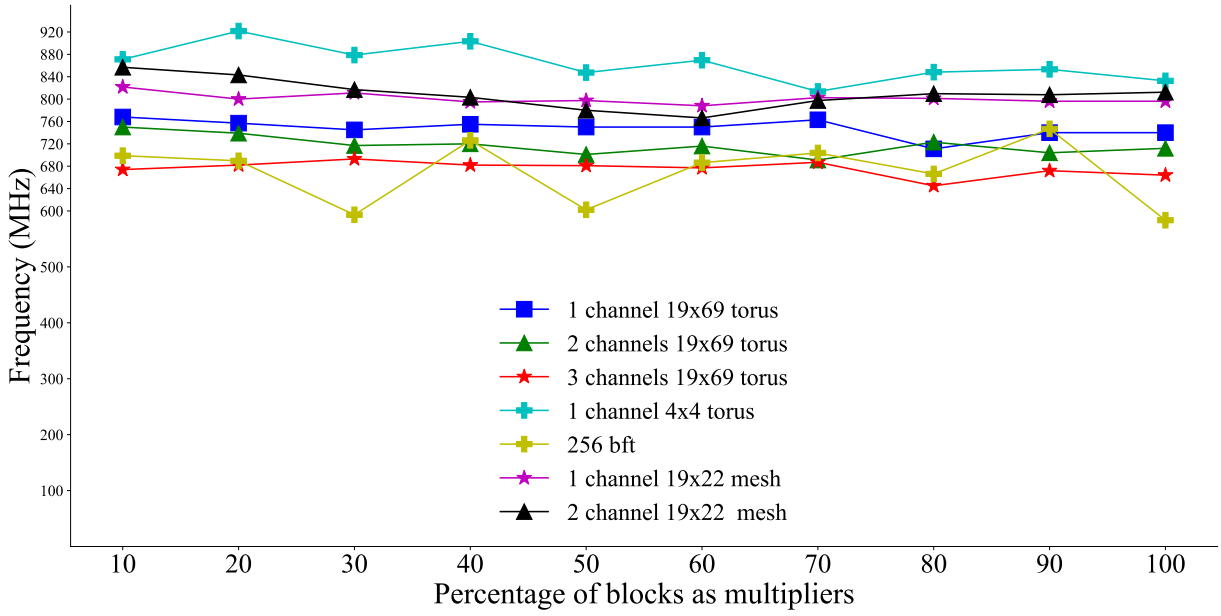

Figure 5.3. Frequency for different channel-count and PE configurations.

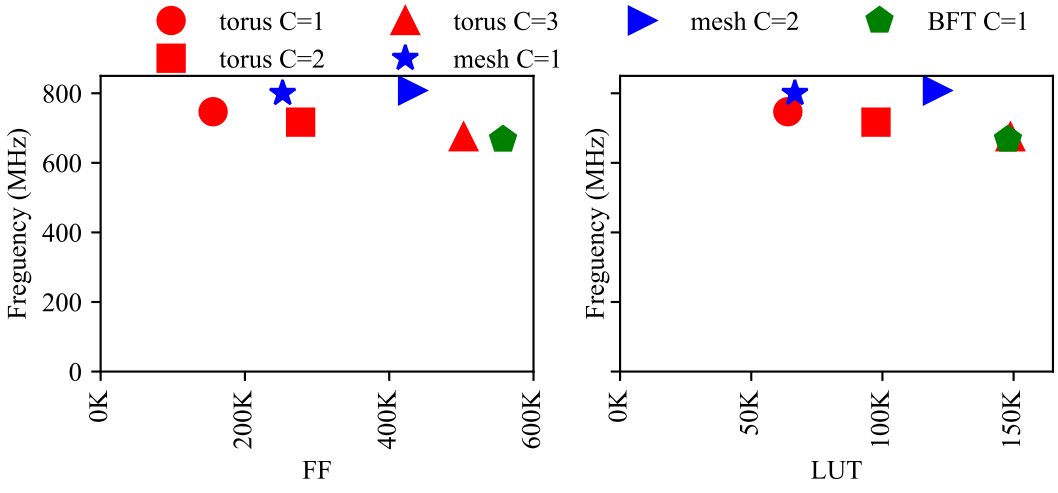

Figure 5.4. LUT and FF usage vs frequency for different channel-count and topologies.

TABLE 5.5
MOCARABE FREQUENCY GAINS OVER OTHER CGRAS

CGRA	C=1	C=2	C=3	C=1 (4×4)
ADRES (U+)	$2\times$	$2\times$	$1.8\times$	$2.4\times$
ADRES (S10)	$3\times$	$2.9\times$	$2.7\times$	$3.5\times$
HyCUBE (U+)	$2.5\times$	$2.4\times$	$2.3\times$	$3\times$
HyCUBE (S10)	$3.4\times$	$3.3\times$	$3.1\times$	$4\times$

GX850. The 32-bit ADRES overlay operates at up to 382 MHz on average on the Ultrascale+ and up to 260MHz on average on the Stratix 10. The 32-bit HyCUBE overlay operates at up to 307 MHz on average on the Ultrascale+ and up to 226 MHz on average on the Stratix 10.

Torus Mocarabe maintains higher operation frequencies while having orders-of-magnitude larger array sizes. In Table 5.4, we show operating frequencies for different CGRAs. For torus Mocarabe we provide frequency ranges, as frequency varies in different operator configurations. In Table 5.5, we show the clock frequency gains for different torus Mocarabe configurations when compared to ADRES and HyCUBE overlays.

