Architectures and Tools for Efficient Reconfigurable Computing

by

Stephen Alexander Chin

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto

© Copyright 2018 by Stephen Alexander Chin
Abstract

Architectures and Tools for Efficient Reconfigurable Computing

Stephen Alexander Chin

Doctor of Philosophy

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto

2018

Recent decades have seen large growth in the silicon industry with transistor scaling and transistor count approximately doubling every two years. With the continued growth of transistors-per-chip and increasing power density, dark silicon challenges have risen. Reconfigurable computing poses a possible solution to some of the challenges through improving performance and energy efficiency by tailoring the hardware to the application. Field-Programmable Gate Arrays (FPGAs) are a platform for realizing reconfigurable computing and have had traction for over a decade. Their recent introduction into mainstream data-centres bodes well for the field. Another type of reconfigurable architecture, Coarse Grained Reconfigurable Arrays (CGRAs), poses one other platform for computing. Being more specialized than FPGAs, CGRAs’ main selling point is increased efficiency versus FPGAs, at the cost of platform flexibility. This dissertation looks at efficient computing first from the perspective of FPGAs, developing new architectures and new CAD tools, making for a more efficient FPGA. The proposed FPGA architecture consists of a hybrid multiplexer / look-up-table logic block that has reduced area with respect to traditional architectures. Then, with the prospect that CGRA architectures hold, we develop an open-source framework, CGRA-ME, for the modelling and exploration of CGRAs. This unifying software framework incorporates, architecture description through a custom language, architecture modelling, application mapping, and RTL generation, and allows further development of CGRA architectures and related CAD tools throughout the research community. Within the CGRA-ME framework, a new architecture-agnostic application mapper formulated in an integer linear program was also developed for generic CGRAs. Through this dissertation, we have made headway towards more efficient reconfigurable architectures through architecture design and related CAD and are optimistic that these contributions will have positive impact on further research and industrial application of reconfigurable architectures.
Acknowledgements

This thesis and the success of my graduate studies would not have been possible without the support of many individuals. I would first like to express my gratitude to my supervisor Prof. Jason Anderson for supporting and guiding me throughout my tenure as a PhD student. I have learned many skills during my time as a graduate student and many can be attributed to him. Without Jason’s support and drive for the CGRA-ME project, it would not be where it is today. I would like to thank my doctoral committee, Prof. Vaughn Betz, Prof. Paul Chow, Prof. Andreas Moshovos, and Dr. Soojung Ryu. I am grateful for the time and effort put into their insightful questions and detailed comments.

I would like to thank everyone who has worked on the CGRA-ME project: Jim Zhao, Allan Rui, Noriaki Sakamoto, Alex Mertens, Steven Yin, Steven Niu, Matthew Walker, Prof. Jongeun Lee, and Prof. Jason Anderson. Without them, the CGRA-ME project would not have been possible. Your collective efforts are greatly appreciated and I am thankful to have worked with all of you.

I would like to thank Prof. Paul Chow for the strong foundation built during my MASc that enabled my pursuit of a PhD. Paul has been a continuing source of guidance throughout, while also providing opportunity for growth, especially through leadership and teaching roles.

I wish to acknowledge the many colleagues and friends within the ECE department who have impacted my career in many ways during my doctoral studies, in no particular order and undoubtedly incomplete: Jason Luu, Safeen Huda, Davor Capalija, Jasmina Vasiljevic, Tahir Diop, Ruediger Willenberg, Charles Lo, Fernando Martin del Campo, Andrew Shorten, Robert Heße, and Mario Badr. The office within PT477 with its many members will always be a fond and cherished memory with its long nights, long lunches, and long coffee breaks. I hope the friendships forged within those confines will continue to endure in the years to come. I am extremely grateful for every single one of my friends outside of graduate school. The importance of balance during grad school cannot be overemphasized and you have all, in your own individual ways, helped to keep my life in balance.

The importance of my family can never be understated. My parents, Stephen and Sharon, you have always championed me throughout my life. Along with my sister Rebecca, brother Nicholas, and brother-in-law Mark, you have all been an unending source of support. My parents-in-law, Stella and James, thank you for your continual love and care – it’s always present when we are together.

Last and most importantly, I would like to thank my wife, Clara, for her patience, understanding, and unconditional love and support for myself and my endeavours throughout the years. This being one of my most significant undertakings, the time spent has of course not gone by without its fair share of trials and challenges. I’m glad to have faced them with you as my teammate and partner. You have been by my side through thick and thin – I am and will be eternally grateful.
Contents

Acknowledgements iii

Contents iv

List of Tables vii

List of Figures ix

1 Introduction 1

1.1 Motivation ........................................ 1
1.2 Contributions ...................................... 4
1.3 Thesis Organization ................................. 5

2 Hybrid LUT/Multiplexer FPGA Architectures 6

2.1 Background ........................................ 6
2.2 Related Work ...................................... 7
2.3 Proposed Architectures ............................... 8
  2.3.1 MUX4: A 4-to-1 Multiplexer-based Logic Element 8
  2.3.2 Logic Elements, Fracturability & MUX4-Based Variants 9
  2.3.3 Hybrid Complex Logic Block ...................... 10
  2.3.4 Area Modelling ................................... 13
2.4 Technology Mapping using ABC ..................... 16
  2.4.1 NaturalMux ....................................... 18
  2.4.2 MuxMap .......................................... 18
  2.4.3 Select Mapping ................................. 19
2.5 Modelling using VPR ................................. 19
2.6 Experimental Evaluation ............................. 20
# List of Tables

1.1 Field Programmable Gate Array (FPGA) implementation overhead compared to custom ASIC implementation \[1\] ................................. 3

2.1 Logic element transistor models with area given in minimum-width transistor area (MWTA) and Delays scaled for a 40 nm process ................................. 15

2.2 Per architecture, minimum relative architectural percentage area and tolerable percentage CLB increase assuming constant routing demand ................................. 16

2.3 Post mapping area estimate for VTR7 and CHStone benchmarks assuming complete CLB packing and no increase in routing demand ................................. 24

2.4 Non-Fracturable Architecture: Post Place & Route results showing architectural sweeps between zero (baseline) to five $MUX_4$ logic elements out of ten, for fracturable hybrid-CLB architectures, using the NaturalMux mapping scheme. All benchmarks were averaged over five placement seeds ................................. 27

2.5 Non-Fracturable Architecture: Post Place & Route results showing architectural sweeps between zero (baseline) to five $MUX_4$ logic elements out of ten, for fracturable hybrid-CLB architectures using the MuxMap mapping scheme. All benchmarks were averaged over five placement seeds ................................. 28

2.6 Fracturable Architecture: Post Place & Route results showing architectural sweeps between zero (baseline) to five $MUX_4$ logic elements out of ten, for fracturable hybrid-CLB architectures, using the NaturalMux mapping scheme. All benchmarks were averaged over five placement seeds ................................. 30

2.7 Fracturable Architecture: Post Place & Route results showing architectural sweeps between zero (baseline) to five $MUX_4$ logic elements out of ten, for fracturable hybrid-CLB architectures, using MuxMap mapping scheme. All benchmarks were averaged over five placement seeds ................................. 31
2.8 Geomean FMax ratio for each architecture for Select mapping relative to the standard ten 6-LUT architecture for each benchmark suite. Here, higher change is better.

4.1 Benchmarks statistics. I/Os also includes memory operations, load and store. The add and multiply benchmarks are trees of operations with 10 to 16 inputs. 2x2-p and 2x2-f are synthetic benchmarks. mac is a multiply-accumulate and weighted sum is as it’s named.

The other benchmarks are Taylor series approximations of common transcendental functions.

4.2 Benchmark mapping results for single and dual context architectures, four array styles and three array sizes. A Mapped benchmark is denoted by M and an Unmapped benchmark denoted by U. An M denotes a benchmark that was able to be mapped in two contexts but not in one context.

4.3 Area/timing results; single-context 4 × 4 architectures.

4.4 Mapping results for both Full and Reduced architectures. M indicates the benchmark was able to be mapped on the architecture. U indicates that the benchmark was unable to be mapped on the architecture. Mapping runtimes for each architecture and benchmark are also reported. Statistics for the benchmarks are also included listing the number of compute operations, number of multiplication operations, and the number of I/Os. Ops is the total number of operations including mult operations.

4.5 FreePDK45 45 nm standard-cell post-place-and-route area and performance of all four designs.

4.6 FPGA area and performance for both architectures on Stratix IV.

5.1 Benchmarks

5.2 Mapping Results for the ILP mapper. 1 signifies a feasible mapping, 0 signifies an infeasible mapping. T, signifies a solver timeout where the solver was unable to find a feasible solution or prove infeasibility.
List of Figures

1.1 Microprocessor trends over the past 42 years [2] ........................................... 2

2.1 The $MUX_4$ logic element depicting optional data input inversions using 2-to-1 multiplexers with configurable SRAM ......................................................... 9

2.2 Fracturable 6-LUT that can be fractured into two 5-LUTs with two shared inputs .... 10

2.3 Dual $MUX_4$ logic element that utilizes dedicated select inputs and shared data inputs . . . 11

2.4 The Hybrid CLB with a 50% depopulated intra-CLB crossbar depicting BLE internals for a non-fracturable (one optional register and one output) or fracturable (two optional registers and two outputs) architecture. ......................................................... 12

2.5 The Hybrid CLB with a 50% depopulated intra-CLB crossbar depicting BLE internals for a non-fracturable (one optional register and one output) or fracturable (two optional registers and two outputs) architecture. ......................................................... 13

2.6 $MUX_4$ logic element model ................................................................. 19

2.7 $MUX_4$ mode in the 6-LUT element model ........................................... 20

2.8 MuxMap varying the weight from zero area to equal 6-LUT area for different $MUX_4$:LUT architectures. A weight equal to LUT area is equivalent to the NaturalMux mapping. As the $MUX_4$ area weighting is increased for some architecture ratios (e.g. 5:5 and 4:6), an increase in relative area is seen due to unbalanced circuit and architecture $MUX_4$:LUT ratios. ................................................................. 22

2.9 Input-size distribution of VTR7 benchmarks with $MUX_4$-embeddable logic elements in blue (light grey) and LUTs in purple (dark grey). ........................................... 25

2.10 Input-size distribution for CHStone benchmarks with $MUX_4$-embeddable cuts in blue (light grey) and LUTs in purple (dark grey). ........................................... 25
3.1 A spectrum of architectures showing a CGRA’s position in the spectrum of efficiency / programmability relative to competing architectures. ................................. 35
3.2 One cell of the RaPilD architecture. (Figure reproduced [3]) ................................. 41
3.3 High-level architecture of the KressArray showing the reconfigurable datapath architecture (rDPA) and associated supporting logic. (Figure reproduced [4]) ................................. 42
3.4 One stripe of the PipeRench architecture. (Figure reproduced [5]) ................................. 43
3.5 High-level architecture of the Morphosys architecture showing the main components of the Reconfigurable Cell array, the host processor, etc. (Figure reproduced [6]) ................................. 43
3.6 The architecture of the Morphosys reconfigurable cell array. (Figure reproduced [6]) ................................. 44
3.7 The architecture of an ADRES CGRA featuring an 8 × 8 array of functional units with local register files. (Figure reproduced [7]) ................................. 45
3.8 Mosaic ‘Island-style’ CGRA showing clusters of tightly coupled functional units that are surrounded by channels of interconnect, connected through switch-boxes. (Figure reproduced [8]) ................................. 45
3.9 Renesas DRP: Tile architecture DRP-4. (Figure reproduced [9]) ................................. 46
3.10 Samsung ULP-SRP high-level architecture (Figure reproduced [10]) ................................. 47
3.11 Wave Computing DPU: High-level architecture showing 1024 clusters, peripheral AXI ports and an example partitioning scheme of 24 groups. (Figure reproduced [11]) ................................. 48
3.12 Wave Computing DPU: Cluster architecture containing four quads of processing elements and eight special arithmetic units. (Figure reproduced [11]) ................................. 48
4.1 CGRA-ME framework overview showing the main components. ................................. 53
4.2 C++ API calls to generate an I/O block. ................................. 55
4.3 Example CGRA processing block made from primitive components: three multiplexers, a functional unit and a register. Configuration cells (not shown) for the functional unit and routing multiplexers are automatically inferred within the framework when using the CGRA-ADL language (Section 4.3.2). ................................. 55
4.4 C++ API calls to generate the block in Figure 4.3 including configuration cells. ................................. 56
4.5 CGRA-ADL architecture description example showing the declaration of a module called “block1” with instances of this block being used to populate a 4×4 architecture. ................................. 57
4.6 Using CGRA-ADL to describe a module (named module1) that contains a single functional unit and also to describe module1’s internal connections to the functional unit’s input and output ports. An internal wire w is also modelled. ................................. 58
4.7 CGRA-ADL interconnect structures. Line 1 models direct connections; line 2 models distributive connections; line 3 models multiplexers, and line 4 models crossbars.

4.8 CGRA block patterns using CGRA-ADL. This code snippet populates a 4-by-4 region of the CGRA with a repeating 2-by-2 pattern of CGRA blocks. col="2" signifies the pattern of four modules is two columns wide and must wrap around to a second row. The row-range and col-range parameters specify the range of block coordinates where the pattern should be applied.

4.9 Example interconnection showing the use of relative position macros for instance names in the red circled portion of the figure. An instance of an FUb and an instance of an I/O module are created. Then two connections are made, one from the IO ‘out’ port to the FUb ‘in1’ port, and one from the FUb ‘out’ port to the ‘in2’ port of FUa.

4.10 CGRA interconnection architectures for which automatic connection patterns are provided.

4.11 Example data-flow graph and textual representation.

4.12 A two context MRRG snippet and a DFG showing how the mapping process associates operations on the DFG with functional units (orange) on the MRRG and routes values on routing resources (blue).

4.13 Floorplans and Layouts of 4×4 homogeneous orthogonal architecture (left) and heterogeneous diagonal architecture (right).

4.14 Full and Reduced variants of ADRES architecture.

4.15 FreePDK45 45 nm standard-cell floorplan for Cadence’s Innovus place and route.

4.16 FreePDK45 45 nm standard-cell layout for all four designs in physical view.

4.17 FreePDK45 45 nm standard-cell layout for all four designs in amoeba view, showing how the loose floorplanning constraints were met by the tool.

4.18 FreePDK45 45 nm standard-cell area breakdown of Full and Reduced architectures, when targeting minimum area and minimum delay.

4.19 Floorplanned Stratix IV implementations of the Full CGRA (left), and Reduced CGRA (right).

4.20 FPGA area breakdown for both architecture variants. Additionally, the Full architecture uses 8 DSP blocks for its 16 32-bit multiply units and the Reduced uses 5 for its 10 32-bit multiply units.

5.1 MRRG fragment for a multiplexer and register showing multiple cycles / contexts.
5.2 MRRG fragments for different latency and initiation interval functional units showing multiple cycles / contexts .............................................. 82

5.3 An example functional block that contains a functional unit (with a latency of 0 cycles), register and input multiplexers, and its corresponding MRRG structure for 1 cycle / 1 context ......................................................... 83

5.4 Pseudo-code for the simulated annealing mapper .............................................. 85

5.5 Two DFG fragments to illustrate value routing/fanout ........................................ 87

5.6 Three illustrative MRRGs ................................................................. 87

5.7 The arrangement of functional blocks in our test architectures for Orthogonal connectivity. The functional blocks are orthogonally connected to their nearest neighbours, while the periphery contains I/Os. Each row of Functional Blocks share connectivity to one memory access port ................................................................. 93

5.8 The mapping flow within our framework for ILP-based mapping and simulated annealing-based mapping. The architecture description and the number of contexts is input into the framework, where it generates an MRRG. For ILP-based mapping, the ILP formulation is created from the input DFG benchmark and the MRRG. The formulation is then solved by the ILP Solver, in our case Gurobi. The simulated annealing-based mapper takes the DFG and MRRG directly as inputs to map the benchmark to the architecture .............................................. 96

5.9 Comparison between the Simulated Annealing Mapper and the ILP Mapper .......... 97
Chapter 1

Introduction

1.1 Motivation

Recent decades have seen large growth in the silicon industry with transistor scaling (Figure 1.1) holding steadfast to Moore’s Law [12], and transistor count approximately doubling every two years. During the mid 2000’s owing to the breakdown of Dennard scaling [13], multicore processors were produced, utilizing growing silicon area and increasing performance. With the continued growth of transistors-per-chip and increasing power density, dark silicon challenges have arose [14]. Specialized circuits and reconfigurable computing pose a possible solution to some of these challenges through improving performance and energy efficiency by tailoring the hardware to the application [15].

Field-Programmable Gate Arrays (FPGAs) are a platform for realizing reconfigurable computing. In embedded systems, reconfigurable silicon chips can outperform microprocessors in speed and energy efficiency by allowing the user to create specialized circuits that are tailored to specific applications. FPGAs are programmable integrated circuits that can be configured by the end user to realize generic digital circuits. The primary logic element in such commercial FPGAs is a Look-Up-Table (LUT), which is a hardware realization of a programmable truth table. A programmable interconnection fabric within the FPGA permits unique routing paths to be chosen for individual logic signals between logic elements, implementing the circuit itself.

FPGAs are a multi-billion dollar industry that steadily gained traction since their inception in the 1980’s. The two major FPGA companies Intel (formerly Altera) and Xilinx have produced commercial products since this time. Currently, these two main FPGA vendors command 90% of the overall FPGA market, and offer chips with very fine-grained programmability that are widely used in many applications,
Chapter 1. Introduction

2

Figure 1.1: Microprocessor trends over the past 42 years [2].

ranging from communications, to automotive to consumer. Initially used for glue logic, FPGAs advanced to communications and DSP applications. And more recently, their market has broadened with FPGAs now appearing in data-centres for high-performance computing, accelerating varied workloads [16, 17, 18, 19].

In 2014, Microsoft announced that FPGAs were being used to accelerate their Bing search engine platform [16] and again in 2017, Microsoft presented the Configurable Cloud data-centre architecture [17]. The data-centres tackle workloads within the fields of machine learning as well as multimedia. Additionally, in 2017, cloud computing providers Amazon and Alibaba offer cloud computing FPGA solutions [18, 19].

Though the use of FPGAs is widespread and they have clear performance gains for tailored applications, FPGAs as a reconfigurable computing platform do have drawbacks. FPGAs are able to outperform other platforms for certain applications but FPGA implementations are not as efficient or high performing compared to a full custom Application Specific Intergrated Chip (ASIC) implementation of the application [20]. Of course, this is the price to be paid for reconfigurability. Because of the overhead required for an FPGA’s fine-grained programmability (shown in Table 1.1), FPGAs are 3-5× slower, and 20-40× less area-efficient than ASICs when implementing a given logic circuit [1]. This is a profitable compromise as evidenced by the uptake of FPGAs in the market over recent decades. Since
Table 1.1: FPGA implementation overhead compared to custom ASIC implementation [1].

<table>
<thead>
<tr>
<th>Metric</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area</td>
<td>20-40×</td>
</tr>
<tr>
<td>Delay</td>
<td>3-5×</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>14×</td>
</tr>
<tr>
<td>Static Power</td>
<td>5-90×</td>
</tr>
</tbody>
</table>

an FPGA’s programmability opens up economies of scale for their production, the fixed costs associated with silicon production are amortized over many FPGAs chips that service many different applications.

Within reconfigurable computing, there is another style of architecture called Coarse-Grained Reconfigurable Arrays (CGRAs). CGRAs are a class of programmable hardware architectures with coarse-grained logic blocks, often resembling Arithmetic Logic Units (ALUs) and datapath-style interconnect, where buses of signals are routed together as a group. Computation is performed via an assortment of functional units connected through the reconfigurable interconnect with some distributed temporary storage mechanisms such as registers. In many cases, the schedule of operations and communication of data throughout the architecture is static, requiring fixed delays within functional units and through the interconnect. While FPGAs realize custom hardware designs through bits and Boolean functions, CGRAs realize circuits on a bus and functional unit level. That is, FPGAs use single-bit interconnect, routed individually, and a mix of fine LUTs and coarse-grained blocks (Digital Signal Processing blocks (DSPs)) to implement logic, while CGRAs use multi-bit routing and ALUs to facilitate computation. The value proposition of CGRAs is improved speed and area-efficiency relative to FPGA owing to there being less overhead for programmability. CGRAs are less “generic” than FPGAs, which may provide benefit for applications whose computational requirements align closely with the functionality of the CGRA architecture. One can think of a CGRA as being midway between an FPGA and an ASIC, providing some programmability like FPGAs while delivering superior power, performance, and cost characteristics for aligned applications. We believe CGRAs are especially attractive in this context, when tailored to specific domains where computational requirements are known, yet programmability is desired, possibly to accommodate an evolving technical specification.