5.6 Mocarabe vs Vivado HLS

To evaluate our framework as an HLS tool, we scale up the Mesh array to span the entire chip (up to 19×69, similar to Torus), and compare our chip-spanning Torus and Mesh overlays with Vivado HLS. To do so, we implement each benchmark with a given initiation interval (II) and unroll the resulting array to take advantage of every available PE (19×69). We give Vivado HLS the same benchmark, II, unroll factor and frequency target. Though Vivado provides the option to constrain the number of adders and multipliers, we outperform Vivado without providing these constraints, as shown in Figure 5.5. Vivado HLS can, in some cases, retain a high f_{max} for benchmarks with no resource sharing ($II = 1$), but this quickly drops off by factors of up to $2\times$ for $II = 2$, $4.5\times$ for $II = 3$, and $5.5\times$ for $II = 4$ as more operations are assigned to one functional unit. At $II = 5$, the lowest f_{max} achieved by Vivado HLS is with the `poly10` at 78.06 MHz, $9\times$ away from Mocarabe peak at 750 MHz. These results do not apply any resource constraints to Vivado HLS.

For all benchmarks, Mocarabe has met the target II, requiring the number of communication channels shown in figure 5.1. Vivado HLS can also meet target II when not given any resource constraints. However, when we count the number of functional units we use and force Vivado

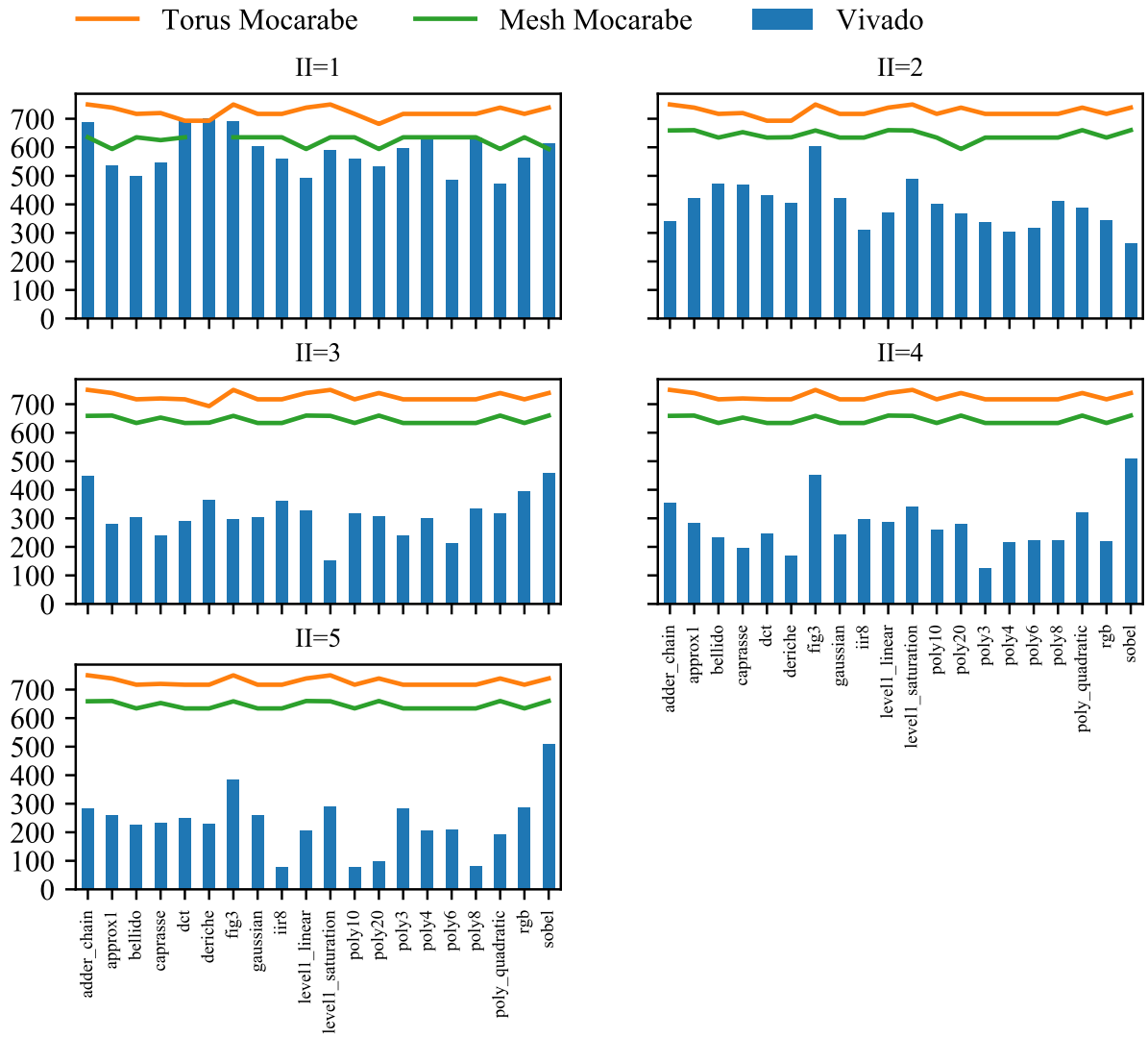

Figure 5.5. Vivado HLS f_{max} vs. Torus and Mesh Mocarabe. (`deriche` at II = 1 needs C=3 for Mesh that is not supported)

HLS to use the same, Vivado's resource sharing simply fails beyond $II = 1$, as shown in Figure 5.6. II can increase by up to 2 cycles for $II = 4$, and up to 4 for $II = 5$.

Tables 5.6 and 5.7 compare Vivado HLS resource usage to Torus and Mesh Mocarabe resource usage. Torus and Mesh Mocarabe use a similar number of DSP blocks in most cases

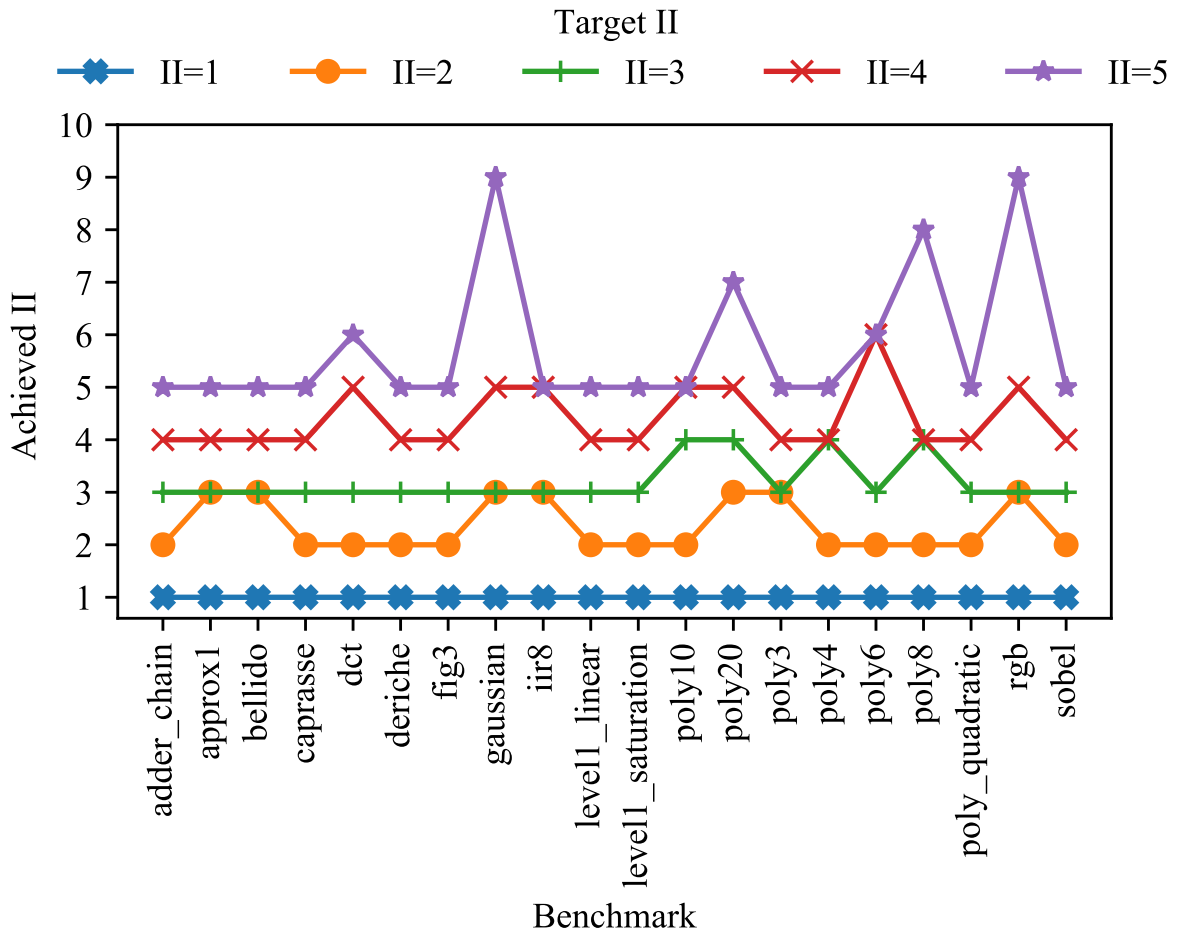

Figure 5.6. Achieved II vs. Target II for various dataflow benchmarks in Vivado HLS. Mocarabe always meets target II.

while using more LUTs, which is in part due to the communication network. Furthermore, as II increases, Mesh Mocarabe uses less resources compared to Torus Mocarabe, as it requires fewer communication channels to run the benchmarks. Vivado HLS failed to implement `dct`, `poly10`, and `poly20` for targeting $II = 5$ with the same unroll as us. Implementation failures are indicated by "DNF" in the table. Tables 5.8 and 5.9 compare Torus and Mesh Mocarabe latency in cycles to Vivado HLS latency. Torus Mocarabe latency can be up to 6.8× (3.1 - 3.4× mean), and Mesh Mocarabe latency can be up to 6.8× (2.8 - 3.2× mean) larger than Vivado HLS depending on the number of communication channels. This is because the PE takes 6-8 cycles

and the router takes 2-3 cycles for high-frequency pipelined operation.

TABLE 5.6
TORUS MOCARABE VS VIVADO HLS RESOURCE USAGE
(TORUS MOCARABE/VIVADO HLS)

Benchmark	II=1		II=2		II=3		II=4		II=5	
	DSP	LUT	DSP	LUT	DSP	LUT	DSP	LUT	DSP	LUT
adder_chain	–	22.6	–	18.8	–	18.6	–	5.3	–	4.9
approx1	1.0	23.4	1.0	9.4	1.0	7.5	1.0	10.6	1.0	7.2
bellido	1.0	46.3	0.9	16.1	1.0	7.9	1.0	9.2	1.0	9.5
caprasse	1.5	21.2	1.0	6.7	1.0	5.1	1.5	9.3	1.5	7.4
dct	1.0	47.9	1.0	13.8	1.0	6.8	1.0	4.0	DNF	DNF
deriche	1.4	58.5	1.0	19.0	1.0	9.2	1.0	8.8	1.0	6.5
fig3	1.0	20.4	1.0	16.1	1.0	12.0	1.0	6.0	1.0	6.6
gaussian	1.0	31.9	1.0	10.5	1.0	7.8	1.0	4.1	1.0	3.2
iir8	0.9	12.7	1.0	10.8	1.0	7.3	1.0	3.9	1.0	3.5
level1_lin	1.0	25.7	1.0	8.0	1.0	9.7	1.0	6.7	1.0	6.4
level1_sat	1.0	21.3	1.0	9.3	1.0	5.1	1.5	10.5	1.5	9.4
poly10	0.9	19.4	1.0	5.6	1.0	4.3	1.0	2.9	DNF	DNF
poly20	0.9	24.3	1.0	5.1	1.0	4.7	1.0	3.0	DNF	DNF
poly3	1.0	16.7	1.0	6.9	1.0	8.2	1.0	7.1	1.0	8.2
poly4	1.1	33.8	1.0	4.3	1.0	6.9	1.1	12.3	1.1	7.8
poly6	1.0	25.0	1.0	5.4	1.0	3.8	1.0	4.0	1.0	4.0
poly8	0.9	20	1.0	4.8	1.0	4.0	1.0	2.6	1.0	2.8
poly_quad	1.0	85.4	1.0	13.3	1.0	37.4	1.0	8.0	1.0	8.2
rgb	1.0	28.2	1.0	11.3	1.0	10.6	1.0	4.7	1.0	7.0
sobel	4.0	32.6	2.0	14.9	1.0	14.1	1.0	12.5	1.0	18.8