CGRAs are thus less flexible than FPGAs, affecting both the ease of application mapping, as well as the power/performance/cost, elaborated upon below. From the early beginnings of CGRAs designs sought to increase silicon efficiency of existing reconfigurable computing platforms, FPGAs with early CGRA architectures such as RaPiD [3], CGRAs have been studied in academia for over a decade, and a variety of different architectures have been proposed [21] [22] [23]. Commercially, recent years have also seen CGRAs appear in the market [9] [10] [11].
Chapter 1. Introduction

A key benefit of CGRAs versus FPGAs is in application mapping runtime. Owing to their reduced flexibility, CAD tools for CGRAs need to make fewer decisions, reducing CAD complexity significantly. This brings runtimes closer to software compilation, as opposed to the hours or days that may be required for application mapping to FPGAs. Moreover, applications targeted to CGRAs are typically specified at a higher abstraction level than traditional hardware design. Applications are specified as data-flow graphs or in software, as opposed to Verilog/VHDL RTL.

Many prior works on CGRA architectures have been “point solutions”, wherein a research group or company proposes a CGRA architecture and then demonstrates its utility for a basket of applications. The relative performance between these “point solutions” are also difficult to judge since not all studies include detailed circuit modelling for their architectures. Compounded with differing CAD tool flows that map benchmarks to these architectures, it makes for difficult and possibly inaccurate comparisons between architectures. While there exist software tools that allow the modelling and evaluation of fine-grained FPGA architectures [24], to my knowledge, there is no analogous framework that permits the scientific exploration of the CGRA architectural space.

1.2 Contributions

In this dissertation, we target and make headway to close the efficiency gap between traditional reconfigurable computing, using FPGAs and ASICs. We attack the problem from two fronts – the first from the perspective of FPGAs and the second from the perspective of CGRAs.

The first contribution of this thesis is an architectural study of FPGAs that use new hybrid structures based on a combination of traditional LUTs and an alternative logic element. Chapter 2 presents this architecture study. The proposed alternative logic element is based on a 4-input multiplexer and incorporated alongside traditional LUTs within a Configurable Logic Block (CLB). The new logic element is less flexible compared to LUTs because in comparison, it has a smaller set of mappable functions. However, the new logic element costs \( \sim 10\% \) the area as compared to a 6-LUT while maintaining delay characteristics. A number of FPGA architectures having different ratios of two types of logic elements are proposed. These proposed hybrid architectures are modelled within VPR [24] and the VTR tool flow [25] was used for evaluation with standard benchmarks. A modified technology mapper was also developed within the VTR framework for targeted mapping towards these new logic elements. A preliminary study focused on non-fracturable architectures was first published at the IEEE International Conference on Field-Programmable Technology (ICFPT) [26], with the full study that also considers fracturable architectures published in IEEE Transactions on Very Large Scale Integration Systems (TVLSI) [27].
The second contribution is a new open-source CGRA architecture modelling and evaluation framework, named CGRA-ME \cite{28}. CGRA-ME fills a significant gap within CGRA research as there is no open-source or standard evaluation framework for CGRAs. Chapter \ref{chapter4} introduces the CGRA-ME framework with its design and its capabilities. Two experimental studies demonstrate the framework’s ability to model a variety of architectures, compile and map benchmarks, and produce RTL designs of hypothetical CGRA architectures. This framework was first published in the proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP) \cite{29} with a follow-up study presented in the proceedings of the ACM/SIGDA International Symposium on Physical Design (ISPD) \cite{30} that includes the modelling of more complex architectures and the implementation of CGRA overlays.

Finally, the third contribution of this dissertation is a new architecture agnostic integer linear programming-based CGRA mapping formulation. The Modulo Routing Resource Graph (MRRG) representation has emerged as a generic way of abstraction for the device model of the CGRA. The integer linear programming formulation leverages this MRRG representation, allowing for a generic formulation that is generated based on the candidate mapping architecture. Chapter \ref{chapter5} presents this formulation and provides a comparison versus a baseline Simulated Annealing \cite{31} based mapper within CGRA-ME. This mapper formulation will be presented at the Design Automation Conference in 2018 \cite{32}.

\section{1.3 Thesis Organization}

This dissertation is organized as follows. Chapter \ref{chapter2} presents the architectural study on hybrid logic structures within FPGAs. Chapter \ref{chapter3} provides background on CGRA architecture development and research. Chapter \ref{chapter4} introduces the CGRA-ME framework, demonstrating its use on two experimental studies. Chapter \ref{chapter5} discusses mapping within the CGRA-ME framework and the Simulated Annealing \cite{31} based mapper and introduces constraint-based CGRA application mapping along with the formulation and implementation of the architecture agnostic integer linear programming based mapper. A demonstration of the mapper’s performance and capabilities using multiple benchmarks and architectures is also given. Finally, conclusions and paths for future work are provided in Chapter \ref{chapter6}.
Chapter 2

Hybrid LUT/Multiplexer FPGA Architectures

Throughout the history of Field-Programmable Gate Arrays (FPGAs), Look-up-Tables (LUTs) have been the primary logic element used to realize combinational logic. A $K$-input LUT is generic and very flexible – able to implement any $K$-input Boolean function. The use of LUTs simplifies technology mapping as the problem is reduced to a graph covering problem. However, an exponential area price is paid as larger LUTs are considered. $K$ between 4 and 6 is typically seen in industry and academia, and this range has been demonstrated to offer a good area/performance compromise \[33\] \[34\]. Recently, a number of other works have explored alternative FPGA logic element architectures for performance improvement \[35\] \[36\] \[37\] \[38\] \[39\] to close the large gap between FPGAs and ASICs (application-specific integrated circuits) \[1\]. In this chapter, we propose incorporating (some) hardened multiplexers in FPGA logic blocks as a means of increasing silicon area efficiency and logic density, towards closing the gap of FPGAs to ASICs. Non-fracturable and fracturable architectures are considered. The study on non-fracturable architectures was published at the IEEE International Conference on Field-Programmable Technology (ICFPT) \[26\] and the study on fracturable architectures was published in the IEEE Transactions on Very Large Scale Integration Systems (TVLSI) \[27\].

2.1 Background

Multiplexer-based logic blocks for FPGAs have seen success in early commercial architectures, such as the Actel ACT-1/2/3 architectures, and efficient mapping to these structures has been studied \[40\] in the
early '90s. However, their use in commercial chips has waned, perhaps partly due to the ease with which logic functions can be mapped into LUTs, simplifying the entire computer-aided design (CAD) flow. Nevertheless, it is widely understood that LUTs are inefficient at implementing multiplexers, and that multiplexers are frequently used in logic circuits. To underscore the inefficiency of LUTs implementing multiplexers, consider that a six input LUT (6-LUT) is essentially a 64-to-1 multiplexer (to select 1 of 64 truth-table rows) and 64 SRAM configuration cells, yet it can only realize a 4-to-1 multiplexer (4 data + 2 select = 6 inputs).

We consider a 6-input logic element based on a 4-to-1 multiplexer, $MUX_4$, that can realize a subset of 6-input Boolean logic functions, and a new hybrid complex logic block (CLB) that contains a mixture of $MUX_4$s and 6-LUTs. The proposed $MUX_4$s are small compared to a 6-LUT (15% of 6-LUT area), and can efficiently map all $\{2, 3\}$-input functions and some $\{4, 5, 6\}$-input functions. In addition, we explore fracturability of logic elements – the ability to split logic elements into multiple smaller elements – in both LUTs and $MUX_4$s to increase logic density. The ratio of logic elements that should be LUTs versus $MUX_4$s is also explored towards optimizing logic density for both non-fracturable and fracturable FPGA architectures. Our aim is not to fully replace the LUT but create a hybrid CLB structure that incorporates both elements. The ratio of logic elements that should be LUTs versus $MUX_4$s can then be explored towards optimizing logic density for both non-fracturable and fracturable FPGA architectures.

To facilitate architecture exploration, we developed a CAD flow for mapping into the proposed hybrid CLBs, created using $ABC$ [41] and $VPR$ [24], and describe technology mapping techniques that encourage the selection of logic functions that can be embedded into the $MUX_4$ elements. The main highlights of this chapter are:

- Two hybrid CLB architectures (non-fracturable and fracturable) that contain a mixture of $MUX_4$ logic elements and traditional LUTs yielding up to 8% area savings.
- Mapping techniques called $NaturalMux$ and $MuxMap$ targeted towards the hybrid CLB architecture that optimize for area, while preserving original mapping depth.
- A full post place and route architecture evaluation with VTR7 [25], and CHStone [42] benchmarks facilitated by LegUp HLS [43], VTR [25] showing impact on both area and delay.

### 2.2 Related Work

Works have shown that heterogeneous architectures and synthesis methods can have a significant impact on improving logic density and delay, narrowing the ASIC-FPGA gap. Works by Anderson and Wang
with “gated” LUTs [36], then with asymmetric LUT logic elements [37], show that LUT elements present in commercial FPGAs provide unnecessary flexibility.

Towards improved delay and area, macro cell-based FPGA architectures have been proposed [38, 39]. These works describe significant changes to traditional FPGA architectures, whereas the changes proposed here build on architectures used in industry and academia [33]. Similarly, And-Inverter Cones have been proposed as replacements for LUTs, inspired by And-Inverter graphs [35].

Work by Purnaprajna and Ienne [44] explored the possibility of re-purposing existing multiplexers contained within Xilinx Logic Slices [45]. Similar to this work, they use the ABC priority cut mapper, as well as VPR for packing, place, and route. However, their work is primarily delay based showing an average speedup of 16% using only ten of 19 VTR7 benchmarks.

### 2.3 Proposed Architectures

A number of FPGA architecture variants were evaluated and all are based on the basic $MUX_4$ element described in Section 2.3.1. The design of Logic Elements and considerations for fracturability are addressed in Section 2.3.2 followed by Hybrid-CLB design in Section 2.3.3. The area of all proposed architectures is discussed in Section 2.3.4.

#### 2.3.1 MUX4: A 4-to-1 Multiplexer-based Logic Element

The $MUX_4$ logic element shown in Figure 2.1 consists of a 4-to-1 multiplexer with optional inversion on its inputs that allow the realization of any $\{2, 3\}$-input function, some $\{4, 5\}$-input functions, and one 6-input function – a 4-to-1 multiplexer itself with optional inversion on the data inputs. A 4-to-1 multiplexer matches the input pin count of a 6-LUT, allowing for fair comparisons with respect to connectivity and intra-cluster routing. However, it should be noted that permutability of inputs for $MUX_4$ elements is constrained compared to the 6-LUT.

Naturally, any two-input Boolean function can be easily implemented in the $MUX_4$: the two function inputs can be tied to the select lines, and the truth table values (logic-0 or logic-1) can be routed to the data inputs accordingly. Or alternately, a Shannon decomposition can be performed about one of the two variables – the variable can then feed a select input. The Shannon co-factors will contain at most one variable and can therefore be fed to the data inputs (the optional inversion may be needed).

For three-input functions, consider that a Shannon decomposition about one variable produces co-factors with at most two variables. A second decomposition of the co-factors about one of their two remaining variables produces co-factors with at most one variable. Such single-variable co-factors can be
Figure 2.1: The $MUX_4$ logic element depicting optional data input inversions using 2-to-1 multiplexers with configurable SRAM.

fed to the data inputs (the optional inversion may be needed), with the decomposition variables feeding the select inputs. Likewise, functions of more than four inputs can be implemented in the $MUX_4$ as long as Shannon decomposition with respect to any two inputs produce co-factors with at most one input.

Observe that input inversion on each select input is omitted as this would only serve to permute the four multiplexer data inputs. While this could help routability within the CLB’s internal crossbar, additional inversions on the select inputs would not increase the number of Boolean functions that are able to map to the $MUX_4$ logic element.

### 2.3.2 Logic Elements, Fracturability & $MUX_4$-Based Variants

Two families of architectures were created - one without fracturable logic elements and one with fracturable logic elements. Here, fracturable logic elements refer to an architectural element on which one or more logic functions can be optionally mapped. Non-fracturable logic elements refer to an architectural element on which only one logic function is mapped. In the non-fracturable architectures, the $MUX_4$ element shown in Figure 2.1 is used together with non-fracturable 6-LUTs. This element shares the same number of inputs as a 6-LUT lending for fair comparison with respect to input connectivity.

For the fracturable architecture, we consider an 8-input logic element, closely matched with the
A 6-LUT that can be fractured into two 5-LUTs using eight inputs is shown in Figure 2.2. Two 5-input functions can be mapped into this logic element if two inputs are shared between the two functions. If no inputs are shared, two 4-input functions can be mapped to each 5-LUT. For the $MUX_4$ variant, $Dual MUX_4$, we use two $MUX_4$s within a single 8-input logic element. In the configuration shown in Figure 2.3, the two $MUX_4$s are wired to have dedicated select inputs and shared data inputs. This configuration allows this structure to map two independent (no shared inputs) 2-input functions, while larger functions may be mapped dependent on the shared inputs between both functions.

An architecture in which a 4-to-1 multiplexer ($MUX_4$) is fractured into two smaller 2-to-1 multiplexers was first considered. However, since a 2-to-1 multiplexer’s mapping flexibility is quite limited, as it can only map 2-input functions and the 3-input 2-to-1 MUX itself, little benefit was added compared to the overheads of making the $MUX_4$ fracturable and poor area results were observed.

### 2.3.3 Hybrid Complex Logic Block

A variety of different architectures were considered – the first being a non-fracturable architecture. In the non-fracturable architecture, the CLB has 40 inputs and ten basic logic elements (BLEs), with each BLE having six inputs and one output following empirical data in prior work [33]. Figure 2.4 shows
Figure 2.3: *Dual MUX4* logic element that utilizes dedicated select inputs and shared data inputs.
Figure 2.4: The Hybrid CLB with a 50% depopulated intra-CLB crossbar depicting BLE internals for a non-fracturable (one optional register and one output) or fracturable (two optional registers and two outputs) architecture.

This non-fracturable CLB architecture with BLEs that contain an optional register. We vary the ratio of \textit{MUX4}s to LUTs within the ten element CLB from 1:9 to 5:5 \textit{MUX4}s:6-LUTs. The \textit{MUX4} element is proposed to work in conjunction with 6-LUTs, creating a hybrid CLB with a mixture of 6-LUTs and \textit{MUX4}s (or \textit{MUX4} variants). Figure 2.4 illustrates the organization of our CLB and internal BLEs.

For fracturable architectures, the CLB has 60 inputs and ten BLEs, with each BLE having eight inputs and two outputs emulating an Altera Stratix Adaptive-LUT [46]. The same sweep of \textit{MUX4} to LUT ratios was also performed. Figure 2.5 shows the fracturable architecture with 8 inputs to each BLE that contains two optional registers. We evaluate the fracturability of logic elements in the context of \textit{MUX4} elements since fracturable LUTs are common in commercial architectures. For example, Altera Adaptive 6-LUTs in Stratix IV and Xilinx Virtex 5 6-LUTs can be fractured into two smaller LUTs with some limitations on inputs.

The crossbar for fracturable architectures is larger than the non-fracturable architectures for two reasons. Due to the virtual increase of logic elements, a larger number of CLB inputs are required, which increases crossbar size. Since there are now twice as many outputs from the logic elements, these additional outputs need to also be fed back into the crossbar, also increasing its size. Due to this disparity in crossbar size, fair comparisons cannot be made between fracturable and non-fracturable architectures.
Figure 2.5: The Hybrid CLB with a 50% depopulated intra-CLB crossbar depicting BLE internals for a non-fracturable (one optional register and one output) or fracturable (two optional registers and two outputs) architecture.

Therefore, we compare non-fracturable hybrid CLB architectures to a baseline LUT-only non-fracturable architecture and we compare fracturable hybrid CLB architectures to a baseline LUT-only fracturable architecture.

Sparse crossbars have been previously studied [47] and here, we model a 50% depopulated crossbar within the CLB for intra-cluster routing for both non-fracturable and fracturable architectures. Extended discussion on architecture modelling follows in Section 2.5.

2.3.4 Area Modelling

MUX4 Logic Element

Initial estimates of the $MUX_4$ element showed that the $MUX_4$ is approximately 10% the area of a 6-LUT overall. A 4-to-1 multiplexer can be realized with three 2-to-1 multiplexers. Hence, the $MUX_4$ element contains seven 2-to-1 multiplexers, four SRAM cells, and four inverters in total (see Figure 2.1). The optional inversion uses the four SRAM cells, whereas the rest of the logic element configuration is performed through routing. Additionally, the depth of the multiplexer tree is halved compared to the 6-LUT, which has six 2-to-1 multiplexers on its longest paths. Conservatively, assuming constant
pass transistor sizing and that the area of a 2-to-1 multiplexer and 6 transistor SRAM cell are roughly equivalent, the $MUX_4$ element has $\frac{1}{10}$th the SRAM area and $\frac{1}{8}$th the multiplexer area of a 6-LUT.

These estimates were revised using transistor level modelling of the circuit blocks. Transistor-level optimization of the constituent circuit blocks of an FPGA requires an understanding of the optimal area-delay trade-offs for each individual circuit block. This requires extracting a ‘representative critical path’, which is a path whose composition of blocks and topology will be similar to the critical path of a specific design. Extracting the representative critical path allows us to judge to what extent each individual block is timing critical, establishing area-delay trade-off goals for each block – this is in line with the transistor-level optimization tool developed previously [48]. We use the results of prior work [48] to establish the optimal area-delay trade-off for 6-LUTs in a conventional island-style FPGA architecture with typical architectural parameters. The resulting 6-LUT delay serves as a point of reference for optimization for the circuits considered in this work: in the interest of maximizing area reduction while allowing performance to be maintained (ignoring the differences in cell counts between mapping to a conventional LUT and the logic elements proposed in this work), we attempt to match the delay of a 6-LUT while minimizing the area of each of the variants of the $MUX_4$ circuits. Transistor level modelling and optimizations were based on a predictive 22 nm high performance process [49], while the area model presented in prior work [48] was used to estimate the area of various circuit structures. With this methodology, we determined an area-delay optimal 6-LUT has an area of 930 minimum-width transistors, and a worst-case delay of 261 ps. For the $MUX_4$ cell and Dual $MUX_4$ cell, a minimum area and a minimum delay cell was created. The minimum area $MUX_4$ cell has an area of 95 minimum-width transistors and a delay of 204 ps. All transistors were minimum-width in this case, and the minimum area solution for this circuit was able to meet (and improve upon) the worst-case delay target of a 6-LUT. Similarly, the Dual $MUX_4$ cell has an area of 249 minimum-width transistors while meeting the worst-case delay requirement. However, we chose to use the minimum delay design for both the $MUX_4$ and Dual $MUX_4$ elements for the rest of the study as there is not a significant increase in area over the minimum area design.

Although the modelling was performed in a 22 nm process, the standard VPR architecture we use has all parameters (routing delays, crossbar delays, etc.) scaled to a 40 nm process. In this standard VPR architecture, parameters are compounded from a multitude of sources, some also in other lithographic processes, and subsequently scaled to 40 nm. Likewise, we linearly scale our delays by comparing the worst case delays of our 22 nm 6-LUT (261 ps) and the 6-LUT in the standard architecture (398 ps). The delays for each design after scaling to 40 nm are shown in Table 2.1.
### Table 2.1: Logic element transistor models with area given in minimum-width transistor area (MWTA) and Delays scaled for a 40 nm process.

<table>
<thead>
<tr>
<th>Logic Element Design</th>
<th>Area (MWTA)</th>
<th>% 6-LUT Area</th>
<th>Scaled Max. Delay (ps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUX4 Min. Area</td>
<td>95</td>
<td>10.2%</td>
<td>311</td>
</tr>
<tr>
<td>MUX4 Min. Delay</td>
<td>108</td>
<td>11.6%</td>
<td>248</td>
</tr>
<tr>
<td>Dual MUX4 Min. Area</td>
<td>249</td>
<td>26.7%</td>
<td>398</td>
</tr>
<tr>
<td>Dual MUX4 Min. Delay</td>
<td>255</td>
<td>27.4%</td>
<td>375</td>
</tr>
<tr>
<td>6-LUT</td>
<td>930</td>
<td>100.0%</td>
<td>398</td>
</tr>
</tbody>
</table>

FPGA Area Model

Although determining the area of a MUX4 element relative to a 6-LUT is important, we need to also examine global FPGA area considering the number of CLB tiles, area overheads within the CLB and routing area per CLB. Throughout this work, global FPGA area was estimated assuming that, per tile, 50% of the area is inter- and intra-cluster routing, 30% of the area is used for LUTs, and 20% for registers and other miscellaneous logic, following Anderson and Wang [36] and a private communication [50]. It is important to note that this 50%-30%-20% model is an estimate based on a traditional full FPGA design where-by the routing and internal CLB crossbars are optimized towards 6-LUTs. Production of an optimized FPGA utilizing our new MUX4 elements would surely change said model. However, optimizing the entire routing architecture towards our MUX4 variants, measuring the routing architecture, and closing the loop by creating a more accurate model is out of the scope of this work.

Using this model we can make some observations about the hybrid CLB architecture. The 30% that normally would account for ten 6-LUT logic elements within the tile is now split between the smaller MUX4 elements and 6-LUTs. For example, in a 3 MUX4 : 7 6-LUT architecture, the area relative to the reference area model can be estimated by deducing the LogicChange% = (3 × 0.116 + 7)/10 (3 MUX4s each at 0.116 the area of a 6-LUT and 7 6-LUTs), and multiplying LogicChange% × 30% = 22.0% of total FPGA area. If routing and miscellaneous area were held constant, our overall architecture area is Area3:7 = 50% + 20% + 22.0% = 92.0% of the reference area – 8% area savings. However, this is the maximum area savings and it can only be realized by circuits that have a natural (i.e. inherent) MUX4:LUT ratio greater than or equal to the architecture ratio. And, since any function that can be mapped to a MUX4 element can also be mapped into a 6-LUT, all excess MUX4 functions can be mapped to 6-LUTs. If the natural MUX4:LUT ratio of the circuit is less than the architecture ratio, additional CLBs will be required to supply more LUTs. Additionally, the number of CLBs may also increase during CLB packing (CLBChange%) and routing demand may increase post placement and routing (RoutingChange%). In general, the model used to estimate area relative to the baseline 6-LUT.
only architecture (non-fracturable or fracturable) is as follows:

\[
\text{Area\%} = \text{CLBChange\%} \times (50\% \times \text{RoutingChange\%} + 30\% \times \text{LogicChange\%} + 20\%) \tag{2.1}
\]

Using this model, it is useful to calculate how many additional CLBs can be tolerated for our new architectures. Again, consider a 3:7 \textit{MUX}_4\text{-LUT} architecture. Disregarding packing, placement, and routing effects:

\[
\text{NumCLB}_{3:7} \leq \frac{1}{\text{Area}_{3:7}} \times \text{NumCLB}_{\text{LUT}} \leq 1.08 \times \text{NumCLB}_{\text{LUT}} \tag{2.2}
\]

This means that an area win can only be achieved if the number of CLBs needed to implement circuits in a hybrid 3:7 architecture is less than \(1.08\times\) the number needed for a traditional LUT-only architecture.

Similarly, the calculation is performed for the fracturable architecture with the larger \textit{Dual MUX}_4\text{} element. A full table for all architectures showing the architectural minimum area and tolerable CLB increase is shown in Table 2.2.

<table>
<thead>
<tr>
<th>Arch. 1:9 % Area</th>
<th>Arch. 2:8 % Area</th>
<th>Arch. 3:7 % Area</th>
<th>Arch. 4:6 % Area</th>
<th>Arch. 5:5 % Area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Frac. 97.3</td>
<td>102.7</td>
<td>94.7</td>
<td>105.6</td>
<td>92.0</td>
</tr>
<tr>
<td>Frac. 97.8</td>
<td>102.2</td>
<td>95.6</td>
<td>104.6</td>
<td>93.5</td>
</tr>
</tbody>
</table>

Table 2.2: Per architecture, minimum relative architectural percentage area and tolerable percentage CLB increase assuming constant routing demand.

### 2.4 Technology Mapping using ABC

ABC [41] was used for technology mapping, with modifications that allow for \textit{MUX}_4\text{}-embeddable function identification and custom mapping. The internal data structure used within ABC is an \textit{AND}-inverter graph (AIG), where the logic circuit is represented using 2-input \textit{AND} gates with inverters. \textit{Priority Cuts} mapping in ABC (invoked with the \texttt{if} command) [51] was modified to perform our custom technology mapping. This mapper traverses the AIG from primary inputs to primary outputs finding intermediate mappings for internal nodes and finally the primary outputs, using a dynamic programming approach. The priority cuts mapper performs multiple passes on the AIG to find the best cut per node. For depth-
oriented mapping, the mapper first prioritizes mapping depth then optimizes for area discarding cuts whose selection would increase the overall depth of the mapped network.

Based on this standard mapper, two mapper variants were produced and evaluated. The first variant, NaturalMux, evaluates and identifies internal functions that are MUX4-embeddable, agnostic of the target architecture; i.e. this flow uses the default priority cuts mapping and performs a post-processing step to identify MUX4-embeddable functions. From this mapping, we can evaluate what area savings are possible without any mapper changes. The second variant MuxMap, area-weights the MUX4-embeddable cuts relative to 6-LUT cuts, thereby establishing a preference for selection/creation of MUX4-embeddable solutions.

In all mapper variants, cuts that are MUX4-embeddable need to be identified, meaning that we must determine whether the logic function implied by such cuts can be implemented in a MUX4 logic element. For LUTs that are identified as MUX4-embeddable, we “tag” them in the mapped network written out by ABC so that VPR is able to pack these elements into the hybrid-CLB architectures. The identification function essentially performs a 2-level Shannon decomposition for all combinations of select inputs – a maximum of $C_6^2$ times for a cut of size 6. For a logic function $f$, let $\text{Inputs}(f)$ represent its variable set, $\{x_0, \ldots, x_i\}$, and let $f_{x_i x_j}$ represent the Shannon co-factor of $f$ with respect to its variables $x_i$ and $x_j$. Logic function $f$ can be implemented in a MUX4 if and only if:

$$|\text{Inputs}(f)| \leq 3 \quad (2.3)$$

or

$$\exists x_i, x_j \in \text{Inputs}(f) \text{ such that}$$

$$|\text{Inputs}(f_{x_i x_j})| \leq 1,$$

$$|\text{Inputs}(f_{x_i x_j})| \leq 1,$$

$$|\text{Inputs}(f_{x_i x_j})| \leq 1,$$

$$|\text{Inputs}(f_{x_i x_j})| \leq 1$$

$$|\text{Inputs}(f_{x_i x_j})| \leq 1$$

$$|\text{Inputs}(f_{x_i x_j})| \leq 1$$

$$|\text{Inputs}(f_{x_i x_j})| \leq 1$$

That is, any function up to three inputs can be implemented in a MUX4, and for functions with four or more inputs, there must exist two variables such that the Shannon co-factors with respect to such variables have one or fewer inputs.
Likewise conditions must hold true for $MUX_2$-embeddability using a 1-level Shannon decomposition. Note that a 1-level Shannon decomposition technique has previously been leveraged for mapping into asymmetric-LUT architectures [37]. Any function up to two inputs can be implemented in a $MUX_2$. The only three input function that can be implemented in a $MUX_2$ is a 2-to-1 multiplexer with optional data input inversion. This limited flexibility was quickly seen with the implementation of a $MUX_4$ that was fracturable into two $MUX_2$s. These $MUX_2$s do not provide ample function diversity to justify the overheads of getting the signals into and out of the hybrid CLB and led to the creation of the Dual $MUX_4$.

Each of the select input(s) and data inputs to the $MUX_4$ element are classified by the mapper on a pin-by-pin basis so that accurate pin-correct packing and routing can be performed in VPR – all multiplexer inputs are fully modelled for wholly accurate $MUX_4$ input mapping.

### 2.4.1 NaturalMux

NaturalMux mapping invokes the standard priority cuts mapper. Following mapping, we use the preceding approach to determine if the LUT logic functions in the mapping are $MUX_4$-embeddable. This is needed so we can identify which LUTs are $MUX_4$-embeddable in the subsequent packing stage.

### 2.4.2 MuxMap

In default ABC technology mapping, each LUT has a unit area of 1.0. In our MuxMap approach, we use a lower weight for the cases where logic functions are $MUX_4$-embeddable. Following the area model where 50% of an FPGA tile area is routing, 30% is 6-LUTs and 20% is miscellaneous circuitry (FFs + other), we can derive the weight of a $MUX_4$ element versus a 6-LUT. Dividing an FPGA tile into ten sub-tiles that contain a single 6-LUT plus the 6-LUT’s associated routing and miscellaneous circuitry, the 6-LUT or logic portion of a sub-tile is 3% and the miscellaneous circuitry and routing is 7% of a complete tile. Recall from Section 2.3.4 that a $MUX_4$ element consumes 15% of the area of a 6-LUT. Therefore, the area of a sub-tile with a $MUX_4$ is 7.45% of the entire tile - that is, 7% routing and miscellaneous circuitry area plus $15\% \times 3\%$ logic area. The area ratio of a sub-tile with a $MUX_4$ versus a sub-tile with a 6-LUT would be roughly $7.45\%/10\% = 0.745$ (assuming the routing and other circuitry is held constant). Following this reasoning, we weight $MUX_4$s conservatively at 80% of a 6-LUT during technology mapping. Experimental results shown in Section 2.6.1 show that this is a reasonable choice.
2.4.3 Select Mapping

Depending on the circuit, *NaturalMux* or *MuxMap* may be preferred. In Select Mapping, the circuit is first mapped using *NaturalMux*. Following from the discussion in Section 2.3.4, we know that if a circuit’s *MUX4*:LUT ratio is higher than the architectural ratio, maximum area reductions are realized. So, if the *Natural* ratio of the circuit is higher than our target architectural ratio, we use this mapping. Otherwise, if the *Natural* ratio is lower than the architectural ratio, we re-run mapping with the *MuxMap* mapper to encourage the selection of more *MUX4*-embeddable logic elements. Note that technology mapping run-time is a small fraction of that required for placement and routing.

2.5 Modelling using VPR

VPR was used to perform architectural evaluation. The standard ten 6-LUT CLB architecture in 40 nm included with the VPR distribution was used for baseline modelling [52]. The hybrid CLBs shown in Figure 2.4 and Figure 2.5 were modelled using the XML-based VPR architectural language [52]. The snippet from the architecture file for the physical block hardened *MUX4* element is shown in Figure 2.6 – this code specifies a *MUX4* as a 6-input 1-output blackbox to VPR. Additionally, since all *MUX4*s can also be mapped to 6-LUTs, an additional *mode* was added to the 6-LUT physical block, shown in Figure 2.7. The *mode* concept allows the VPR packer to pack LUTs into LUTs (as usual), but also enables *MUX4*s to be packed into LUTs. Architectures with CLBs having *MUX4*:LUT ratios from 1:9 to 5:5 were created from the baseline 40 nm architectures with delays obtained through circuit simulations of the *MUX4* variants.

```xml
<pb_type name='mux4hard' blif_model='.' subckt 'mux4' num_pb='1'>
  <input name='s' num_pins='2'/>
  <input name='in' num_pins='4'/>
  <output name='out' num_pins='1'/>
</pb_type>
```

Figure 2.6: *MUX4* logic element model.

Importantly, we made minor modifications to the VPR packing algorithm [52] itself so that *MUX4* netlist elements are preferred to be packed into *MUX4* logic elements in the architecture (while limiting packing *MUX4* netlist elements into LUTs). The modifications involved changing the attraction function during CLB packing. One change was to ensure that the logic functions that were *MUX4* embeddable were preferentially packed into a physical *MUX4* element and not into a LUT. Another was to apply a negative weight on *MUX4*-embeddable functions when the current CLB’s physical *MUX4* elements are all occupied - also preventing *MUX4*-embeddable functions from being placed into LUTs. Without this,
2.6 Experimental Evaluation

To determine the benefits of these new architectures, evaluation was performed for each architecture using multiple benchmark suites and mapping schemes. Within these new architectures, no carry-chains were modelled. Two benchmarks suites were used to evaluate our hybrid architectures: VTR7 [52], and CHStone [42]. Over the non-fracturable and fracturable architecture families, two sets of experiments were performed using the NaturalMux and MuxMap mapping schemes. After both mappings were performed, the Select mapping was performed as described in Section 2.4.3.

While the VTR7 benchmarks are input to our flow as Verilog, the CHStone benchmarks are input as C. LegUp 3.0 [43] was used for C-to-verilog HLS for these CHStone benchmarks. Differing from our preliminary work [26] (that had two separate tool flows) modifications were made to both LegUp and ODIN-II tools so that CHStone benchmarks could be carried through the typical full VTR flow (ODIN-II, ABC, VPR) using ODIN-II for front-end synthesis. In the preliminary work [26], Altera Quartus II is used for front-end synthesis thus producing an Altera-specific mapping for DSP blocks, Memory blocks, and Carry Chains - all of which are not supported by standard VPR architectures. This restricted us to only be able to make post mapping estimates. In this new tool-flow, ODIN-II is able to synthesize the Verilog output from Leg-Up. ODIN-II thereby produces compatible DSP and Memory blocks for our modified standard FPGA architectures, allowing for post place and route results (previously non-existing).

Lastly, the mcml and LU32PEEng benchmarks were omitted from the VTR7 benchmark suite due
to excessive runtime.

2.6.1 Mapping

Using ABC, we performed technology-independent optimization using the standard resyn2 script included with the ABC distribution \[41\]. Then, we performed technology mapping with the priority cuts mapper (\textit{if} command), targeting a LUT size of 6. Our modified priority cuts mapper was invoked using the \textit{if} command with a cut set size of 32 also limiting the maximum cut size to 6. The initial unaltered \textit{Baseline} mapper was run without any mapping to MUX4s, and then \textit{NaturalMux} mapping was performed for the identification of \textit{MUX4-Embeddable} functions. Our second mapper \textit{MuxMap}, seeks to improve mapping by reducing the area cost of a MUX4-embeddable function within priority cuts mapping.

After mapping with both mappers, we computed the ratio of the number of MUX4-embeddable logic elements to the total number of logic elements – i.e. the MUX4 percentage. Based on this ratio for each circuit, we projected the area benefits of hybrid architectures 1:9 to 5:5 MUX4:LUT ratios in the non-fracturable architectures. The area projections were made assuming complete (full) CLB packing, and the tile area breakdown described in Section 2.3.4 (assuming no routing area change). The results are shown in Table 2.3 and demonstrate the maximum area savings to be had for each benchmark since these projections ignore the effects of packing, placement and routing constraints. For the fracturable architectures, this prediction is made difficult since pairing of functions to be packed into fractured logic elements is largely determined by pin-sharing on the logic element inputs – a packing constraint. No predictions are made for the fracturable architectures, however full packed, placed and routed results for the fracturable architectures are discussed in Section 2.6.3.

\textit{MuxMap} improves mapping by increasing the number of MUX4-embeddable logic functions by reducing the area cost of a MUX4-embeddable function, thereby encouraging the mapper to create more MUX4-embeddable functions, increasing the MUX4-embeddable ratio of each circuit overall. However, improvements in the ratio may come at the cost of a higher total number of logic elements. If there is a significant increase in logic elements (greater than the limits shown in Table 2.2), no area savings may be seen. Figure 2.8 shows a sweep of the weighting of MUX4-embeddable logic elements from 0% cost to 100% cost of a 6-LUT over the VTR7 suite; projected area is shown on the vertical axis (normalized to a traditional LUT-based architecture). Area is minimized for more aggressive architectures (5:5 and 4:6) around a weighting of 50-60%, whereas less aggressive architectures (3:7, 2:8, 1:9) do not benefit at all from a reduced weighting. This can be seen as the minimum area is for a weighting of 100%.
Figure 2.8: *MuxMap* varying the weight from zero area to equal 6-LUT area for different *MUX4*:LUT architectures. A weight equal to LUT area is equivalent to the *NaturalMux* mapping. As the *MUX4* area weighting is increased for some architecture ratios (e.g. 5:5 and 4:6), an increase in relative area is seen due to unbalanced circuit and architecture *MUX4*:LUT ratios.

As the number of logic elements grows, packing, placement and routing effects play a greater roll in the final circuit area. In the remainder of the work, a weighting of 80% was chosen for *MuxMap* as this gave a good balance of additional *MUX4*-embeddable logic elements. Lower weightings result in many additional logic elements, exacerbating the losses due to packing, placement and routing.

The left-hand side of Table 2.3 shows the projected area results for *NaturalMux* mapping, as well as the baseline statistics of each benchmark in the two benchmark suites. The right-hand side shows projected area results for *MuxMap* mapping with a weighting of 0.8. In both mappers, we show projected results for architectures with ratios ranging from 1:9 to 5:5. Again, Table 2.2 shows the lower-bound architectural areas relative to a traditional 6-LUT-based architecture for the configurations tested and this lower bound can only be achieved when the *Natural* *MUX4*-embeddable ratio exceeds the architecture ratio. When the *Natural* ratio is lower than the architecture ratio, more CLBs are required, increasing area. Table 2.3 shows the lower bound on area for each benchmark and non-fracturable architecture since this projection assumes perfect packing, placement, and routing of each benchmark on each architecture.
Looking at the left half of Table 2.3, the baseline number of logic elements (LEs) along with the *Natural MUX4*-embeddable ratio is given along with projected areas for each architecture for *NaturalMux*.

VTR7 benchmarks are projected to perform well, with a high *Natural MUX4*-embeddable ratio of 40% over all benchmarks. For the VTR7 benchmarks, a 3:7 architecture seems to be the best architecture for *NaturalMux* mapping, yielding \( \sim 7\% \) projected area reduction post-mapping. *MuxMap* shows an increase in *MUX4*-embeddable ratio to 46% over all benchmarks; however, the increase in LE count associated with *MuxMap* leads to worse projected area results per architecture vs. *NaturalMux*, except for the 4:6 and 5:5 architectures. Employing the *Select* mapping scheme yields marginally better area reduction for the best architecture ratio, 3:7 (\( \sim 1\% \) above the architectural minimum).

The CHStone benchmarks all show large percentages of *MUX4*-embeddable functions using the *NaturalMux* mapper, 55%. In fact, the percentage of *MUX4*-embeddable functions is so large that in nearly all cases, the architectural minimum area is reached. In this case, the *MuxMap* mapper and *Select* schemes have no further improvements. From these post-mapping estimates, it seems as though even a very aggressive 5:5 architectural ratio for the CHStone suite may be best as we reach the architectural minimum area for non-fracturable 5:5 architecture. This variation between benchmark suites is interesting and the CHStone benchmarks suiting more aggressive architectures could be a function of how LegUp-HLS transforms the CHStone benchmarks, written in C, to hardware. Again, we see that different architectural conclusions can be made based on the benchmark circuits employed in an architectural study [53].