TABLE 5.7
MESH MOCARABE VS VIVADO HLS RESOURCE USAGE
(MESH MOCARABE/VIVADO HLS)

Benchmark	II1		II2		II3		II4		II5	
	DSP	LUT	DSP	LUT	DSP	LUT	DSP	LUT	DSP	LUT
adder_chain	0.0	27.0	0.0	12.2	0.0	12.1	0.0	3.4	0.0	3.2
approx1	1.0	28.1	1.0	5.8	1.0	4.7	1.0	4.4	1.0	3.0
bellido	1.0	55.7	0.9	10.0	1.0	4.9	1.0	5.7	1.0	4.0
caprasse	1.5	25.6	1.0	4.1	1.0	3.1	1.5	5.6	1.5	4.5
dct	1.0	38.4	1.0	5.6	1.0	4.1	1.0	2.4	DNF	DNF
deriche	DNF	DNF	1.0	15.3	1.0	7.4	1.0	3.2	1.0	2.6
fig3	1.0	24.4	1.0	10.4	1.0	7.8	1.0	3.9	1.0	4.3
gaussian	1.0	38.4	1.0	6.3	1.0	4.7	1.0	2.5	1.0	2.0
iir8	0.9	15.2	1.0	6.6	1.0	4.4	1.0	2.4	1.0	2.1
level1_linear	1.0	30.9	1.0	4.9	1.0	6.0	1.0	4.2	1.0	4.0
level1_saturation	1.0	25.5	1.0	6.0	1.0	3.3	1.5	4.6	1.5	4.2
poly10	0.9	23.3	1.0	3.4	1.0	2.6	1.0	1.7	DNF	DNF
poly20	0.9	19.5	1.0	6.1	1.0	2.9	1.0	1.8	DNF	DNF
poly3	1.0	20.0	1.0	4.2	1.0	5.0	1.0	4.3	1.0	3.3
poly4	1.1	40.7	1.0	2.6	1.0	4.2	1.1	7.4	1.1	3.1
poly6	1.0	30.0	1.0	3.3	1.0	2.3	1.0	2.4	1.0	2.4
poly8	0.9	24.1	1.0	2.9	1.0	2.4	1.0	1.6	1.0	1.7
poly_quadratic	1.0	102.8	1.0	8.2	1.0	23.2	1.0	5.0	1.0	5.1
rgb	1.0	33.9	1.0	6.8	1.0	6.4	1.0	2.8	1.0	2.8
sobel	4.0	39.2	2.0	9.2	1.0	8.7	1.0	7.8	1.0	7.9

TABLE 5.8
TORUS MOCARABE VS VIVADO HLS LATENCY IN CYCLES
(MOCARABE/VIVADO HLS RATIO IS SHOWN IN RED)

Benchmark	II=1		II=2		II=3		II=4		II=5	
	Moc	HLS	Moc	HLS	Moc	HLS	Moc	HLS	Moc	HLS
adder_chain	63.0	10.0 (6.3)	64.0	10.0 (6.4)	66.0	13.0 (5.1)	68.0	10.0 (6.8)	70.0	11.0 (6.4)
approx1	93.0	30.0 (3.1)	94.0	34.0 (2.8)	99.0	28.0 (3.5)	100.0	31.0 (3.2)	105.0	29.0 (3.6)
bellido	93.0	23.0 (4.0)	96.0	25.0 (3.8)	102.0	22.0 (4.6)	104.0	20.0 (5.2)	105.0	22.0 (4.8)
caprasse	109.0	30.0 (3.6)	108.0	27.0 (4.0)	114.0	28.0 (4.1)	112.0	31.0 (3.6)	135.0	29.0 (4.7)
dct	152.0	98.0 (1.6)	128.0	88.0 (1.5)	132.0	82.0 (1.6)	144.0	89.0 (1.6)	140.0	87.0 (1.6)
deriche	647.0	212.0 (3.1)	666.0	186.0 (3.6)	498.0	155.0 (3.2)	492.0	172.0 (2.9)	505.0	172.0 (2.9)
fig3	63.0	17.0 (3.7)	66.0	17.0 (3.9)	63.0	17.0 (3.7)	72.0	17.0 (4.2)	70.0	18.0 (3.9)
gaussian	165.0	45.0 (3.7)	166.0	58.0 (2.9)	180.0	43.0 (4.2)	192.0	45.0 (4.3)	195.0	56.0 (3.5)
iir8	243.0	119.0 (2.0)	268.0	166.0 (1.6)	261.0	136.0 (1.9)	272.0	140.0 (1.9)	295.0	131.0 (2.3)
level1_lin	75.0	25.0 (3.0)	76.0	24.0 (3.2)	78.0	24.0 (3.2)	76.0	24.0 (3.2)	90.0	25.0 (3.6)
level1_sat	85.0	31.0 (2.7)	86.0	28.0 (3.1)	90.0	30.0 (3.0)	96.0	32.0 (3.0)	95.0	30.0 (3.2)
poly_quad	77.0	25.0 (3.1)	80.0	25.0 (3.2)	81.0	24.0 (3.4)	84.0	24.0 (3.5)	85.0	25.0 (3.4)
poly10	293.0	104.0 (2.8)	324.0	106.0 (3.1)	333.0	108.0 (3.1)	344.0	101.0 (3.4)	350.0	98.0 (3.6)
poly20	1025.0	206.0 (5.0)	1108.0	238.0 (4.7)	987.0	233.0 (4.2)	1020.0	214.0 (4.8)	975.0	210.0 (4.6)
poly3	91.0	32.0 (2.8)	96.0	34.0 (2.8)	96.0	31.0 (3.1)	96.0	33.0 (2.9)	100.0	32.0 (3.1)
poly4	121.0	44.0 (2.8)	130.0	40.0 (3.2)	138.0	48.0 (2.9)	140.0	41.0 (3.4)	140.0	40.0 (3.5)
poly6	189.0	64.0 (3.0)	188.0	62.0 (3.0)	198.0	59.0 (3.4)	204.0	68.0 (3.0)	200.0	68.0 (2.9)
poly8	247.0	83.0 (3.0)	246.0	82.0 (3.0)	252.0	84.0 (3.0)	252.0	80.0 (3.1)	280.0	96.0 (2.9)
rgb	59.0	33.0 (1.8)	60.0	47.0 (1.3)	60.0	36.0 (1.7)	64.0	34.0 (1.9)	70.0	54.0 (1.3)
sobel	149.0	32.0 (4.7)	156.0	32.0 (4.9)	159.0	30.0 (5.3)	164.0	29.0 (5.7)	165.0	33.0 (5.0)
geomean		3.1		3.1		3.3		3.4		3.3

TABLE 5.9
MESH MOCARABE VS VIVADO HLS LATENCY IN CYCLES
(MOCARABE/VIVADO HLS RATIO IS SHOWN IN RED)

Benchmark	II=1		II=2		II=3		II=4		II=5	
	Moc	HLS	Moc	HLS	Moc	HLS	Moc	HLS	Moc	HLS
adder_chain	63.0	10.0 (6.3)	64.0	10.0 (6.4)	66.0	13.0 (5.1)	68.0	10.0 (6.8)	75.0	11.0 (6.8)
approx1	91.0	30.0 (3.0)	94.0	34.0 (2.8)	96.0	28.0 (3.4)	100.0	31.0 (3.2)	110.0	29.0 (3.8)
bellido	93.0	23.0 (4.0)	96.0	25.0 (3.8)	99.0	22.0 (4.5)	96.0	20.0 (4.8)	105.0	22.0 (4.8)
caprasse	111.0	30.0 (3.7)	112.0	27.0 (4.1)	114.0	28.0 (4.1)	124.0	31.0 (4.0)	120.0	29.0 (4.1)
dct	111.0	98.0 (1.1)	112.0	88.0 (1.3)	114.0	82.0 (1.4)	124.0	89.0 (1.4)	125.0	87.0 (1.4)
deriche	DNF	212.0 (DNF)	336.0	186.0 (1.8)	360.0	155.0 (2.3)	400.0	172.0 (2.3)	355.0	172.0 (2.1)
fig3	61.0	17.0 (3.6)	64.0	17.0 (3.8)	66.0	17.0 (3.9)	68.0	17.0 (4.0)	75.0	18.0 (4.2)
gaussian	151.0	45.0 (3.4)	152.0	58.0 (2.6)	159.0	43.0 (3.7)	172.0	45.0 (3.8)	175.0	56.0 (3.1)
iir8	235.0	119.0 (2.0)	238.0	166.0 (1.4)	240.0	136.0 (1.8)	252.0	140.0 (1.8)	270.0	131.0 (2.1)
level1_lin	77.0	25.0 (3.1)	78.0	24.0 (3.2)	84.0	24.0 (3.5)	84.0	24.0 (3.5)	90.0	25.0 (3.6)
level1_sat	81.0	31.0 (2.6)	84.0	28.0 (3.0)	90.0	30.0 (3.0)	92.0	32.0 (2.9)	90.0	30.0 (3.0)
poly_quad	77.0	25.0 (3.1)	80.0	25.0 (3.2)	81.0	24.0 (3.4)	80.0	24.0 (3.3)	95.0	25.0 (3.8)
poly10	285.0	104.0 (2.7)	294.0	106.0 (2.8)	303.0	108.0 (2.8)	316.0	101.0 (3.1)	320.0	98.0 (3.3)
poly20	743.0	206.0 (3.6)	686.0	238.0 (2.9)	699.0	233.0 (3.0)	716.0	214.0 (3.3)	745.0	210.0 (3.5)
poly3	89.0	32.0 (2.8)	90.0	34.0 (2.6)	96.0	31.0 (3.1)	100.0	33.0 (3.0)	100.0	32.0 (3.1)
poly4	121.0	44.0 (2.8)	122.0	40.0 (3.0)	129.0	48.0 (2.7)	136.0	41.0 (3.3)	140.0	40.0 (3.5)
poly6	177.0	64.0 (2.8)	184.0	62.0 (3.0)	192.0	59.0 (3.3)	196.0	68.0 (2.9)	195.0	68.0 (2.9)
poly8	227.0	83.0 (2.7)	230.0	82.0 (2.8)	237.0	84.0 (2.8)	244.0	80.0 (3.0)	250.0	96.0 (2.6)
rgb	53.0	33.0 (1.6)	56.0	47.0 (1.2)	57.0	36.0 (1.6)	56.0	34.0 (1.6)	65.0	54.0 (1.2)
sobel	133.0	32.0 (4.2)	140.0	32.0 (4.4)	144.0	30.0 (4.8)	148.0	29.0 (5.1)	155.0	33.0 (4.7)
geomean		2.9		2.8		3.0		3.2		3.1

5.6.1 Torus vs Mesh latency per area

Using the data we got by running the benchmarks, we compare Torus and Mesh Mocarabe in terms of latency per area (LUTs used), as shown in Figures 5.7 and 5.8. For each topology/II combination, Figure 5.7 shows average latency per area over all benchmarks, and Figure 5.8 shows latency per area for each benchmark. To make figures easier to read, we have normalized data per benchmark. As the figures show, Mesh has less latency per area compared to Torus, making it a suitable candidate for applications where a smaller chip area with low latency is desirable.