We can also examine how the weighting of the *MuxMap* mapper affects the used input-size distribution for LUTs in the circuits versus the *NaturalMux* mapper. The input-size distributions for the *NaturalMux* and *MuxMap* mappings are shown in Figures 2.9 and 2.10 for VTR7, and CHStone benchmarks, respectively. Each bar in the distribution shows the portion of logic elements (with a given number of used inputs) that are *MUX4*-embeddable (in light-blue). The VTR suite has only a small percentage of functions with more than 3 inputs that are *Naturally MUX4*-embeddable (\( \sim 10\% \)). With *MuxMap*, the percentage of larger five and six-input functions is somewhat reduced with these functions being implemented mostly as three or four-input *MUX4*-embeddable functions. Looking at the 6-input functions specifically, we also observe that very few are *MUX4*-embeddable. Recall that the only 6-input logic function that is *MUX4*-embeddable is a 4-to-1 multiplexer (with possible data input inversions). Two and three-input *MUX4*-embeddable functions are also quite abundant in both VTR7 and CHStone benchmarks, comprising the majority of all *MUX4*-embeddable functions. Concerning these small functions, note that over 35% of functions in the VTR7 circuits and 40% in CHStone have three or fewer inputs (Figure 2.9), which bodes well for the proposed hybrid architectures. Additionally, in both bench-
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline &amp; NaturalMux</th>
<th>MuxMap - 0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgm</td>
<td>31480</td>
<td>25</td>
</tr>
<tr>
<td>mkPktMerge</td>
<td>6036</td>
<td>22</td>
</tr>
<tr>
<td>boundtop</td>
<td>2932</td>
<td>43</td>
</tr>
<tr>
<td>ch_intrinsics</td>
<td>382</td>
<td>29</td>
</tr>
<tr>
<td>dfdiv</td>
<td>466</td>
<td>33</td>
</tr>
<tr>
<td>diffeq</td>
<td>317</td>
<td>32</td>
</tr>
<tr>
<td>LUSPEEng</td>
<td>21927</td>
<td>43</td>
</tr>
<tr>
<td>mkDelayWorker32B</td>
<td>5467</td>
<td>43</td>
</tr>
<tr>
<td>mkSMAdapter4B</td>
<td>232</td>
<td>74</td>
</tr>
<tr>
<td>or1200</td>
<td>1995</td>
<td>32</td>
</tr>
<tr>
<td>sha</td>
<td>2215</td>
<td>35</td>
</tr>
<tr>
<td>stereoVision0</td>
<td>11232</td>
<td>76</td>
</tr>
<tr>
<td>stereoVision1</td>
<td>10059</td>
<td>69</td>
</tr>
<tr>
<td>stereoVision2</td>
<td>28596</td>
<td>52</td>
</tr>
<tr>
<td>stereoVision3</td>
<td>173</td>
<td>51</td>
</tr>
<tr>
<td>Geomean</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Select Geomean</th>
<th>CHStone</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>adpcm</td>
<td>13797</td>
</tr>
<tr>
<td></td>
<td>aec</td>
<td>14265</td>
</tr>
<tr>
<td></td>
<td>blowfish</td>
<td>15584</td>
</tr>
<tr>
<td></td>
<td>dfadd</td>
<td>6523</td>
</tr>
<tr>
<td></td>
<td>dfdiv</td>
<td>7292</td>
</tr>
<tr>
<td></td>
<td>dfmul</td>
<td>4381</td>
</tr>
<tr>
<td></td>
<td>dsin</td>
<td>19714</td>
</tr>
<tr>
<td></td>
<td>gsm</td>
<td>12580</td>
</tr>
<tr>
<td></td>
<td>jpeg</td>
<td>36400</td>
</tr>
<tr>
<td></td>
<td>mips</td>
<td>4736</td>
</tr>
<tr>
<td></td>
<td>motion</td>
<td>5180</td>
</tr>
<tr>
<td></td>
<td>sha</td>
<td>13607</td>
</tr>
<tr>
<td></td>
<td>Geomean</td>
<td>-</td>
</tr>
</tbody>
</table>

Select Geomean | - | 97.3 | 94.7 | 92.0 | 89.4 | 86.9

CHStone
|                 | - | 97.3 | 94.7 | 92.0 | 89.4 | 86.9

Table 2.3: Post mapping area estimate for VTR7 and CHStone benchmarks assuming complete CLB packing and no increase in routing demand.
Figure 2.9: Input-size distribution of VTR7 benchmarks with \( MUX_4 \)-embeddable logic elements in blue (light grey) and LUTs in purple (dark grey).

Figure 2.10: Input-size distribution for CHStone benchmarks with \( MUX_4 \)-embeddable cuts in blue (light grey) and LUTs in purple (dark grey).

mark suites \textit{MuxMap} encourages a decrease in 5-input elements and an increase in \( MUX_4 \)-embeddable 3-input elements.

### 2.6.2 Non-Fracturable Architectures: Packing, Place & Route

Of course, the previous post-mapping results do not consider the possible impact of the addition of \( MUX_4 \) elements to the FPGA architecture on packing, placement and routability. Therefore, the mapped netlists were packed, placed, and routed using VPR into the architectures discussed in Section \ref{sec:2.5}. The final area estimates in Tables \ref{tab:2.4} and \ref{tab:2.5} show the area reduction relative to the baseline mapper and a traditional \textit{non-fracturable} 6-LUT architecture. We use the area equation shown in Section \ref{sec:2.3} for these estimates. VPR was configured to find minimum channel width with timing driven
routing. For all benchmarks, five different placement seeds were used to account for the random effects of placement on final routing. For CLBChange%, we use the change in packed CLBs over the baseline. For RoutingChange%, we use the change in routing area of the minimum channel width per logic tile (measured in min-width transistors) reported by VPR. Finally, we compute LogicChange% according to the chosen hybrid CLB architecture ratio – this is constant for a given architecture and represents the physical reduction in logic element area.

Tables 2.4 and 2.5 show the results for each benchmark using the NaturalMux and MuxMap mapping schemes respectively, over all non-fracturable architectures. In both tables, the baseline CLB count is shown with the natural MUX4-embeddable logic element ratio. For each architecture, the percentage change in CLBs and routing with respect to the baseline 6-LUT-only architecture is shown with area based on the model in Section 2.3.4. For each benchmark suite, we take the geomean area across all benchmarks in the suite. Each of these final mean relative area numbers is shown in bold. Table 2.5 has additionally a row showing the Select geomean, which is the result of applying the Select scheme between both mappers for each architecture.

Overall, we see increases in packed CLBs for both benchmark suites, but for most architectures, this gain is offset by the smaller hybrid CLB area. Looking at the minimum geomean areas in Table 2.4, we can see that the best architecture suited towards NaturalMux mapping is the 2:8 for VTR7 and 4:6 for CHStone. But for the MuxMap mapping scheme in Table 2.5, VTR7 performs best on a 3:7 architecture and CHStone performs best on a 5:5 architecture. However, the best results overall are shown with Select. For Select mapper, VTR7 on a 3:7 architecture is the best with \( \sim 4\% \) area reduction, but it is down four percentage points from the post-map estimate. CHStone performs best in a 4:6 architecture at \( \sim 9\% \) area reduction. The post-mapping projections again suggested three percentage points better area for the 5:5 architecture.

For the non-fracturable architectures, a ratio of 3:7 seems fitting as we see gains for both benchmark suites - though not full gains for CHStone.

### 2.6.3 Fracturable Architectures: Packing, Place, and Route

The final area estimates in Tables 2.6 and 2.7 show the area reduction relative to the baseline mapper and a traditional fracturable 6-LUT architecture. The same experimental methodology used for non-fracturable architectures is used for the fracturable architectures.

Fracturable architectures overall show vastly differing performance compared to non-fracturable architectures. Using NaturalMux mapping shown in Table 2.6, the best architectures are the 1:9 archi-
Table 2.4: Non-Fracturable Architecture: Post Place & Route results showing architectural sweeps between zero (baseline) to five \textsc{MUX4} logic elements out of ten, for fracturable hybrid-CLB architectures, using the NaturalMux mapping scheme. All benchmarks were averaged over five placement seeds.
<table>
<thead>
<tr>
<th>Natural MUX4 Ratio</th>
<th># CLBs</th>
<th>Arch. 1:9 % CLBs</th>
<th>% Rtg</th>
<th>% Area</th>
<th>Arch. 2:8 % CLBs</th>
<th>% Rtg</th>
<th>% Area</th>
<th>Arch. 3:7 % CLBs</th>
<th>% Rtg</th>
<th>% Area</th>
<th>Arch. 4:6 % CLBs</th>
<th>% Rtg</th>
<th>% Area</th>
<th>Arch. 5:5 % CLBs</th>
<th>% Rtg</th>
<th>% Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgm</td>
<td>0.25</td>
<td>3599</td>
<td>101.2</td>
<td>100.4</td>
<td>98.7</td>
<td>101.3</td>
<td>108.8</td>
<td>100.3</td>
<td>101.3</td>
<td>142.8</td>
<td>114.8</td>
<td>110.3</td>
<td>150.0</td>
<td>126.1</td>
<td>132.0</td>
<td>152.7</td>
</tr>
<tr>
<td>blobmerge</td>
<td>0.22</td>
<td>604</td>
<td>101.0</td>
<td>101.8</td>
<td>99.2</td>
<td>101.0</td>
<td>108.9</td>
<td>100.1</td>
<td>101.0</td>
<td>106.3</td>
<td>102.4</td>
<td>121.7</td>
<td>103.7</td>
<td>111.0</td>
<td>142.1</td>
<td>108.8</td>
</tr>
<tr>
<td>boundtop</td>
<td>0.43</td>
<td>300</td>
<td>100.3</td>
<td>91.0</td>
<td>93.2</td>
<td>100.3</td>
<td>93.8</td>
<td>91.9</td>
<td>100.3</td>
<td>100.4</td>
<td>92.6</td>
<td>100.3</td>
<td>101.1</td>
<td>94.2</td>
<td>101.0</td>
<td>115.4</td>
</tr>
<tr>
<td>ch_intrinsics</td>
<td>0.29</td>
<td>39</td>
<td>100.0</td>
<td>100.0</td>
<td>97.3</td>
<td>100.0</td>
<td>100.4</td>
<td>94.9</td>
<td>102.6</td>
<td>103.1</td>
<td>96.0</td>
<td>117.9</td>
<td>102.7</td>
<td>140.0</td>
<td>94.9</td>
<td>118.7</td>
</tr>
<tr>
<td>diffeq1</td>
<td>0.33</td>
<td>47</td>
<td>102.1</td>
<td>106.3</td>
<td>102.6</td>
<td>102.1</td>
<td>107.3</td>
<td>100.5</td>
<td>102.1</td>
<td>105.9</td>
<td>97.0</td>
<td>102.1</td>
<td>105.0</td>
<td>93.8</td>
<td>114.9</td>
<td>104.2</td>
</tr>
<tr>
<td>diffeq2</td>
<td>0.32</td>
<td>32</td>
<td>103.1</td>
<td>96.3</td>
<td>98.5</td>
<td>103.1</td>
<td>97.8</td>
<td>96.5</td>
<td>103.1</td>
<td>102.5</td>
<td>96.2</td>
<td>112.5</td>
<td>103.2</td>
<td>104.4</td>
<td>115.4</td>
<td>126.9</td>
</tr>
<tr>
<td>LUSPEng</td>
<td>0.43</td>
<td>2579</td>
<td>100.0</td>
<td>101.8</td>
<td>98.8</td>
<td>100.0</td>
<td>103.1</td>
<td>96.8</td>
<td>100.5</td>
<td>110.3</td>
<td>97.7</td>
<td>100.6</td>
<td>124.8</td>
<td>100.6</td>
<td>143.3</td>
<td>109.0</td>
</tr>
<tr>
<td>mkDelayWorker32B</td>
<td>0.43</td>
<td>550</td>
<td>101.3</td>
<td>100.4</td>
<td>98.8</td>
<td>101.3</td>
<td>103.4</td>
<td>97.6</td>
<td>101.3</td>
<td>103.6</td>
<td>95.1</td>
<td>104.7</td>
<td>113.2</td>
<td>100.5</td>
<td>115.6</td>
<td>105.8</td>
</tr>
<tr>
<td>mkPktMerge</td>
<td>0.74</td>
<td>24</td>
<td>100.0</td>
<td>95.9</td>
<td>95.3</td>
<td>100.0</td>
<td>98.5</td>
<td>93.9</td>
<td>100.0</td>
<td>98.0</td>
<td>91.0</td>
<td>100.0</td>
<td>90.9</td>
<td>89.8</td>
<td>100.0</td>
<td>96.5</td>
</tr>
<tr>
<td>or1200</td>
<td>0.38</td>
<td>298</td>
<td>102.3</td>
<td>101.4</td>
<td>100.4</td>
<td>102.3</td>
<td>102.2</td>
<td>98.1</td>
<td>102.3</td>
<td>100.8</td>
<td>94.6</td>
<td>103.7</td>
<td>110.4</td>
<td>98.1</td>
<td>109.7</td>
<td>106.2</td>
</tr>
<tr>
<td>raygentop</td>
<td>0.31</td>
<td>240</td>
<td>102.5</td>
<td>100.7</td>
<td>100.1</td>
<td>102.5</td>
<td>102.0</td>
<td>98.1</td>
<td>102.5</td>
<td>99.3</td>
<td>94.0</td>
<td>103.8</td>
<td>100.0</td>
<td>92.7</td>
<td>105.0</td>
<td>106.5</td>
</tr>
<tr>
<td>sha</td>
<td>0.35</td>
<td>225</td>
<td>100.4</td>
<td>104.3</td>
<td>100.0</td>
<td>100.4</td>
<td>104.4</td>
<td>97.3</td>
<td>103.6</td>
<td>107.9</td>
<td>99.4</td>
<td>107.6</td>
<td>113.1</td>
<td>103.2</td>
<td>126.7</td>
<td>119.3</td>
</tr>
<tr>
<td>stereovision0</td>
<td>0.76</td>
<td>1494</td>
<td>101.7</td>
<td>104.3</td>
<td>101.2</td>
<td>101.7</td>
<td>102.6</td>
<td>97.6</td>
<td>101.7</td>
<td>103.0</td>
<td>95.1</td>
<td>101.7</td>
<td>105.9</td>
<td>93.9</td>
<td>101.7</td>
<td>104.8</td>
</tr>
<tr>
<td>stereovision1</td>
<td>0.69</td>
<td>1356</td>
<td>101.0</td>
<td>100.0</td>
<td>97.5</td>
<td>100.1</td>
<td>102.7</td>
<td>96.2</td>
<td>101.0</td>
<td>103.2</td>
<td>93.7</td>
<td>100.1</td>
<td>105.0</td>
<td>92.0</td>
<td>100.1</td>
<td>105.0</td>
</tr>
<tr>
<td>stereovision2</td>
<td>0.52</td>
<td>3584</td>
<td>101.1</td>
<td>98.1</td>
<td>97.4</td>
<td>101.0</td>
<td>100.6</td>
<td>95.7</td>
<td>101.0</td>
<td>97.9</td>
<td>92.0</td>
<td>101.0</td>
<td>103.6</td>
<td>92.3</td>
<td>101.1</td>
<td>112.3</td>
</tr>
<tr>
<td>stereovision3</td>
<td>0.51</td>
<td>19</td>
<td>100.0</td>
<td>100.0</td>
<td>97.3</td>
<td>100.0</td>
<td>104.0</td>
<td>94.7</td>
<td>100.0</td>
<td>106.3</td>
<td>95.9</td>
<td>100.0</td>
<td>104.9</td>
<td>91.8</td>
<td>100.0</td>
<td>105.3</td>
</tr>
</tbody>
</table>

| Geomean           | -      | -               | 98.6  | -      | 96.9            | -      | -      | 96.5            | -      | -      | 99.2            | -      | -      | 100.5          | -      | -      |
| Select Geomean    | -      | -               | 97.5  | -      | 96.0            | -      | -      | 95.7            | -      | -      | 98.9            | -      | -      | 104.7          | -      | -      |

| CHStone          | -      | -               | 99.0  | -      | 96.9            | -      | -      | 94.4            | -      | -      | 91.9            | -      | -      | 91.4            | -      | -      |
| Select Geomean   | -      | -               | 97.3  | -      | 95.2            | -      | -      | 93.2            | -      | -      | 91.1            | -      | -      | 91.2            | -      | -      |

Table 2.5: Non-Fracturable Architecture: Post Place & Route results showing architectural sweeps between zero (baseline) to five MUX4 logic elements out of ten, for fracturable hybrid-CLB architectures using the MuxMap mapping scheme. All benchmarks were averaged over five placement seeds.
tecture for VTR7 and the 2:8 architecture for CHStone. Only up to \( \sim 2\% \) area savings are seen for NaturalMux mapping for fracturable architectures. Using MuxMap mapping (Table 2.7), the best architectures are the 1:9 architecture for VTR7 but now the 3:7 architecture for CHStone. Again, only up to \( \sim 2\% \) area savings are seen for MuxMap mapping for fracturable architectures. For Select mapping, we get marginal gains and show the best overall result for both VTR7 and CHStone benchmark suites. A 1:9 architecture looks best for VTR7 at an overall 2.1\% area reduction and a 2:8 or 3:7 looks best for the CHStone suite at 2.6\% and 2.7\% reduction respectively.

Recall that the fracturable 6-LUT architecture presented is very flexible – it can map two functions with up to 4 disjoint inputs each. This flexibility is quite powerful as seen with the reduced performance of the fracturable architectures versus the non-fracturable architectures. The Dual MUX4 element can only map some combinations of 3 and 4 input functions since 4 inputs to the logic element are shared (see Figure 2.3). The constraints (pin sharing) on the input pins of the Dual MUX4 restrict many pairs of functions, usually 2 to 4 inputs, to be mapped, whereas the fracturable LUT is able to map any pair of functions up to 4-inputs each.

Looking again at the function input distributions in Figures 2.9 and 2.10, we can see that over 50\% of the functions in each benchmark suite are 4-input or smaller. So, these 4-input or smaller functions are ideal candidates for mapping into the fracturable LUTs, allowing the fracturable LUT to perform very well in terms of logic density. Also from the distributions, there are not many 5 or 6 input function that can be mapped to MUX4 elements. So most of these elements also require LUTs leaving not many functions left to be mapped to Dual MUX4 elements.

2.6.4 Performance

Worst-case delay for both the 6-LUT and MUX4 variants was used to determine the longest delay and FMax. Worst-case was chosen to serve as a level field for each logic element, as both have dissimilar input delays. Table 2.8 shows the geomean FMax change for Select mapping for non-fracturable and fracturable architectures for each benchmark suite. By using the Select mapping results, we are taking the best mapping scheme based on area and then measuring performance. Over all tested architectures we observed between a 2\% improvement to a 7\% slowdown. The general trend seems to be that the more aggressive architectures see more slowdown especially for fracturable architectures.