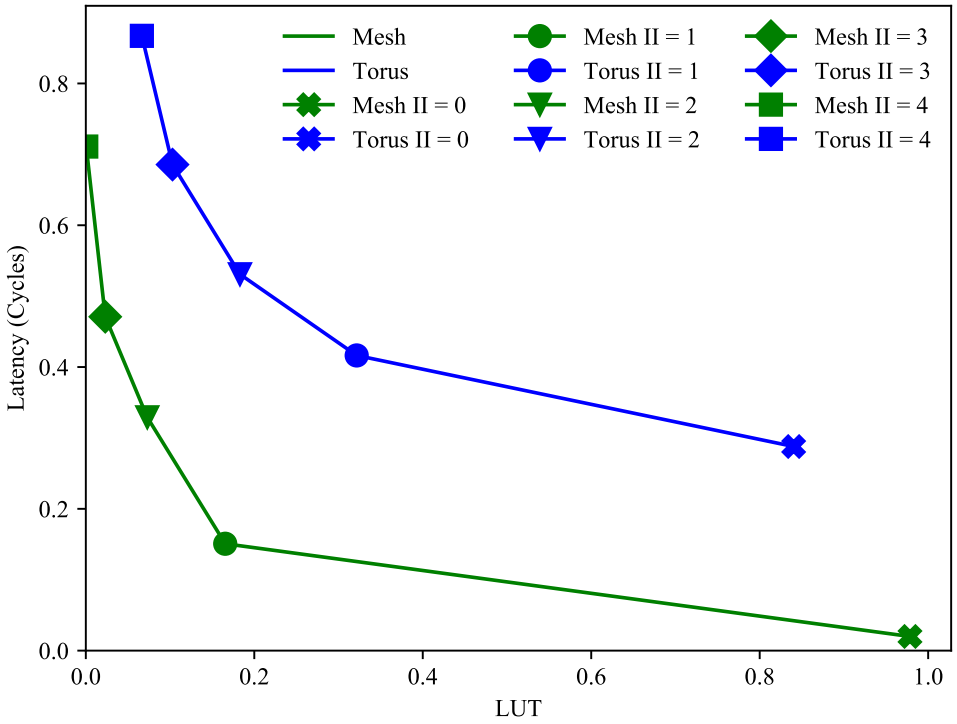

Figure 5.7. Torus vs Mesh average latency per area (LUTs used).