For the non-fracturable architecture, the best area savings were seen in 3:7 and 4:6 ratios, compared to the baseline 6-LUT. The performance is relatively constant for the 3:7 architecture, while for the 4:6 architecture, the VTR7 benchmarks have a 2.5\% hit and the CHStone benchmarks have a 2.5\% gain.
<table>
<thead>
<tr>
<th>Baseline</th>
<th>Natural MUX Ratio</th>
<th># CLBs</th>
<th>% CLBs</th>
<th>% Rttg</th>
<th>Area</th>
<th>% CLBs</th>
<th>% Rttg</th>
<th>Area</th>
<th>% CLBs</th>
<th>% Rttg</th>
<th>Area</th>
<th>% CLBs</th>
<th>% Rttg</th>
<th>Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgm</td>
<td>0.25</td>
<td>2462</td>
<td>97.8</td>
<td>109.2</td>
<td>100.2</td>
<td>96.1</td>
<td>151.6</td>
<td>116.8</td>
<td>107.7</td>
<td>145.5</td>
<td>125.2</td>
<td>124.3</td>
<td>138.0</td>
<td>137.1</td>
</tr>
<tr>
<td>blobmerge</td>
<td>0.22</td>
<td>424</td>
<td>101.9</td>
<td>107.3</td>
<td>103.4</td>
<td>111.8</td>
<td>107.4</td>
<td>111.0</td>
<td>125.7</td>
<td>101.0</td>
<td>118.2</td>
<td>145.3</td>
<td>95.7</td>
<td>129.5</td>
</tr>
<tr>
<td>boundtop</td>
<td>0.43</td>
<td>187</td>
<td>100.0</td>
<td>102.8</td>
<td>99.2</td>
<td>100.5</td>
<td>114.8</td>
<td>103.6</td>
<td>101.6</td>
<td>118.5</td>
<td>104.4</td>
<td>113.9</td>
<td>123.1</td>
<td>117.1</td>
</tr>
<tr>
<td>ch_intrinsics</td>
<td>0.29</td>
<td>27</td>
<td>100.0</td>
<td>98.4</td>
<td>97.0</td>
<td>103.7</td>
<td>98.8</td>
<td>98.5</td>
<td>114.8</td>
<td>99.2</td>
<td>106.8</td>
<td>133.3</td>
<td>97.5</td>
<td>120.0</td>
</tr>
<tr>
<td>dffeq1</td>
<td>0.33</td>
<td>31</td>
<td>100.0</td>
<td>100.5</td>
<td>98.1</td>
<td>103.2</td>
<td>98.4</td>
<td>97.9</td>
<td>116.1</td>
<td>103.2</td>
<td>110.4</td>
<td>132.3</td>
<td>98.7</td>
<td>119.9</td>
</tr>
<tr>
<td>dffeq2</td>
<td>0.32</td>
<td>22</td>
<td>100.0</td>
<td>102.8</td>
<td>99.2</td>
<td>104.5</td>
<td>104.8</td>
<td>102.5</td>
<td>118.2</td>
<td>100.8</td>
<td>110.9</td>
<td>136.4</td>
<td>100.0</td>
<td>124.5</td>
</tr>
<tr>
<td>LUSTEEng</td>
<td>0.43</td>
<td>174</td>
<td>98.9</td>
<td>105.6</td>
<td>99.4</td>
<td>98.2</td>
<td>115.5</td>
<td>101.6</td>
<td>97.9</td>
<td>125.9</td>
<td>104.2</td>
<td>104.4</td>
<td>137.1</td>
<td>114.6</td>
</tr>
<tr>
<td>mKDelayWorker32B</td>
<td>0.43</td>
<td>354</td>
<td>100.3</td>
<td>99.1</td>
<td>97.7</td>
<td>101.7</td>
<td>107.5</td>
<td>101.1</td>
<td>106.5</td>
<td>118.6</td>
<td>109.5</td>
<td>117.5</td>
<td>117.9</td>
<td>117.8</td>
</tr>
<tr>
<td>mkPktMerge</td>
<td>0.74</td>
<td>13</td>
<td>100.0</td>
<td>105.0</td>
<td>100.3</td>
<td>100.0</td>
<td>101.9</td>
<td>96.6</td>
<td>107.7</td>
<td>102.7</td>
<td>102.1</td>
<td>100.0</td>
<td>105.0</td>
<td>93.8</td>
</tr>
<tr>
<td>mkSMAAdapter4B</td>
<td>0.32</td>
<td>127</td>
<td>100.0</td>
<td>102.1</td>
<td>98.9</td>
<td>102.4</td>
<td>105.9</td>
<td>100.9</td>
<td>111.0</td>
<td>107.4</td>
<td>107.9</td>
<td>127.6</td>
<td>108.0</td>
<td>121.6</td>
</tr>
<tr>
<td>or1200</td>
<td>0.38</td>
<td>196</td>
<td>100.5</td>
<td>99.7</td>
<td>98.2</td>
<td>103.1</td>
<td>104.7</td>
<td>101.0</td>
<td>110.7</td>
<td>106.1</td>
<td>106.9</td>
<td>125.0</td>
<td>104.9</td>
<td>117.2</td>
</tr>
<tr>
<td>raygentop</td>
<td>0.31</td>
<td>167</td>
<td>96.4</td>
<td>101.3</td>
<td>94.9</td>
<td>96.4</td>
<td>105.5</td>
<td>93.9</td>
<td>97.0</td>
<td>109.7</td>
<td>95.4</td>
<td>109.0</td>
<td>113.1</td>
<td>106.6</td>
</tr>
<tr>
<td>sha</td>
<td>0.35</td>
<td>162</td>
<td>100.6</td>
<td>103.6</td>
<td>100.2</td>
<td>101.2</td>
<td>115.5</td>
<td>104.7</td>
<td>112.3</td>
<td>123.3</td>
<td>118.1</td>
<td>129.6</td>
<td>120.1</td>
<td>131.3</td>
</tr>
<tr>
<td>stereovision0</td>
<td>0.76</td>
<td>899</td>
<td>96.4</td>
<td>102.2</td>
<td>95.4</td>
<td>94.2</td>
<td>106.3</td>
<td>93.1</td>
<td>92.2</td>
<td>108.2</td>
<td>90.0</td>
<td>91.1</td>
<td>112.0</td>
<td>88.6</td>
</tr>
<tr>
<td>stereovision1</td>
<td>0.69</td>
<td>842</td>
<td>96.2</td>
<td>100.8</td>
<td>94.5</td>
<td>93.7</td>
<td>103.0</td>
<td>91.0</td>
<td>90.7</td>
<td>104.3</td>
<td>86.8</td>
<td>92.5</td>
<td>102.7</td>
<td>85.7</td>
</tr>
<tr>
<td>stereovision2</td>
<td>0.52</td>
<td>2304</td>
<td>97.3</td>
<td>105.0</td>
<td>97.6</td>
<td>95.8</td>
<td>117.7</td>
<td>97.2</td>
<td>94.0</td>
<td>124.2</td>
<td>99.2</td>
<td>92.8</td>
<td>136.7</td>
<td>101.8</td>
</tr>
<tr>
<td>stereovision3</td>
<td>0.51</td>
<td>12</td>
<td>100.0</td>
<td>103.2</td>
<td>99.4</td>
<td>100.0</td>
<td>98.9</td>
<td>100.0</td>
<td>122.0</td>
<td>104.6</td>
<td>109.5</td>
<td>105.7</td>
<td>103.8</td>
<td>103.9</td>
</tr>
<tr>
<td>Geomean</td>
<td></td>
<td></td>
<td>98.4</td>
<td></td>
<td></td>
<td>112.3</td>
<td></td>
<td>105.5</td>
<td></td>
<td></td>
<td></td>
<td>112.3</td>
<td></td>
<td>105.5</td>
</tr>
</tbody>
</table>

**Table 2.6:** Fracturable Architecture: Post Place & Route results showing architectural sweeps between zero (baseline) to five MUX4 logic elements out of ten, for fracturable hybrid-CLB architectures, using the NaturalMux mapping scheme. All benchmarks were averaged over five placement seeds.
Table 2.7: Fracturable Architecture: Post Place & Route results showing architectural sweeps between zero (baseline) to five MUX\_1 logic elements out of ten, for fracturable hybrid-CLB architectures, using MuxMap mapping scheme. All benchmarks were averaged over five placement seeds.
For fracturable architectures, we saw the best area savings in a 1:9 architecture for VTR7 and for performance, we now see a 1% slowdown. For CHStone, based on area, the preferable architecture was a toss-up between a 2:8 and 3:7 architecture but here we see that a 2:8 architecture is preferable since we take a smaller performance hit of 1.5%.

Overall, the performance impact on the architectures that yielded the best area savings were minimal for fracturable architectures while some gains were seen for non-fracturable architectures.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>VTR</th>
<th>CHStone</th>
<th>VTR</th>
<th>CHStone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arch. 1:9</td>
<td>1.008x</td>
<td>1.007x</td>
<td>1.003x</td>
<td>0.975x</td>
</tr>
<tr>
<td>Arch. 2:8</td>
<td>1.010x</td>
<td>1.020x</td>
<td>1.012x</td>
<td>1.025x</td>
</tr>
<tr>
<td>Arch. 3:7</td>
<td>1.012x</td>
<td>1.025x</td>
<td>1.001x</td>
<td>1.000x</td>
</tr>
<tr>
<td>Arch. 4:6</td>
<td>0.990x</td>
<td>0.965x</td>
<td>0.959x</td>
<td>0.943x</td>
</tr>
<tr>
<td>Arch. 5:5</td>
<td>0.997x</td>
<td>0.985x</td>
<td>0.965x</td>
<td>0.956x</td>
</tr>
</tbody>
</table>

Table 2.8: Geomean FMax ratio for each architecture for Select mapping relative to the standard ten 6-LUT architecture for each benchmark suite. Here, higher change is better.

### 2.6.5 Discussion

The results show that relative to baseline non-fracturable LUT-based logic element architectures, the inclusion of hardened $MUX_4$s to form a hybrid LUT/$MUX_4$-based CLB provides an area benefit of up to 8.9% and a 2.5% increase in FMax (for the case of the CHStone circuits, 4:6 architecture and Select mapping). In essence, this result reaffirms conventional wisdom that LUTs are in a sense “over-engineered” for many logic functions that regularly occur in application circuits. For such functions, one can “get away with” using a logic element with reduced flexibility.

With respect to a baseline of fracturable LUT-based elements, the incorporation of $MUX_4$s offers at most a modest area benefit of 2.6% with a small performance hit of 1.5% (CHStone, with a 2:8 ratio). The notion of making LUTs fracturable is, in fact, a feature whose purpose is rooted in the over-engineered nature of non-fracturable LUTs: fracturable LUTs permit area recovery for implementing small logic functions. Thus, the underlying purpose of making LUTs fracturable is not orthogonal to the purpose of incorporating $MUX_4$s, and hence, it is perhaps unsurprising that $MUX_4$s provide a small benefit in this case. Nevertheless, we consider the results encouraging in that they point out that the LUT, while being the long-standing lynchpin of FPGA logic, may not be the only circuit structure capable of delivering high logic density. Other structures, such as $MUX_4$s, are worthy of consideration.
2.7 Summary

In this chapter, we have proposed a new hybrid CLB architecture containing \( MUX_4 \) hard multiplexer elements and shown techniques for efficiently mapping to these architectures. Weighting of \( MUX_4 \)-embeddable functions with our \textit{MuxMap} technique combined with a \textit{Select} mapping strategy provided aid to circuits with low \textit{Natural} \( MUX_4 \)-embeddable ratios. We also provided analysis of the benchmark suites post mapping, discussing the distribution of functions within each benchmark suite. From our first set of experiments with non-fracturable architectures, area reductions of up to 8\% were seen for a 4:6 \( MUX_4 \):LUT architecture in the CHStone suite with a 2:8 architecture most viable for the VTR suites with \( \sim 5\% \) area savings. Our second set of experiments with fracturable architectures showed that the flexibility of a fracturable LUT is very powerful, reducing the impact of the \( MUX_4 \) logic elements, yielding smaller \( \sim 2-3\% \) area savings over the VTR7 and CHStone benchmark suites with a less aggressive 2:8 and 1:9 architectures, respectively. Interestingly, we again found that different architectural conclusions can be made based on the benchmark circuits employed in an architecture study [53], since CHStone benchmarks generally preferred more aggressive \( MUX_4 \):LUT architecture ratios. The CHStone benchmarks being high-level synthesized with LegUp-HLS also showed marginally better performance and this could be due to the way LegUp performs HLS on the CHStone benchmarks themselves. Overall, the addition of \( MUX_4 \)s to FPGA architectures minimally impact FMax and show potential for improving logic-density in non-fracturable architectures and modest potential for improving logic-density in fracturable architectures.
Chapter 3

Coarse-Grained Reconfigurable Architectures

This chapter provides an overview of coarse-grained reconfigurable architectures. It gives a description of what constitutes a Coarse-Grained Reconfigurable Array (CGRA) as well as the benefits and challenges of such architectures. It overviews common architectural designs and concepts and also reviews some key CGRA projects with their corresponding architectures and tooling.

3.1 Background

We introduce CGRAs within the computing landscape and aim to illustrate the varied nuances between architectures, while highlighting key concepts related to CGRAs.

3.1.1 What is a CGRA?

A CGRA is a class of reconfigurable architecture where computation is performed via an assortment of functional units connected through reconfigurable interconnect with some distributed temporary storage mechanisms such as register files. In many cases, the schedule of operations and data throughout the architecture is static requiring known cycle latencies within functional units and through the interconnect.

Figure 3.1 shows the relative position of CGRAs to competing platforms on the spectrum of efficiency and programmability. Starting on the far right of Figure 3.1 lies general purpose processors. They are very flexible, able to execute arbitrary programs, but the cost of this flexibility is low efficiency. GPGPUs
Figure 3.1: A spectrum of architectures showing a CGRA’s position in the spectrum of efficiency / programmability relative to competing architectures.

have shown to be very high performing. However, in terms of power efficiency, they have been shown to be generally less efficient than FPGAs. At the same time, GPGPUs are less flexible than general purpose processors; the set of applications able to be efficiently implemented by a GPGPU is smaller than a general purpose processor. Field-Programmable Gate Arrays (FPGAs) are even less flexible, though they can be more efficient than GPGPUs though the use of specialized circuitry. To the far left of Figure 3.1 lies custom ASICs. They generally have limited programmability due to their specialized nature but these specialized circuits perform very efficiently in terms of area, performance, and power. In between ASICs and FPGAs lies CGRAs. CGRAs aim to close the gap between FPGAs and Application Specific Intergrated Chips (ASICs) by reducing hardware flexibility with respect to FPGAs. This means less flexibility with higher efficiency circuits.

Throughout the related literature, the term ‘CGRA’ has been loosely used to describe a very wide gamut of hardware computing architectures. These architectures are generally data-centric rather than control, being composed of large configurable data-paths. We address some questions that arise regarding what constitutes a CGRA in the following discussion.

How ‘coarse’ are elements within the architecture? Fine-grained reconfigurability generally refers to FPGA-like structures that contain bit-wide interconnect and Boolean-function-configurable logic ele-
ments. On the other hand, coarse-grained elements refers to multi-bit interconnect and functional units that perform functions on buses of bits, such as Mathematical or Logical operations. The term ‘coarse’ has sometimes been extended to refer to arrays whose core functional units are larger and ‘self-sufficient’, such as a processor as in MPPAs [54, 55]. Here, self-sufficient implies that one of these large functional units could theoretically perform all the required operations in the application without the help of neighbouring functional units. These self-sufficient functional units have the capability of a general processor, having a local instruction memory and supporting local control mechanisms such as instruction branching. These styles of architecture, such as TRIPS [54] or Raw [55], are generally excluded from the term CGRA.

What level of ‘reconfigurability’ is required of the architecture? In CGRAs, the configurable data-path is the main construct that allows for CGRA performance. Hence, the level of ‘reconfigurability’ must be that the data-path itself must have multiple configurations. Furthermore, some CGRAs have dynamic-reconfigurability in which the array reconfigures itself as part of the application execution. Does the ‘A’ stand for Architecture or Array? And, does the architecture need-be array-like? Both Architecture and Array have been seen throughout the literature and are generally interchangeable. However, when Architecture is used, it can refer to non-array like reconfigurable data-path processors and sometimes to architectures that sit outside our definition of what is fundamentally a CGRA. Usually these are architectures that contain very coarse processing elements and are more closely related to MPPAs. We believe an array is not a CGRA if a processing element within the array is stand-alone and able to single-handedly process an entire application. In general, most CGRAs are array-like since this eases design complexity especially when there are many functional units. Some examples of these architectures will be overviewed in the following sections.

CGRAs have been studied since the early 90’s and have had moderate success. There is a wide gamut of coarse-grain architectures that have been proposed throughout the literature (some of which are further discussed in Section 3.2). Survey papers have catalogued many of the key works during this time [21, 22, 23]. Some early designs focused on one-dimensional arrays of functional units employing deep pipelining [3, 5, 56]. PipeRench [5, 56] was one of the first architectures to introduce virtualization and reuse of processing elements through dynamic-reconfiguration. This is a pre-cursor to the software pipelining techniques employed in more modern styles of architectures such as ADRES [7]. Earlier two-dimensional designs were also proposed but had limited interconnect capabilities [4, 6]. Recent years have also seen multiple commercial CGRA designs being produced. These are discussed in Section 3.2.7.
CGRA Benefits

Compared to traditional processors, CGRAs aim to increase silicon efficiency through the use of configurable hardware - similar to FPGAs. One significant source of inefficiency in traditional processor architectures is the high energy overhead of computation. In a case study, Hameed et al. [57] showed that roughly 90% of energy is overhead per instruction with 10% doing actual functional unit computation, for general-purpose chips in a multimedia application. 33% of energy was devoted to instruction fetch and decode, 19% data cache access, and 22% was used by pipeline registers, busses and clocking - areas that could be readily targeted by CGRAs. A plenary talk by Horowitz [58] further estimates the total instruction energy to be $10-100 \times$ the energy of the operation itself (depending on the operation e.g. add vs. mult).

To tackle these inefficiencies, the idea of spatial computing is employed in CGRAs, decentralizing and distributing data throughout the architecture, near the functional units where the data will be consumed. This distribution of data near functional units aims to optimize data movement, reducing energy.

Relative to FPGAs, the value proposition of CGRAs is improved speed, area-efficiency and power, owing to less overhead for programmability. Such benefits arise for applications whose computational requirements align closely with the available CGRA logic and interconnect. One can think of a CGRA as being midway between an FPGA and an ASIC, providing some of the programmability of an FPGA, while delivering superior power, performance, and cost characteristics for aligned applications. CGRAs are especially attractive when tailored to specific domains, where computational requirements are known, yet programmability is desired to accommodate an evolving technical specification.

CGRAs aim for higher performance than FPGAs for a narrower set of applications by limiting the flexibility of the architecture. The sea-of-gates fabric that makes FPGAs very flexible incurs large silicon overhead [1]. Modern FPGAs contain many hardened units throughout the FPGA fabric, such as BRAMs and DSPs, that positively impact silicon efficiency. The addition of these units into the traditional FPGA fabric has had a great impact on the performance of many applications. One perspective is that CGRAs take this idea of hardening one step further and the CGRA can be viewed as a collection of high performance DSP units with local storage connected together through a hardened low-area overhead interconnect. From a fine-grained perspective, comparing CGRAs to FPGAs, CGRAs: have less configuration overhead, have predictable critical path timing (fixed FMax), lower complexity CAD, efficient functional units and interconnect.

From a much more coarse-grained perspective, comparing CGRAs to MPPAs, CGRA functional units
or processing elements are not stand-alone, and cannot be individually programmed. Each functional
unit in the CGRA executes together in lock-step and does not allow branching. In multi-context CGRA
architectures, the functional units cycle through the same sequence of instructions. In single-context or
static CGRAs, the functional units have a set instruction that is always executed. MPPAs may have
larger local memories associated with a processing element, whereas a CGRA's individual functional
units may at most have an associated large register file for storage.

CGRA Challenges

While CGRAs have many benefits, they face challenges that may hinder their success. One major hurdle
for CGRAs is fair evaluation between CGRA designs, as well as CGRA alternatives and this is one of
the main driving factors behind the framework proposed in Chapter [4]. There are also different potential
areas for deployment be it low-power embedded or high-performance systems. Both areas would have
quite different approaches for the design of architectures. In yet another dimension, there are the
different application domains where CGRAs could be tailored – DSP, Machine Learning, Cryptography,
Multimedia, etc. In much of the literature, many different permutations of these parameters have been
explored, but with little comparison amongst CGRA designs or with alternatives.

With the many architectural decisions for each component, this large CGRA design space is also
tightly coupled with CAD and compiler techniques for mapping applications to the CGRA. The tight
coupling between benchmarks, CAD tools, and architectures poses difficulties for architecture evaluation.
In the GPC domain, there exists simulators that level the playing field. In the CGRA domain, there is
no standard simulator.

3.1.2 Components

In this section, an overview of the components and their variations within a CGRA is given, along with
some overview of the types of programming tools needed for different architectures.

Architecture Layouts

Arrays have had varying layout topologies for processing elements. Some designs have had one dimen-
sional arrays of processing elements, while most have had two dimensional orthogonal layouts. There
have also been interesting cases with hexagonal layouts [59]. Hierarchical or clustered layouts have also
been seen requiring different styles of interconnect [8].
Interconnect

Interconnect between elements has been varied from sparse connectivity to dense. Bi-directional and uni-directional connections, as well as bus based connectivity has been seen. Different interconnect topologies such as all-to-all, nearest neighbour and even island-style connectivity with channels and interconnection routers has been seen, similar to FPGAs. There may also be a mix of styles of interconnection structure, depending on the layout of the functional units (clustered, unclustered, and hierarchical).

Functional Units & Processing Elements

Processing Elements have been varied as well, some with simple functionality that may only perform simple arithmetic or logical operations, while others with more complex compound or composite functionality such as multiply-accumulate. The connectivity within the processing elements is varied as well, some with many registers or register files and others with none.

Reconfigurability & Dynamic Reconfigurability

Reconfigurability is one of the main features within a CGRA. However as expected, the extent that an architecture is reconfigurable is varied. This element of reconfigurability is mostly a function of the aforementioned three attributes.