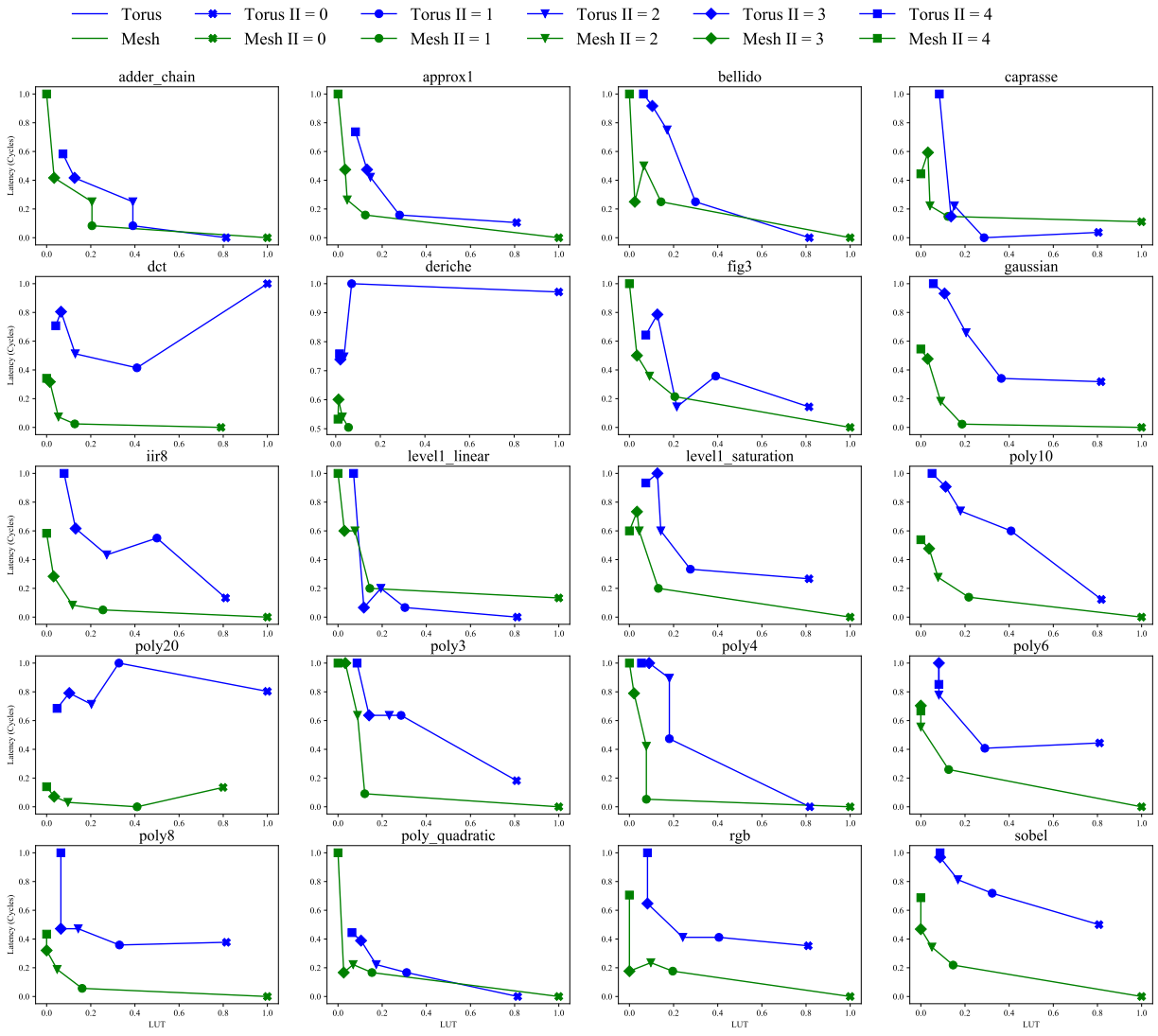

Figure 5.8. Torus vs Mesh latency per area (LUTs used) for each benchmark.

Chapter 6

Conclusion and future research

As Moore's scaling is coming to an end, various alternative gardware architectures are being used to accelerate applications and provide better computation capabilities. Custom ASICs provide high performance with no reconfigurability and suffer from high development time and manufacturing costs. FPGAs, on the other hand, provide rapid deployment and fine-grained reconfigurability but suffer from low performance and their performance varies greatly on the given application they implement. CGRAs are becoming a competitive computation candidate as an alternative to ASICs and FPGAs by providing better and more predictable performance than FPGAs while still providing some coarse-grained reconfigurability in terms of interconnect and operations configurations. In this work we introduce Mocarabe, a new CGRA overlay architecture for Xilinx FPGAs and its associated communication-aware compiler.

We propose a novel topology-agnostic ILP placer that reduces placement cost up to 37% and optimizes placement even for non-linear cost functions (like quadratic wirelength) by pre-computing costs and feeding them as an input argument to the ILP problem, at the cost of increasing runtime from less than a minute up to hours and timing out for larger benchmarks. We present a scalable and flexible FPGA overlay implementation with torus, mesh and BFT NoC support engineered with generous pipelining and careful floorplanning to achieve 650MHz+ torus, 766MHz+ mesh, and 583MHz+ BFT operation. Our chip-spanning torus implementation outperforms other CGRA overlay implementations on FPGAs with 1.8–3$\times$ higher frequency. Furthermore, to evaluate our framework as an HLS platform, we compare our work with Vivado HLS. At scale, our torus overlay outperforms Vivado HLS with up to 9.2$\times$ higher frequency. We can share functional resources very effectively and provide efficient communication between functional units, while HLS struggles to do so at higher II. Our CGRA maintains it's high performance between different configurations with small frequency variations, which makes it a suitable HLS platform

as the performance varies very little between different applications. We compare Mesh and Torus topologies, and show Mesh has less latency per area compared to Torus.

6.1 Future reseach

Here we present the possible future work and research ideas based on the work done in this thesis:

- We can add support for branches and conditional execution to add support for more applications. We can use an idea similar to [15] by mapping multiple operations to each PE and decide which one to execute at runtime using the added control lines. We can do so by having multiple context memories instead of one per NoC output, PE operand selection, etc. and select which context memory to use at runtime by adding multiplexers and control lines. Doing so will also require updating the compiler infrastructure as the GCC-python plugin currently does not support conditional execution.
- We can have larger BFT overlays with more communication channels by using all SLRs available on the chip. That would require more elaborate floorplanning and possibly a different number of pipeline registers per each level, as higher levels and SLR-crossing levels would need more pipelining to achieve high performance.
- We can consider accelerating FPGA placement by leveraging tools such as RapidWright [22] and exploiting the regular structure of out architecture (as in [54]) may eliminate the very painful task of finding an optimal floorplan for each target device and speed up the end-to-end flow.
- We can Break up the DFG into separate but adjacent arrays, when possible, would allow for even larger applications to be easily mapped. Coalescing different operations (e.g. sequential multiply and add) into one PE could reduce communication between PEs.

References

[1] A. Adriahantenaina, H. Charlery, A. Greiner, L. Mortiez, and C.A. Zeferino. Spin: a scalable, packet switched, on-chip micro-network. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 70–73 suppl., 2003.

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[3] Gerald G. Brown and Robert F. Dell. Formulating integer linear programs: A rogues' gallery. *INFORMS Transactions on Education*, 7(2):153–159, 2007.

[4] Anupam Chattopadhyay. Ingredients of adaptability: A survey of reconfigurable processors. *VLSI Design*, 2013, 07 2013.

[5] Raghunandan Chaware, Ganesh Hariharan, Jeff Lin, Inderjit Singh, Glenn O'Rourke, Kenny Ng, S. Y. Pai, Chien-Chen. Li, Zill Huang, and S. K. Cheng. Assembly challenges in developing 3d ic package with ultra high yield and high reliability. In *2015 IEEE 65th Electronic Components and Technology Conference (ECTC)*, pages 1447–1451, 2015.

[6] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, New York, NY, USA, 2014. Association for Computing Machinery.

[7] S. Alexander Chin and Jason H. Anderson. An architecture-agnostic integer linear programming approach to cgra mapping. In *DAC '18*, San Francisco, CA, 2018. ACM.

[8] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. Cgra-me: A unified framework for cgra modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–189, Seattle, WA, 2017.