Dynamic Reconfigurability is the ability of an architecture to change routing or the operations that functional units execute on a cycle-be-cycle basis \[7\]. A configuration refers to the desired routing and operations for the array. In modern designs, a dynamically reconfigurable CGRA has multiple contexts in which configurations are executed. A context contains a configuration of the array for a certain execution cycle within the overall schedule. These configurations then dynamically reconfigure the array for different contexts.

Control

Control and scheduling of functional unit execution is varied throughout the spectrum of CGRAs. The large majority of CGRAs have static scheduling, though others may have dynamic control. For CGRAs without dynamic reconfigurability (single context CGRAs), deep-pipelines can be generated that may have back-pressure pipeline control or FIFO buffers.

If the CGRA has dynamic reconfigurability, static scheduling is generally employed with configurations being swapped out between contexts in a cyclic fashion.
External Connectivity

Marshalling data on and off of the array can be challenging depending on the application and architecture. Some CGRAs are directly integrated with host-processors, utilizing the host-processor’s external connectivity. Others are stand-alone and could be bus connected through AXI, or have local scratchpad memories that are populated and depopulated by external mechanisms.

Programming Tools & CAD

Application specification has ranged from many custom languages to specifications in standard high-level languages such as C. Early works on CGRAs relied on hand mapping applications to architectures. Some had graphical user interfaces to accomplish this task [6]. Simulated annealing has been used extensively for placement of operations onto functional units [31]. Methods for routing data values over the interconnect have used Pathfinder-like approaches [60]. With the dawn of multi-context architectures, software pipelining [61] and iterative modulo scheduling [62] have been integrated into CAD tools along with annealers.

3.2 Notable Architectures

In this section, we describe notable designs starting from some of the earliest CGRAs up to more recent architectures, including some commercial designs.

3.2.1 RaPiD - Reconfigurable Pipelined Datapath

The RaPiD architecture [3] is a one-dimensional architecture that is designed to be reconfigurable for pipelining applications, such as DSP filters. Figure 3.2 shows one cell of the architecture with a 16-bit configurable datapath with configurable switches connecting a row of functional units, registers and local RAMs. This cell is replicated linearly to create the entire array. Control is mostly statically programmed similar to FPGAs with some dynamic configuration and with mappings created by hand.

3.2.2 KressArray

The Kress Array architecture [4] consists of a global bus-connected register file. There is a local unidirectional mesh interconnect between a two-dimensional grid of functional units. The architecture is depicted in Figure 3.3. An external host controls the array through an external bus. Local control marshals data in and out of the array also utilizing a local register file. Only feed-forward acyclic data-flows
can be mapped. Operator placement is performed using a simulated annealing-based algorithm \cite{31}. Routing is uncomplicated in the architecture, since the local interconnect is a simple direct connection to the nearest neighbour. List scheduling is performed with operator mobility determining the ranking of the operations - lowest first.

Follow-up works include KressArray Xplorer \cite{63} that provided an environment to perform design space exploration of KressArrays.

### 3.2.3 PipeRench

The PipeRench architecture was developed at Carnegie Mellon University \cite{5}. This architecture consists of an array of functional units arranged in horizontal ‘stripes.’ Each stripe acts as a pipeline stage with registered output and communicates unidirectionally to the downstream stripe. Figure 3.4 shows one stripe of the array with an interconnection network between each stripe. There is also limited direct (i.e. non-registered) communication between elements within the stripe. Due to the feed-forward nature of the architecture, only recurrences or cycles within the data-flow of length one can be implemented. Here, the interconnect within the stripe is used to implement this feed-back path. All paths in the interconnect between stripes are registered, making each stripe a pipeline stage. Their special ‘pass register file’ is also forwarded (passed) to downstream pipeline stages.

Applications are written in a custom language specifying the data-flow. Their compiler performs operator decomposition on multiply operations and the backend mapper was designed to be fast and is a deterministic, linear-time, greedy algorithm \cite{64}.
One interesting innovation in this architecture is their ability to virtualize stripes of processing elements. By doing so, larger and deeper pipelines can be implemented on a limited size array. This is akin to the notion of configuration ‘contexts’ that is now widely adopted in current CGRA designs.

### 3.2.4 MorphoSys

The MorphoSys architecture [6] is a hybrid architecture consisting of a TinyRISC processor [65] coupled with an 8\(\times\)8 array of coarse-grain reconfigurable cells. It was designed primarily for multimedia processing. Figure 3.5 shows the high-level architecture of the array with the controlling processor, as well as the array itself. Within the reconfigurable cell array, shown in Figure 3.6, each cell can be dynamically reconfigured while other cells execute. Rows and columns of cells share memories that store these configuration words. The array executes in SIMD fashion, with each row or column sharing the same configuration word that is broadcast across the row or column.

The TinyRISC processor is used to load configuration array memories and start execution of the array. The SUIF Compiler [66] is used to program the TinyRISC, while the array itself is manually
3.2.5 ADRES

The ADRES family of architectures [7] are a tightly coupled CGRA design that is rooted in VLIW architecture and software techniques. A prototypical ADRES architecture, depicted in Figure 3.7, is a hybrid CGRA/VLIW architecture. The CGRA comprises a two-dimensional array of processing elements that consist of functional units and register files connected with a bidirectional mesh interconnect. All routes and functional units have individual configuration memories and can change their behaviour on a cycle-by-cycle basis. There is also a large multi-ported shared register file that is coupled with a single row of functional units in the array. This creates the VLIW side of the architecture.

Software pipelining [61] is a typical technique utilized to efficiently use VLIW architectures and this technique is also used for programming the ADRES architecture, by first mapping to the VLIW
row of processing elements, then iteratively moving operations onto the CGRA through a simulated annealing process. This is the core technique in the accompanying DRESC compiler [67].

3.2.6 Mosaic

The Mosaic project at the University of Washington consists of a custom application language and front-end compiler, *Macah*, the *Electric VLSI* architecture generator and the *SPR* mapper [68]. The styles of architectures studied are ‘island-style’ CGRAs as shown in Figure 3.8. Similar to modern FPGAs, the architecture features island clusters of tightly coupled functional units that are surrounded by channels of interconnect, though the interconnect is now multi-bit routes. The precise capabilities of the application language and the architecture generator are unknown and published works provide few details on these components, though the types of benchmarks and architectures in the published works provides the reader with enough information to make some inference. The Electric VLSI architecture generator appears to be limited to being a template engine, as only architecture parameters are able to be input to the generator and not generic architecture descriptions.

For these island style CGRAs, the project has explored topics such as, the impact of time-multiplexing interconnect versus static interconnect [8], register file styles [69], and fused or compound operator functional units [70].

3.2.7 Commercial Architectures

There have been a number of commercial CGRA developments in recent years. Here, we overview three: one from Renesas Electronics, Samsung, and a new startup, Wave Computing.
Figure 3.7: The architecture of an ADRES CGRA featuring an $8 \times 8$ array of functional units with local register files. (Figure reproduced [7])

Figure 3.8: Mosaic ‘Island-style’ CGRA showing clusters of tightly coupled functional units that are surrounded by channels of interconnect, connected through switch-boxes. (Figure reproduced [8])
Chapter 3. Coarse-Grained Reconfigurable Architectures

Figure 3.9: Renesas DRP: Tile architecture DRP-4. (Figure reproduced [9])

Renesas DRP

The Dynamically Reconfigurable Processor (DRP) from Renesas Electronics [9] was designed and targeted for multimedia applications. This evolution, DRP-4, stemmed from prior iterations [71, 72]. The architecture features tiles that contain arrays of PEs employing multiple contexts with centralized control and many local memories. From Figure 3.9, we see the high-level architecture of a tile showing the array of processing elements surrounded by memories with the centralized state transition controller. For CAD, they produced a tool flow [9] that maps code in C to the architecture. Additionally, the architecture has other features such as dynamic frequency control.

Samsung: ULP-SRP

The Ultra Low Power Samsung Reconfigurable Processor (ULP-SRP) [10] was designed for low-power medical applications. The architecture is a variation of ADRES [7] with a small footprint design compared to the original ADRES architectures. The design contains nine functional units arranged in a $3 \times 3$ grid. Like ADRES, it has a VLIW mode in where only two functional units are active. It further has a low performance mode where four functional units are active and a high-performance mode where all nine are active. Communication between functional units is achieved through nearest neighbour interconnect including diagonal units. Figure 3.10 shows the high-level architecture of the SRP, illustrating the functional units that are active during different performance modes.
Wave Computing: DPU

Recently in 2017, Wave Computing developed a very large CGRA targeted towards deep neural network training and inferencing named the Wave Computing Dataflow Processing Unit \[11\]. Their design contains a huge number of functional units, 16,384, arranged in 1024 clusters. These 1024 clusters form a grid of $32 \times 32$ that is surrounded by AXI interfaces. The design is hostless and is ‘stand-alone’. The array is multi-context and is statically scheduled. The array is also able to be partitioned into smaller independent clusters of elements according to computing demands. Figure 3.11 shows the high-level architecture of the chip with 1024 clusters, partitioned into 24 groups.

Each cluster contains 16 processing elements. Figure 3.12 shows the architecture of a cluster. Elements are further grouped into quads that are fully connected allowing for fast shuffling of data. The DPU Arithmetic units implement operations such as multiply-accumulate, and act as co-processors to the main processing elements, executing in parallel. Another architecture highlight is the use of local clocks within clusters to improve clock rates.

LLVM is used as a front-end compiler and the LLVM intermediate representation is used as input to the backend CAD flow. Their CAD tools partition the data-flow graphs derived from the LLVM IR over the clusters in the CGRA. Individual cluster mapping is performed through Boolean Satisfiability. They also provide a framework to stitch together pre-compiled libraries of kernels.
Figure 3.11: Wave Computing DPU: High-level architecture showing 1024 clusters, peripheral AXI ports and an example partitioning scheme of 24 groups. (Figure reproduced [11])

Figure 3.12: Wave Computing DPU: Cluster architecture containing four quads of processing elements and eight special arithmetic units. (Figure reproduced [11])
3.3 Summary

In this chapter, we have outlined what constitutes a CGRA and where it sits within reconfigurable computing and computing platforms in general. We have highlighted key architectural features within CGRAs and overviewed notable architectures throughout the history of CGRAs.
Chapter 4

The CGRA-ME Framework

In this chapter, we discuss the design and creation of an open-source framework, CGRA-ME (CGRA Modelling and Exploration), that is freely available to the research community. The CGRA-ME framework is a collaborative project at the University of Toronto. We believe the framework will open up a wide variety of research opportunities for Coarse-Grained Reconfigurable Array (CGRA) architecture, algorithms and applications by providing a foundation on which further research can be performed. Background and motivation is provided for the creation of CGRA-ME, followed by a description of the framework itself. Two experimental studies are shown, showcasing the capabilities of the framework. The first, published in the proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP) [29], exercises the simulated annealing based mapper over basic CGRA architectures with physical design of the architectures evaluated using a TSMC 65 nm standard-cell library. The second, published in the proceedings of the ACM/SIGDA International Symposium on Physical Design (ISPD) [30], exercises the integer linear programming based mapper modelling ADRES-like architectures [7] with physical design of the architectures evaluated using a FreePDK 45 nm standard-cell library [73] and as an overlay using a Stratix IV Field Programmable Gate Array (FPGA).

4.1 Background & Motivation

CGRAs have been studied in academia and a variety of different architectures have been proposed [21, 22, 23], while commercially, recent years have also seen CGRAs in the market, as discussed in Chapter 3. These prior works on CGRAs have been largely constrained within their respective frameworks, wherein a research group or company proposes a single style or template of CGRA design and demonstrates its utility for a set of applications. However, the design space for CGRAs is very large with many
architectural decisions for each component, as well as overall functionality, layout, and architecture, as seen in the varied designs in Chapter 3. This large design space is also tightly coupled with CAD and compiler techniques for mapping applications to the CGRA, where there has also been considerable prior work, some of which is discussed in Chapter 5.

A CGRA can be physically realized in a number of ways: 1) as a custom chip, with manual layout, 2) as a standard-cell ASIC, or 3) on an FPGA as an overlay. We believe the first option to be prohibitively expensive, and prior works focus on the second [5, 7, 74, 75] and third [76, 77] options. We also focus on these options in the initial studies using CGRA-ME. A standard-cell CGRA will have comparatively less silicon area dedicated to programmability than an FPGA, therefore offering better power/performance than an FPGA for applications whose computational and communication needs are aligned with the CGRA architecture’s capabilities. For physical design, there are a number of related works but they were not produced within a larger unifying framework, which makes it difficult to compare different architectures in a holistic way.

Open-source modeling and evaluation frameworks have long existed for traditional processors (e.g. gem5 [78]), GPUs [79], and fine-grain FPGAs [24] but there is no analogous framework for CGRAs. Such an analogous framework would permit the scientific exploration of CGRAs where, for example, architecture or CAD parameters can be independently isolated and evaluated. There is a need, then, for a tool that permits an architect to model hypothetical CGRA architectures and implement CAD algorithms, to evaluate the area, speed, and power of such designs over a set of applications in a domain of interest.

There have been different CGRA designs and mapping flows proposed, the summary of which can be found in excellent survey articles [80, 22, 23, 81, 82]. Several recent works are related to sub-pieces of the CGRA-ME framework. For example, CGADL [83] provides an architecture description language that offers features similar to CGRA-ME’s XML-based language and architecture modelling (discussed in Section 4.3), but is disconnected from mapping of benchmarks and is not able to output RTL. Other works [84, 85] describe simulators for CGRAs, however, these are disconnected from physical CGRA implementation and modelling. There are works focused on mappers such as DRESC [67], SPR [68] and others [86, 87, 88] but are not integrated into a larger generic architecture modelling framework. For example, some of the capabilities of the simulated annealing based mapper within CGRA-ME bears similarity to the SPR tool from the University of Washington [68], which provides a generic mapper for CGRAs, able to adapt to a user-provided architecture specification. This is also similar to the DRESC mapper [67].

While these prior works bear some overlap with portions of CGRA-ME’s capabilities, CGRA-ME’s
Chapter 4. The CGRA-ME Framework

over-arching contribution is a free open-source unified framework. CGRA-ME aims to amalgamate many aspects of these prior works within a larger, and more complete CGRA architecture evaluation framework that encompasses architecture description, architecture modelling, application mapping, system simulation, and physical implementation. Producing such an all-encompassing framework is a lofty goal due to its large scope. However, having a standard unified framework is a requirement for sound scientific evaluation of CGRAs and CGRA-related CAD tools.

4.2 Framework Introduction and Overview

CGRA-ME is an open-source software framework written in C++ that provides architecture modelling, application mapping from C to bitstream, and device RTL for modelled CGRAs. The framework is designed modularly, targeting each of these aspects. The framework provides a flexible scheduling, mapping, placement and routing tool, leveraging the popular LLVM compiler framework [89], that maps an optimized C-language benchmark onto the specified CGRA. Automatically generated Verilog RTL for the CGRA device permits simulation with ModelSim for functional verification, and also allows physical realization of the CGRA to assess area, performance and power of hypothetical architectures. With CGRA-ME, a wide variety of CGRA architectures can be modelled by a human architect using an XML-based language or a C++ API. The CGRA-ME framework accepts the architecture description as input, as well as an application benchmark to be mapped onto the CGRA. Through this process, new architectures can be developed and explored by using the framework to map different application benchmarks, individually, to the new architectures.

Verilog RTL for the CGRA is produced automatically from the device model, along with a bitstream configuring the device for the given benchmark. The generated Verilog can be synthesized to a standard-cell ASIC or an FPGA-overlay CGRA implementation to assess performance of the architecture. The RTL and configuration bitstream can also be simulated for functional correctness.

Here, we will discuss each of these aspects by starting first with a full overview of the framework. Figure 4.1 depicts the CGRA-ME framework, its internal components, and data-flow. Circled labels are shown on each component. The inputs to the framework are a C-language application benchmark ①, as well as a textual description of the CGRA architecture ②. The textual description is given in an XML-based CGRA-ME architecture description language, called CGRA-ADL and is elaborated upon in Section 4.3. As an alternate route to specifying an architecture through CGRA-ADL, CGRA-ME also allows an architect to specify and model a CGRA architecture through a C++ API. The CGRA-ADL is parsed by an architecture interpreter ③ that interacts with the C++ modelling API (not shown),
producing an in-memory device model of the CGRA \(4\). This device model consists of a hardware module hierarchy – objects representing each hardware component.

A benchmark is parsed and then optimized by the LLVM compiler \(89\). One or more data-flow graphs are extracted from the input benchmark using LLVM \(5\) and input to the CGRA Mapper \(6\) as a Data-Flow Graph (DFG). The specific portions of the program for which data-flow graphs are extracted are based on user annotations to the source code. This is elaborated upon in Section \(4.4\).

The CGRA mapper \(6\) receives the application data-flow graph as input, as well as uses the in-memory device model to map the application onto the CGRA. Within the framework, two mappers are implemented and are described in detail in Chapter \(5\).

The CGRA architecture interpreter \(3\) is also capable of generating Verilog RTL for the specified CGRA \(7\). The generated Verilog has several utilities. First, the Verilog, along with a configuration bitstream produced by the mapper, can be simulated to verify CGRA functionality \(8\). Second, the Verilog can be input into a standard-cell ASIC design flow \(9\), to realize a physical implementation, which can then inform performance/power/area models \(B\). Or third, the RTL can be input into an FPGA design flow \(A\) to realize an FPGA-overlay CGRA implementation, and associated performance/power/area models \(C\). Finally, physical performance/power/area models can be combined with the mapping
results to analyze specific input benchmarks implemented on the modeled CGRA.

In the following sections, we describe each component of the framework in detail starting with the modelling of architectures.

4.3 Architecture Model and Architecture Description Language

The CGRA-ME framework is able to model a wide range of architectures, as can be specified by a human architect in a high-level XML-based architectural language, *CGRA-ADL*, or through a C++ API. The language permits an architect to describe a CGRA in detail, permitting an in-memory device model of the complete CGRA to be constructed. The device model contains primitive components, such as routing multiplexers, registers, register files, and functional units, and the connectivity between them, as well as configuration cells to program the array. An example of a module with primitive components is shown in Figure 4.3. Each primitive within this module has its own parameters such as bit-width, or number of inputs. Custom components can also be implemented within architectures through the language, as long as the user-defined custom block has its own predefined device model.

4.3.1 Architecture Specification C++ API

Within CGRA-ME, a CGRA is modelled as a heirarchy of hardware component objects, *modules*, that store functionality and connectivity. Each hardware module inherits from the *Module* class. Hardware modules can either be primitive components with prescribed functionality or composite modules that contain a collection of modules along with their connectivity. The top-level architecture module CGRA itself is a *Module*.

Through direct calls to the C++ API, a CGRA architect can construct an architecture in entirety, specifying an architecture hierarchically in terms of submodules and ports, and their connectivity to one another. For example, the user calls API functions within the *Module* class such as *addPort*, *addSubModule*, *addConfig*, and *addConnection* to construct a composite module. An example of such a module is a configurable I/O port for the CGRA. To construct an I/O port, the user would write code in C++ shown in Figure 4.2 using calls to the API functions. As another example, the code to construct the block in Figure 4.3 is shown in Figure 4.4. The function names in the API are fairly intuitive and full documentation can be found online. *Register* and *TriState* and *ConfigCell* are primitive hardware module components whose implementations are provided by the framework.