[9] J. Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 433–438, 2006.

[10] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 684–689, 2001.

[11] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *Micro, IEEE*, 32:38–51, 09 2012.

[12] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2020.

[13] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. Sda: Software-defined accelerator for large-scale dnn systems. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–23, 2014.

[14] Manupa Karunaratne, Aditi Mohite, Tulika Mitra, and Li-Shiuan Peh. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. pages 1–6, 06 2017.

[15] Manupa Karunaratne, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 4d-cgra: Introducing branch dimension to spatio-temporal application mapping on cgras. pages 1–8, 11 2019.

[16] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3), September 2014.

[17] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. *SIGPLAN Not.*, 53(4):296–311, June 2018.

[18] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 296–311, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Dorothy Kucar, Shawki Areibi, and Anthony Vannelli. Hypergraph partitioning techniques. *DYNAMICS OF CONTINUOUS DISCRETE AND IMPULSIVE SYSTEMS SERIES A 11*, 2004.

[20] Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti Forsell, Mikael Millberg, Johnny Öberg, Kari Tiensyrjä, and Ahmded Hemani. A network on chip architecture and design methodology. In *Proceedings IEEE Computer Society Annual Symposium on VLSI*, pages 117–124, United States, 2002. IEEE Institute of Electrical and Electronic Engineers. Project code: NOCARC; 2002 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2002 ; Conference date: 25-04-2002 Through 26-04-2002.

[21] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[22] C. Lavin and A. Kaviani. Rapidwright: Enabling custom crafted implementations for fpgas. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 133–140, 2018.

[23] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Comput. Surv.*, 52(6), October 2019.

[24] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Haris Man, and Rudy Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. pages 61–70, 09 2003.

[25] Mikael Millberg, Erland Nilsson, Rikard Thid, Shashi Kumar, and Axel Jantsch. The nostrum backbone - a communication protocol stack for networks on chip. volume 17, pages 693–696, 01 2004.

[26] K. Niu and J. H. Anderson. Compact area and performance modelling for cgra architecture evaluation. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 126–133, 2018.

[27] Partha Pratim Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, 54(8):1025–1040, 2005.

[28] Ishwar Parulkar, Alan Wood, Sun Microsystems, James C. Hoe, Babak Falsafi, Sarita V. Adve, and Josep Torrellas. Opensparc: An open platform for hardware reliability experimentation. In *In Proc.ofthe Workshop on Silicon Errorsin Logic - System Effects*, 2008.

[29] Kunjan Patel, Séamas McGettrick, and Chris J. Bleakley. Syscore: A coarse grained reconfigurable array architecture for low energy biosignal processing. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '11, page 109–112, USA, 2011. IEEE Computer Society.

[30] S. B. Patil, T. Liu, and R. Tessier. A bandwidth-optimized routing algorithm for hybrid fpga networks-on-chip. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 25–28, 2018.

[31] Matthew Perry. simanneal: Python module for simulated annealing, 2019.

[32] A. Podobas, K. Sano, and S. Matsuoka. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743, 2020.

[33] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402, 2017.

[34] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, 2015.

[35] A. Ritz, B. Avent, A. Pratapa, and T. Murali. halp: Hypergraph algorithms package, 2018.

[36] Kirk Saban. *Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency (WP380 (v1.2))*, December 2011.

[37] Hayden Kwok-Hay So and Cheng Liu. *FPGA Overlays*, pages 285–305. Springer International Publishing, Cham, 2016.

[38] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[39] I. Taras and J. H. Anderson. Impact of fpga architecture on area and performance of cgra overlays. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 87–95, 2019.

[40] Ian Taras and Jason H. Anderson. Impact of fpga architecture on area and performance of cgra overlays. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 87–95, 2019.

[41] Vaishali Tehre and Ravindra Kshirsagar. Survey on coarse grained reconfigurable architectures. *International Journal of Computer Applications*, 48:1–7, 06 2012.

[42] Thomas N. Theis and H. S. Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science and Engg.*, 19(2):41–50, March 2017.

[43] Frederick Tombs, Alireza Mellat, and Nachiket Kapre. Mocarabe: High-performance time-multiplexed overlays for fpgas. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 115–123, 2021.

[44] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling, and S. Hauck. Static versus scheduled interconnect in coarse-grained reconfigurable arrays. In *2009 International Conference on Field Programmable Logic and Applications*, pages 268–275, 2009.

[45] M. J. P. Walker and J. H. Anderson. Generic connectivity-based cgra mapping via integer linear programming. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 65–73, 2019.

[46] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. Cascade: High throughput data streaming via decoupled access-execute cgra. *ACM Trans. Embed. Comput. Syst.*, 18(5s), October 2019.

[47] Inc Xilinx. *UltraFast Design Methodology Guide for the Vivado Design Suite (UG949 (v2020.1))*, August 2020.

[48] Inc Xilinx. *Vivado Design Suite User Guide, Synthesis (UG901 (v2019.2))*, January 2020.

[49] Inc Xilinx. *Xilinx AI Engines and Their Applications(WP506 (v1.1))*, July 2020.

[50] Inc Xilinx. *Alveo U280 Data Center Accelerator Card Data Sheet (DS963 (v1.4))*, September 2021.

[51] Inc Xilinx. *Versal ACAP Programmable Network on Chip and Integrated Memory Controller v1.0 (PG313 (v1.0))*, November 2021.

[52] Inc Xilinx. *Xilinx Alveo Product Selection Guide*, 2021.

[53] Deheng Ye. Bitgpu. https://github.com/YeDeheng/bitgpu, 2015.

[54] N. Zhang, X. Chen, and N. Kapre. Rapidlayout: Fast hard block placement of fpga-optimized systolic arrays using evolutionary algorithms. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 145–152, 2020.