These function calls add the necessary ports, submodules, configuration cells, and connections. Through such calls, a software architecture model of a candidate CGRA architecture is generated. From
Chapter 4. The CGRA-ME Framework

4.3.2 The CGRA-ADL Language

CGRA-ADL allows the textual specification of a CGRA architecture in a concise XML-based language. This XML-based language is inspired by the XML-based language used in the VTR FPGA architecture evaluation framework [25]. CGRA descriptions specified in the XML-based language above are parsed to generate the in-memory CGRA device model through C++ API calls. The complete language definition is available on the CGRA-ME website [29]; here, we overview a few aspects of the language at a high
```cpp
int num_inputs = 4;

// add input muxes
addSubModule(new Multiplexer("mux_a", num_inputs));
addSubModule(new Multiplexer("mux_b", num_inputs));

// add FU
addSubModule(new FuncUnit("func",
    {
        OGRAPH_OP_ADD,
        OGRAPH_OP_MUL,
        OGRAPH_OP_SUB,
        ...
    },
    {});
// add reg
addSubModule(new Register("register"));

// add output mux
addSubModule(new Multiplexer("mux_out", 2));

// config cells
addConfig(new ConfigCell("MuxAConfig"), {"mux_a.select"});
addConfig(new ConfigCell("MuxBConfig"), {"mux_b.select"});
addConfig(new ConfigCell("MuxOutConfig"), {"mux_out.select"});
addConfig(new ConfigCell("FuncConfig"), {"func.select"});

// input ports
for (int i = 0; i < num_inputs; i++)
    addPort("in" + std::to_string(i), PORT_INPUT);

// output port
addPort("out", PORT_OUTPUT);

// connections to mux_a
for (int i = 0; i < num_inputs; i++)
    addConnection("this.in" + std::to_string(i), "mux_a.in" + std::to_string(i));

// connections to mux_b
for (int i = 0; i < num_inputs; i++)
    addConnection("this.in" + std::to_string(i), "mux_b.in" + std::to_string(i));

// connections to funcunit
addConnection("mux_a.out", "func.in_a");
addConnection("mux_b.out", "func.in_b");

// to reg
addConnection("func.out", "register.in");

// to output mux
addConnection("register.out", "mux_out.in0");
addConnection("mux_b.out", "mux_out.in1");

// to output
addConnection("mux_out.out", "this.out");
```

Figure 4.4: C++ API calls to generate the block in Figure 4.3 including configuration cells.
level through examples.

**Example Architecture Description**

Figure 4.5 shows a sample of the overall structure of an architecture description. The `<cgra>` tag on Line 1 opens the architecture description. Following this, the architect may make one or more definitions (Line 2), which are similar to `#define` macros in C. Next, the architect describes the one or more CGRA blocks that will comprise the architecture, including intra-block logic and routing (Lines 3–7). The later Lines 8–12 compose the overall array architecture by instantiating a pattern of the defined blocks and specifying the inter-block connectivity.

```xml
1 : <cgra>
2 : <definition .../>
3 : <module name="block1">
4 :   <input name="in"/>
5 :   <inst module="fu"/>
6 :   <connection .../>
7 : </module>
8 : <architecture row="4" col="4">
9 :   <pattern>
10:   <block module="block1"/>
11: </pattern>
12: </architecture>
13: </cgra>
```

Figure 4.5: CGRA-ADL architecture description example showing the declaration of a module called “block1” with instances of this block being used to populate a 4x4 architecture.

**Example Hierarchial [CGRA] module**

An example CGRA block along with its CGRA-ADL description is shown in Figure 4.6. Lines 2–4 declare the module ports and their directions. Line 5 declares a wire. Line 6 instantiates a functional unit inside the CGRA block. `<FuncUnit>` is a primitive type in the language for a two-input single-output functional unit (elaborated on below). Lines 7–10 form the connections between the ports, wire and the functional unit.

**Example Interconnection Patterns**

Figure 4.7 shows interconnection structures available for modelling intra-CGRA block connectivity. Four structures may be used, shown from top to the bottom in the figure, referred to as direct, distributive,
1: <module name="module1"/>
2:  <input name="in1"/>
3:  <input name="in2"/>
4:  <output name="out"/>
5:  <wire name="w"/>
6:  <inst name="fu" module="FuncUnit"/>
7:  <connection from="this.in1" to="w"/>
8:  <connection from="w" to="fu.in1"/>
9:  <connection from="this.in2" to="fu.in2"/>
10: <connection from="fu.out" to="this.out"/>
11: </module>

Figure 4.6: Using CGRA-ADL to describe a module (named module1) that contains a single functional unit and also to describe module1’s internal connections to the functional unit’s input and output ports. An internal wire w is also modelled.

*multiplexer* and *crossbar*. The multiplexer and crossbar interconnection on Lines 3 and 4, respectively, imply the presence of SRAM configuration cells on the select inputs of the multiplexer(s).

**Example Top-level Patterns**

The top-level architecture of CGRA blocks is represented as an array. Each tile or block is a grid position. These tiles or blocks can be any type of module (register files, functional units, composite modules, etc.). The architect can use the `<pattern>` tag to populate a selected sub-region of the array with a repeating pattern of previously-defined CGRA blocks. Figure 4.8 shows such an example.

**Example Top-level Connectivity**

The language also provides low-level control of inter-block interconnect as illustrated in Figure 4.9 permitting an architect to specify virtually *any* desired connectivity. In this case, relative location constraints (`rel <deltaRow> <deltaCol>`) are used to specify a connection between two adjacent tiles.
Figure 4.7: CGRA-ADL interconnect structures. Line 1 models direct connections; line 2 models distributive connections; line 3 models multiplexers, and line 4 models crossbars.

This permits architects to explore any interconnection pattern of their choosing, whether it be for a 2-dimensional or 1-dimensional array, or an irregular pattern. Additionally, we also provide a “syntactic sugar” shortcut for two interconnect patterns we believe to be the most widely used in CGRA architecture research, depicted in Figure 4.10. The first (Figure 4.10(a)) is an architecture with North, South, East, West connectivity; the second (Figure 4.10(b)) adds the diagonal connections.

### 4.4 Mapper Front-end: LLVM

The LLVM framework was chosen to implement the front-end compiler support for the mapper within the CGRA-ME framework. The LLVM framework is popular and well supported compiler framework that has widespread use in industry and academia. The LLVM-based front-end is integrated with the framework supporting the extraction of DFGs, which are the input to the backend mapper that performs
Figure 4.8: CGRA block patterns using CGRA-ADL. This code snippet populates a 4-by-4 region of the CGRA with a repeating 2-by-2 pattern of CGRA blocks. \texttt{col=2} signifies the pattern of four modules is two columns wide and must wrap around to a second row. The \texttt{row-range} and \texttt{col-range} parameters specify the range of block coordinates where the pattern should be applied.

mapping onto modelled CGRA architectures.

At the input to the application mapping flow, benchmarks written in C are parsed, optimized and translated into DFGs using the CLANG compiler front-end and a custom LLVM pass. The LLVM framework translates the C code into LLVM intermediate representation (IR). LLVM’s IR can be viewed as a machine-independent assembly code representation of an input program. The IR is based on a reduced/simple instruction set of basic logical and arithmetic operations, such as multiply, add, subtract, exclusive-OR, etc.

The custom analysis pass examines C benchmarks, extracting kernels from inner-loops. The current application flow focuses on the implementation of computationally intensive loop kernels. However, branching within the kernel is currently not supported. Through the loop-analysis pass built within LLVM, operations are translated into a DFG. Each operation in the DFG is created on an instruction-by-instruction basis within the loop body.

These DFGs are written out into a dot graph format \cite{dot} that includes metadata, such as labeling inputs, outputs, operations, and operands within the operations. An example of this is shown in Figure 4.11. A DFG is a directed graph $G(V,E)$, where vertices represent operations and edges are data values passed from one operation to the next (dependencies between operations). In many CGRA...
Figure 4.9: Example interconnection showing the use of relative position macros for instance names in the red circled portion of the figure. An instance of an FUb and an instance of an I/O module are created. Then two connections are made, one from the IO 'out' port to the FUb 'in1' port, and one from the FUb 'out' port to the 'in2' port of FUa.

Figure 4.10: CGRA interconnection architectures for which automatic connection patterns are provided in CGRA-ADL.
tool-flows, a DFG is used to represent kernel computations. The DFG accounts for all computation operations, memory accesses and I/O, as well as the data dependencies and data transfer between these operations. This graph representation is also able to capture loop-carried dependencies through back-edges within the DFG structure. Edges within the graph represent the flow of data from outputs to inputs of operations. Each operation within the DFG corresponds to one LLVM IR instruction and the computational capabilities of a CGRA functional unit are specified in terms of the types of LLVM instructions they are capable of supporting. Each functional unit on the CGRA is able to process a single LLVM instruction at a time and each functional unit may be capable of supporting one or more LLVM instructions. In this way, the compiler’s (i.e. LLVM’s) representation of the computations and computational capabilities of the modelled CGRA architecture are aligned with each other.

In addition, the CGRA-ME framework also contains a custom parser that is used to read generic DFGs in dot graph format. Using the dot graph format, handcrafted benchmarks can be made or custom user front-ends can be built adhering to the standard.

4.5 Mapper

The process of implementing a benchmark onto a CGRA is called mapping. This process involves, at the highest level, a translation of a benchmark from a high-level software language such as C to a DFG shown in Section 4.4. This DFG is then implemented on the CGRA through association of operations within the application’s DFG to functional units on the architecture. After associating an operation within the DFG to a functional unit, the mapping process needs to ensure a route between inputs and

![Diagram of a DFG](image)
outputs of each operation, while also accounting for registers and obeying a correct operation schedule.

To model the physical routes and functional units within the CGRA architecture, an MRRG [67] is constructed from the Device Model (Figure 4.1 component 4) of the CGRA. The MRRG is created by querying each module instance within the device model’s hierarchy, producing a final graph from sub-graphs of the MRRG building the MRRG in a piece-meal fashion. The MRRG is commonly used to model the CGRA resources onto which the application DFG is mapped. This graph is described in more detail in Chapter 5.

At a high-level, the graph is an abstract representation of the physical architecture (focused on the data-path) of a CGRA. The graph contains, as vertices, all resources of the CGRA: the routes within the physical architecture, the storage elements, and the functional units that execute operations. The graph also formulates the constraints of the modulo scheduling problem, operator placement, and value routing, all within the graph itself. Many recent architectures and tools rely on some form of this representation [91, 92, 68, 87, 86]. The MRRG representation is especially useful for multi-context architectures where routes and functional units can be shared to perform different operations on subsequent cycles in a repeating pattern. To model such multi-context architectures, the MRRG is composed of copies of the physical resources – one for each cycle of dynamic context. A register in the underlying CGRA architecture, which stores a value across clock cycles, is translated into an edge between nodes in two subsequent copies of the physical resources. The mapping of DFG operations and edges to the MRRG reflects the computations each functional unit is performing in each dynamic context, and also the routing connectivity. The mapping problem is then reduced to associating a DFG with an MRRG. This process of associating each operation in the DFG with a functional unit in the MRRG while establishing a route between inputs and outputs of each operation, is non-trivial with some specific formulations for mapping shown to be NP-complete [86]. Figure 4.12 shows an example of the mapping process where functional units in the MRRG are associated with operations on the DFG.

In this dissertation, we use the term Mapper to refer to Figure 4.1 component 6. There are two mappers implemented within CGRA-ME. The first mapper is a simulated annealing-based mapper as traditionally mappers have employed simulated annealing-based techniques [67, 68]. In this mapper, simulated annealing [31] is used to direct the placement of operations onto physical units and timeslots and a PathFinder-like [60] approach used for routing connections between functional units. Since we map the DFG onto the MRRG that already accounts for resource usage on a cycle-by-cycle basis, the scheduling of operations is implicit in the placement and routing. The second mapper implemented within the framework is constraint based. The graph association problem is formulated as an integer linear programming (ILP) problem. While our ILP mapper is not the first formulation to be applied to
Figure 4.12: A two context MRRG snippet and a DFG showing how the mapping process associates operations on the DFG with functional units (orange) on the MRRG and routes values on routing resources (blue).

CGRAs [93, 94, 95], the main difference with CGRA-ME’s ILP formulation [32] is that the formulation is valid over any architecture for which an MRRG can be generated, while other works are constrained to specific architectures or architectural templates. Details of both mappers are given in Chapter 5.

4.5.1 Bitstream Generation

A successful mapping associates all Operations and Values with one or more MRRG nodes. The MRRG is iterated over and a configuration map is created from each module instance to a list of associated Operations, Values and MRRG nodes. The configuration hardware modules are then walked in scan chain order. For each configuration module, the configuration connected module instance is queried to generate the configuration bits required. This is achieved through inspection of the mapped Operation and Values on the associated MRRG nodes. For example, by observing the inputs and output of a routing multiplexer, the correct bits within the configuration are set. The generated configuration data for each module is then stored in the configuration map. For simulation, a Verilog testbench is automatically produced containing the configuration bitstream, and also simulates programming an instance of a top-level CGRA design.
Chapter 4. The CGRA-ME Framework

4.6 RTL Generation

Using the Device Model (Figure 4.1 component 4) of the CGRA, the framework is able to generate Verilog RTL for modelled CGRA architectures. The generated Verilog RTL can be simulated for functional verification and it also can be synthesized so that physical implementations of the CGRA can be produced and evaluated. An Application Specific Integrated Chip (ASIC) implementation can be produced by using a standard-cell toolchain or alternately, an FPGA overlay can be realized using commercial FPGA vendor tools. This framework-produced RTL has been validated through the study shown in Section 4.7 and Section 4.8.

The Verilog RTL is produced bottom up within CGRA-ME, starting with all the base hardware component primitives (multiplexers, functional units, etc.). All base hardware component primitives have parametrized Verilog generation routines. As an example of parameters for primitives, a multiplexer primitive takes the number of inputs and the data bit-width as hardware parameters for RTL production. Another example of parameters is in ALU generation, where the architect is able to specify a list of operations that ALU is able to perform. The in-memory objects representing modules are organized in a hierarchical manner. Each of these objects represent a logical hardware unit, either a primitive or a composite module, that encapsulates its connections, ports, sub-modules, and configuration sub-modules. This hardware tree hierarchy is traversed recursively to generate the full architecture. Starting from the leaves, which are all hardware primitives, the Verilog generation routine is run for each unique module type, and a parameterized Verilog file is emitted with ports. The datapath connections are then wired between primitives' ports at that level of the hierarchy. During this phase, the shift register connections between all configuration cells is generated automatically. Sub-modules containing configuration registers are also hooked up to the module they configure, and automatically linked together in the chain. This connects each primitive’s configuration cells together in a known order forming an inter-module scan chain for future programming. All clock and reset connections are also generated. This is subsequently repeated for each level of the hierarchy until the top-level CGRA hardware module is generated.

4.7 Experimental Study #1

This first study demonstrates the viability of the CGRA-ME framework. The study accomplishes this through: 1) demonstrating CGRA-ME’s capability to model a variety of realistic CGRA architectures using the CGRA-ADL; 2) demonstrating the ability of the mapper to map applications onto a CGRA using the simulated annealing based mapper; that is, we aim to show the mapping algorithms are capable
of recognising and adapting to the modelled CGRA architecture, including its size, interconnect patterns, functional unit capabilities, and number of execution contexts; 3) demonstrating the framework’s ability to produce Verilog RTL for a modelled CGRA, as evidenced by a standard-cell implementation of the CGRA.

A variety of CGRA architectures are considered in this study, chosen to be representative of proposed in the literature and different enough from one another to exhibit varying levels of mapping difficulty. Each architecture comprises an array of CGRA blocks with 32-bit-wide bus-based interconnect. The sample CGRA block we use is the same as in Figure 4.3, with a few differences to the input multiplexers and functional unit capabilities. We consider two function block architectures and two interconnect architectures with varying array sizes. The architecture sizes range from $4 \times 4$ to $6 \times 6$ functional blocks. Additionally, we model these architectures with 1, 2 and 4 execution contexts. In the single execution context case, the CGRA configuration is held constant across clock cycles; whereas, with 2 and 4 contexts, functional unit and routing configuration can change on each cycle.

The two function block architectures are called Homogeneous and Heterogeneous. In the former, all blocks can perform ALU-like operations: shift, logical, add, subtract, multiply. In the latter, only half of the blocks can perform all operations; the other half can do all but multiply, saving area. In all designs, the output of every functional unit is registered. The two interconnect styles are called Orthogonal and Diagonal, reflecting the two routing architectures shown in Figure 4.10, respectively. In the diagonal case, the routing multiplexers shown in Figure 4.3 are widened to accommodate the additional diagonal connections. In all architectures, I/O blocks are placed on the perimeter. A single memory-access unit that can perform loads and stores is dedicated to each row within all architecture variants. The outputs of each block within a row are connected to the inputs of the access unit (address and data-in) while the output of the memory block (data-out) is connected to each of the blocks in the row as an additional input.

4.7.1 Mapping & Architecture Evaluation

The benchmarks are data-flow graphs having varying numbers of operation nodes and edges to reflect varying difficulties of mapping for the target architectures. Table 4.1 shows detailed characteristics of each benchmark.

Table 4.2 shows the mapping results for four different architectures, indicating whether the mapper was successful in finding an implementation for each benchmark in the four architectures. The table is divided into two sections; one for single context architectures and one for dual context. Within these
Table 4.1: Benchmarks statistics. I/Os also includes memory operations, load and store. The add and multiply benchmarks are trees of operations with 10 to 16 inputs. 2x2-p and 2x2-f are synthetic benchmarks. mac is a multiply-accumulate and weighted sum is as it’s named. The other benchmarks are Taylor series approximations of common transcendental functions.

divisions, there are the four architecture styles and within those are three different CGRA sizes. Each cell indicates whether a benchmark was mapped (M) or unmapped (U). The architecture styles are arranged from least flexible (heterogeneous functional units with orthogonal routing) to most flexible (homogeneous functional units with diagonal routing). Broadly speaking, looking at the success-count in each column (last row), the results match with intuition: the largest array with the most flexible architecture can accommodate most of the benchmarks, whereas the smallest array with least flexible architecture can only accommodate a few due to limited compute ability and interconnect. Note that benchmarks using more than 8 multipliers (cf. Table 4.1) are impossible to map within the single-context $4 \times 4$ heterogeneous architectures (as it has only 8 functional units that can perform the multiply operation).

The right side of Table 4.2 shows the mapping results for dual context architectures, where there are two configurations of the functional units and interconnect that alternate on a cycle-by-cycle basis. This effectively doubles the capacity of the array, though now, the throughput of the array is halved. An underlined cell in this half of the table indicates that the same benchmark was not able to be mapped with one context, but is now able to be mapped with two contexts. For these architectures, mapping success is much higher owing to the huge amount of flexibility granted by an additional execution context.

Mapping was also attempted for four-context architectures (not shown) and all benchmarks that did not exceed our 12 hour time-limit were able to be mapped – 178 out of 204 mapped successfully. This
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Single Context</th>
<th>Dual Context</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 5 6 4 5 6 4 5 6 4 5 6 4 5 6</td>
<td>4 5 6 4 5 6 4 5 6 4 5 6 4 5 6</td>
</tr>
<tr>
<td>add 10</td>
<td>M M M M M M M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>add 14</td>
<td>U U M M M M U U M M M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>add 16</td>
<td>U U M U M U M U M U M M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>2x2-f</td>
<td>M M M M M M M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>2x2-p</td>
<td>M M M M M M M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>cos 4</td>
<td>U U U U U U U U M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>cosh 4</td>
<td>U U U U U U M U U U U U M M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>exponential 4</td>
<td>U U U U M M M M U U U U M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>exponential 5</td>
<td>U U U U U U M U U U U U M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>exponential 6</td>
<td>U U U U U U U U U U U U</td>
<td>U U M M M M M U M M M</td>
</tr>
<tr>
<td>multiply 10</td>
<td>U U U U M M M M U U U U U U</td>
<td>U U M M M M M U M M M</td>
</tr>
<tr>
<td>multiply 14</td>
<td>U U U U U U U U M U M M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>multiply 16</td>
<td>U U U U U U U U U U M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>sinh 4</td>
<td>U U U U U U U U U U U U</td>
<td>U U M M M M M U M M M</td>
</tr>
<tr>
<td>taylor series 4</td>
<td>U U U U M M M M U U U U M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>weighted sum</td>
<td>U U U U U U M M M M U U U U U U</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>mac</td>
<td>U U U U U U M M M M U U U U U U</td>
<td>M M M M M M M M</td>
</tr>
</tbody>
</table>

# mapped | 3 3 7 7 10 14 4 5 10 9 14 15 14 15 17 17 17 17 17 17 17 17

Table 4.2: Benchmark mapping results for single and dual context architectures, four array styles and three array sizes. A Mapped benchmark is denoted by M and an Unmapped benchmark denoted by U. An M denotes a benchmark that was able to be mapped in two contexts but not in one context.
result is slightly worse than two context mappings as there are only 7 failures in the two context case. Though, we hypothesise that given additional computation time, all benchmarks would be mapped for the four-context architectures.

Moving from a two-context to a four-context architecture, the \textbf{MRRG} doubles in size, significantly increasing the complexity of the mapping problem. Compounded with the very slow and conservative annealing schedule that was used, the results for the four context architectures are not surprising due to the nature of a simulated annealing algorithm.

We underscore that our intent here is not to draw strict architectural conclusions; rather, to demonstrate that CGRA-ME is capable of modelling and mapping to a variety of CGRA architectures having different degrees of functional and interconnection restrictiveness.

\subsection*{4.7.2 TSMC Standard-Cell Implementation}

Standard-cell implementations for four 4×4 single context architectures were synthesized and evaluated using the flow described in Section 4.6 targeting the TSMC 65 nm cell library. Our standard-cell synthesis flow uses Synopsys Design Compiler for technology mapping to the cell library, Cadence Encounter for placement and routing, and Synopsys PrimeTime for timing analysis. Scripts automatically generate floorplanning constraints after technology mapping, assuming a square die and using the block area estimates produced by Design Compiler. The floorplanning constraints encourage the 2D layout of blocks to match with the logical layout assumptions (e.g. the block at row 1, column 1 is physically adjacent to the block at row 1, column 2). It is worth mentioning that CGRAs with different area, performance, power trade-offs can be produced by simply changing the constraints supplied to the ASIC tools. A standard-cell implementation can be produced in a push-button manner, with a single \texttt{make} target. In this study we direct the ASIC tools to produce area-optimized implementations without external memory or the memory port connections.

Figure 4.13 shows the floorplan and full metal layouts for two of the four 4×4 CGRAs produced. The left two blocks show the floorplan and layout of the CGRA with homogeneous blocks and orthogonal connectivity between blocks; the right two blocks show the floorplan and layout of the heterogeneous CGRA with diagonal and orthogonal connectivity.

The area and timing analysis results for all four architectures evaluated are shown in Table 4.3. The area results reflect the intuition that the homogeneous fabric with richer connectivity consumes the most area; the heterogeneous fabric with reduced connectivity consumes the least area. Observe that the critical path delay in all architectures is roughly the same, and reflects the delay through the multiplier.
(\sim 200\text{MHz} \text{ for all architectures}). We believe the critical delay differences among the architectures may reflect algorithmic noise in the ASIC tools. The results in the table demonstrate that, from the architecture description in XML, CGRA-ME is able to generate a standard-cell implementation, that, combined with mapping results, permits the area/performance of hypothetical CGRA architectures to be evaluated for a set of benchmark applications.

### 4.8 Experimental Study #2

In this second study, we use CGRA-ME to model a more complex architecture, demonstrating the constraint-based ILP mapper (that is further discussed in Chapter 5) to perform application mapping. All benchmarks are coded in C and taken through the LLVM front-end DFG extraction flow, whereas the previous study only had two benchmarks that utilized this tool-flow. Compared to the previous study, all benchmarks are mapped using the newer integer linear programming based mapper. In addition to producing an ASIC design, an FPGA overlay is also created utilizing the same RTL. The configuration generation and programming has been improved and verified through simulation.

#### 4.8.1 Experimental Architectures

We model two architecture variants resembling the ADRES architecture [7]. Figure 4.14 shows a high-level view of both architecture variants that we model. The two targeted variants of ADRES are the...
“Full” and “Reduced” architecture and both are 4×4 grids of functional units. The Full architecture consists of functional units with a full set of operations: add, subtract, multiply, shifts, and, or, and xor, while also having torus connectivity between top and bottom rows, and between leftmost and rightmost columns; the Reduced architecture contains processing elements with only add and subtract capabilities in odd columns, and there are no torus connections. Both variants have a shared register file, I/O, and memory ports connected to the top row of functional units.

### 4.8.2 Benchmarks & Mapping Results

The ILP mapper within CGRA-ME (Chapter 5) was used to map a combined set of synthetic and real benchmarks to the two architecture variants. Table 4.4 shows the benchmark set, along with statistics and mapping results. All benchmarks were able to be mapped into the Full architecture that contains 16 multipliers (one in each functional unit) and toroid connections. In the Reduced architecture, there are only ten multipliers and no toroid connections. Not all benchmarks map to this architecture, especially ones with many multiplication operations. The benchmarks *cap*, *mac2*, *mults1*, and *mults2* are unable to be mapped to the reduced architecture. Though they all have less than ten multiply operations, no feasible mapping exists as shown by the ILP mapper – due to constrained routing.

All benchmarks finished mapping in less than ~12 minutes. The longest runtime for mapping is for *cap* on the Full architecture, followed by *mults1* and *mults2*. In these benchmarks, between 12 and 14
Table 4.4: Mapping results for both Full and Reduced architectures. M indicates the benchmark was able to be mapped on the architecture. U indicates that the benchmark was unable to be mapped on the architecture. Mapping runtimes for each architecture and benchmark are also reported. Statistics for the benchmarks are also included listing the number of compute operations, number of multiplication operations, and the number of I/Os. Ops is the total number of operations including multiplication operations.

4.8.3 FreePDK Standard-Cell Physical Implementation

In this work, we leverage the FreePDK45 standard-cell library \cite{73}, an open-source 45 nm standard-cell library, for the experimental study using standard-cells. The tool-generated Verilog of the targeted architectures is used for physical implementation and is put through the following flow: Synopsys’ Design Compiler is selected to perform technology mapping. Following technology mapping, we analyze area breakdown of targeted architectures. Architectures are mapped for minimum area and minimum delay. With the mapped design netlist as input, all architectures are pushed through standard-cell place and route, with Cadence’s Innovus, producing a standard-cell placement, parasitic information (SPEF file), as well as total chip area. Given the design netlist and SPEF file, we perform static timing analysis with Synopsys’ PrimeTime and determine each designs’ critical path.

For the standard-cell implementation, both architecture variants are synthesized targeting minimum area, as well as minimum delay, producing four sets of results. Soft constraint floorplanning shown in Figure 4.15 is applied to generate more regular designs. All four implementations are given the same aspect ratio. After place and route, we are able to extract the total chip area of all four designs shown in Table 4.5. Figure 4.16 shows the chip layout of all four designs, while Figure 4.17 demonstrates where each sub-module is placed after the tool-flow, which can be compared against the soft floorplan constraints shown in Figure 4.15.

Figure 4.18 presents the area breakdown of the four variants. Within the breakdown, the following
Figure 4.15: FreePDK45 45 nm standard-cell floorplan for Cadence’s Innovus place and route.

Figure 4.16: FreePDK45 45 nm standard-cell layout for all four designs in physical view.
Chapter 4. The CGRA-ME Framework

Figure 4.17: FreePDK45 45 nm standard-cell layout for all four designs in amoeba view, showing how the loose floorplanning constraints were met by the tool.

Figure 4.18: FreePDK45 45 nm standard-cell area breakdown of Full and Reduced architectures, when targeting minimum area and minimum delay.
Table 4.5: FreePDK45 45 nm standard-cell post-place-and-route area and performance of all four designs.

categorization is used: multiplexers of any size are routing resources, registers storing configuration bits are configuration memory, data registers and register files are data memory, and functional units are computation resources. From Figure 4.18, we observe that routing, configuration memory, and data memory are nearly constant across the architecture and optimization variants. Also, increasing computation capabilities within the processing elements results in the largest impact to the overall area required. Additionally, between the minimum area and minimum delay architectures, the multiplier area sees the largest increase.

Static timing analysis (STA) is performed to determine the design’s critical path using the SPEF file and netlist output from the place-and-route tool. Although the FMax generated using FreePDK45 may not be representative of an optimal design, comparisons between architecture variants can still provide relative conclusions. From the STA results, the inter-processing-element routing multiplexers and the multipliers are the bottleneck of the designs. The Full architecture’s additional torus connections contribute to its lower FMax due to wider multiplexers, compared to the Reduced architecture. Maximum operating frequencies of each design are shown in Table 4.5.

4.8.4 FPGA-Overlay Physical Implementation

We also target the 45 nm Alte ra Stratix IV FPGA (same process node as the standard-cell implementation) using Altera/Intel’s Quartus Prime ver. 16.1. A challenge that arose in the FPGA implementation pertains to floorplanning in the presence of CGRA blocks containing multipliers. It is desirable if the rows/columns of the CGRA’s physical realization form a regular grid coherent with the CGRA’s logical representation (e.g. CGRA blocks in column 2 are placed to the right of blocks in column 1, and so on). In the targeted Stratix IV FPGA, the DSP blocks containing hardened multipliers are arranged in columns that are relatively far apart. Interspersed among the DSP columns are comparatively numerous columns of standard LUT-based logic blocks, as well as occasional RAM-block columns. The floorplans of the CGRAs considered here are thus defined primarily in relation to the DSP columns, with the LUT-based blocks between the DSPs being underutilized.

Figure 4.19 shows the layout of the two architectures implemented in Stratix IV. Area and performance results are reported in Table 4.6. Area numbers reflect Stratix IV ALMs (adaptive logic modules),
Chapter 4. The CGRA-ME Framework

Figure 4.19: Floorplanned Stratix IV implementations of the Full CGRA (left), and Reduced CGRA (right).

<table>
<thead>
<tr>
<th></th>
<th>Full Arch</th>
<th>Reduced Arch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area [ALM]</td>
<td>8803</td>
<td>7192</td>
</tr>
<tr>
<td>Area [DSP]</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>FMax [MHz]</td>
<td>89</td>
<td>103</td>
</tr>
</tbody>
</table>

Table 4.6: FPGA area and performance for both architectures on Stratix IV.

each of which contains a dual-output fracturable 6-input LUT, carry-chain circuitry and two flip-flops. Logic blocks with multipliers use Stratix IV DSP blocks.

Figure 4.20 shows an area breakdown of ALMs used on the FPGA. Here only the data memory and configuration memory stay constant and we see an increase in both the computation area as well as routing area between the Reduced architecture and Full architecture.

4.9 Summary

In this chapter, we have introduced CGRA-ME, an open-source unifying CGRA framework that encompasses architecture description, architecture modelling, application mapping, and physical implementation. Two case studies were performed, exhibiting the capabilities of the framework. The CGRA-ADL language was demonstrated as a way for architects to describe a variety of CGRA architectures with varying components, sizes, and contexts. Both the simulated annealing-based mapper and the ILP-based
mapper were shown to be effective at mapping benchmarks, both hand generated and LLVM generated, to the various architectures. Additionally, CGRA-ME was shown to generate Verilog RTL for both a standard-cell and FPGA-overlay physical implementation flow. CGRA-ME is thus a powerful platform for CGRA architecture and CAD research that we believe will be useful to both the academic and industrial communities.

4.10 Acknowledgements

I would also like to acknowledge the many other colleagues that have collaborated to make CGRA-ME a success – Jim Zhao, Allan Rui, Noriaki Sakamoto, Alex Mertens, Steven Yin, Steven Niu, Matthew Walker, Prof. Jongeun Lee, and Prof. Jason Anderson. Without them, this project would not have been possible.
Chapter 5

CGRA Mapping

Coarse-Grained Reconfigurable Array (CGRA) architecture development necessitates adequate tooling. Following architecture development, application mapping techniques have also been studied. In the context of CGRAs, mapping refers to scheduling, placing and routing operations from an application onto a CGRA. In this chapter, we consider CGRA mapping for generic CGRAs, that is, both the application, as well as the CGRA architecture model are an input to the mapper. The two built-in mappers within the CGRA-ME framework, published in ASAP and DAC, are presented. The first mapper is simulated annealing-based and the second is constraint-based, formulated as an integer linear program (ILP).

Mapping for CGRAs has been a difficult problem, with the earliest works resorting to GUI-based tools for manual hand-mapping. More recently, many architectures and tools rely on a graph representation of the CGRA architecture within the mapping flows called a Modulo Routing Resource Graph (MRRG). The graph representation is very flexible and generically captures the capabilities of CGRAs with fixed-latency datapaths and multiple execution contexts, where CGRA behavior changes on a cycle-by-cycle basis and is discussed in Section 5.1.2.

In many prior works, a simulated annealing approach is used as the core algorithm to map an application onto a CGRA, though other heuristic techniques have also been developed. For example, prior works such as DRESC and SPR use simulated annealing as a core part of their mapping heuristics. However, there is no guarantee that simulated annealing approaches find a mapping solution, much less an optimal one, due to the random nature of annealing. For example, no solution would be found if annealing gets stuck in an infeasible minima.

Decisively knowing whether a mapping solution exists or not, and whether a solution is optimal,
can be of great benefit to architects and CAD experts. By using an ILP for CGRA mapping, one can provably determine mapping feasibility or infeasibility of a benchmark to a specific architecture, unlike heuristic methods that could be subject to random noise as in simulated annealing. Knowing feasibility or infeasibility of mapping, the complexity or amount of routing or storage structures can be tuned down to the limit of ‘mappability’ on an architecture for a very specific application domain – eliminating extra silicon area and power. Architects are able to assess precise limits of their architectures in terms of which applications or benchmarks are feasible and their performance. Architecture designs are isolated from mapping noise and there is no uncertainty for architects when tailoring architectures towards a basket of applications.

Since such a formulation creates a bound on what is achievable through heuristic methods, CAD experts benefit by being able to quantify the effectiveness of their own heuristic methods and through improvements in heuristic methods, the gap to the optimum can be narrowed. While constraint-based methodologies have been successfully employed in CAD tools for reconfigurable architectures and some specific CGRA designs, we believe this to be the first that applies directly to a generic CGRA model using an MRRG.

Since the mappers are integrated into CGRA-ME, the target architecture is not fixed nor templated as in many other works, but is described at a high-level and provided to the CGRA mapping tool. This flexibility permits mapping and evaluation of a wide range of CGRA designs allowing architects to independently evaluate architectures. This is analogous to the well-known VTR fined-grained FPGA architecture evaluation framework, where the underlying CAD algorithms are reactive to the user-provided architecture.

A background on CGRA mapping is given in Section 5.1 and in Section 5.2, we overview prior mapping techniques. Section 5.3 presents the simulated annealing-based mapper and Section 5.4 presents the ILP-based mapper. Experimental results comparing the ILP mapper to the simulated annealing mapper are shown in Section 5.5 and a chapter summary is provided in Section 5.6.

5.1 Background

5.1.1 Data-Flow Graph

A data-flow graph (DFG) is a directed graph $G(V, E)$, where vertices represent operations and edges are data values passed from one operation to the next (dependencies between operations). In many CGRA toolflows, a DFG is used to represent the kernel computations. The DFG accounts for all computation
operations, as well as memory accesses and I/O, including the data-dependencies and data-transfer between these operations. This graph representation is also able to capture loop-carried dependencies through back-edges within the DFG structure. Two example DFGs are shown in Section 5.4.

5.1.2 Modulo Routing Resource Graph

The Modulo Routing Resource Graph (MRRG) [67] is an abstract representation of the physical architecture of a CGRA and in this work we use an MRRG to model the CGRA during mapping. The graph contains, as vertices, all of the resources of the CGRA: the routes within the physical architecture, the storage elements, and the functional units that execute operations, including I/O and memory access. An MRRG is capable of modelling multiple contexts, which is a feature of some CGRAs. In these architectures, routes and/or functional units can be shared to perform different routes and/or operations on subsequent cycles in a repeating pattern, hence the MRRG is a ‘modulo’ graph that represents the resources available during multiple cycles of operation of the architecture. For example, with two contexts, the CGRA toggles between two configurations; with three contexts, the CGRA cycles through three configurations, and so on.

More formally, an MRRG is a directed graph $G(V, E)$, where each vertex, $v \in V$, represents a CGRA resource. There are two types of vertices in the graph, one set for the routing resources, $RouteRes$, and one for functional units, $FuncUnits$. These $FuncUnit$ nodes within the MRRG represent an execution time-slot of a physical functional unit – there could be multiple nodes that correspond to a single functional unit, but each node would represent execution on different device contexts and times. Each edge, $e \in E$, represents connectivity between an element of $FuncUnits$ and an element of $RouteRes$, or between two elements of $RouteRes$. The MRRG contains a replica of the device model graph for each context. Each one of these replicas represents the resources available for each cycle of execution. Edges between nodes in different replicas represent the capability to pass data from one context to another – from one cycle to the next. This is usually physically embodied as a register. For the case of $N$ contexts, there would be $N$ replicas, indexed from 0 to $N - 1$. Edges are present between some nodes in replica $i$ and replica $i + 1 \mod N$, representing the ability to produce data in a context, and consume the data in the subsequent context. For example, in the two context case, a node representing a register in context 0 may have an edge to a node representing a computational unit in context 1, embodying the ability to compute a value in context 0 and then consume it in the next context. We illustrate some of the properties of the MRRG in the following examples.

Figure 5.1 shows the translation of a multiplexer and register primitive to their respective MRRGs.
The 2-to-1 multiplexer shown can provide routing of values between functional units. The 2-to-1 multiplexer in this example is dynamically reconfigurable – it is able to route from different inputs in different cycles or contexts. The MRRG shown contains four nodes per cycle, two input and one output RouteRes and an internal multiplexing RouteRes that guarantees exclusivity to a single input. That is, on any cycle, only one input can be routed to the output. Since it is dynamically reconfigurable and this multiplexer may be reused on subsequent cycles for different routes, multiple copies of this four node structure are present for each cycle (each context).

In the case of the register in Figure 5.1, it is modelled in the MRRG as a special wire as it moves a value from one cycle to the next. So its input node starts in cycle $i$ and its output node ends in cycle $i + 1$. Each cycle this register can be reused, so this pattern is repeated for each cycle.

Figure 5.2 shows the translation of three functional units, that perform a multiplication operation, with different latencies (L) and initiation intervals (II) and their respective MRRGs. The first example shows a functional unit that performs a 1-cycle multiply (latency of 1 cycle and initiation interval of 1 cycle). The MRRG for this unit consists of two input RouteRes vertices that represent the input operands of the multiply, a functional unit resource node that represents usage of the hardware at the associated timeslot, and an output RouteRes vertex. Since the functional unit has a latency of 1-cycle, the output vertex is in the subsequent cycle. Since the initiation interval is 1-cycle, this functional unit can take inputs every cycle, so the MRRG is replicated every cycle.

The second example in Figure 5.2 shows a functional unit that performs a multiply that takes 2-cycles (latency of 2 cycles) with no pipelining (initiation interval of 2 cycles). Here, we have a similar structure
but the output node is delayed by a second cycle and instead of repeating this structure every cycle, we repeat every 2 cycles since this resource is only available every two cycles (initiation interval of 2 cycles).

In the last example in Figure 5.2, we show a functional unit performing multiply with 2-cycle latency but is fully pipelined (initiation interval of 1 cycle). Again, we have a similar structure as the second example, but the structure is repeated every cycle since the unit is available to perform a multiply every cycle.

To further illustrate how an MRRG is generated for a composite component, Figure 5.3 shows an example functional block datapath and its corresponding MRRG, showing how a larger MRRG is generated from primitive components. In the example, two input multiplexers feed a functional unit that feeds an output register. The multiplexers’ MRRGs are composited with the functional unit’s MRRG along with the register’s MRRG. The register’s output feeds the MRRG corresponding to the next context or execution cycle, while the output of the register in the current context is fed with the previous cycle’s data.

5.1.3 Mapping

CGRA mapping involves associating the operations and their connectivity within the DFG to the MRRG. The overall CGRA mapping problem is simplified through the use of the MRRG, as the MRRG frames
Figure 5.3: An example functional block that contains a functional unit (with a latency of 0 cycles), register and input multiplexers, and its corresponding MRRG structure for 1 cycle / 1 context.
the constraints of the modulo scheduling problem, operator placement, and data routing, within the structure of the graph itself. Associating the DFG to the MRRG involves placing each operation in the DFG on a valid MRRG \textit{FuncUnit} node, as well as finding a valid data routing through the \textit{RouteRes} nodes to respective operations placed on other \textit{FuncUnit} nodes. Other works also refer to this process as \textit{embedding} [86]. As long as all operations can be mapped to functional units, while having valid routes between functional units, a valid/feasible mapping exists, and the architecture is able to implement the benchmark application.

While the input DFG is constant, the MRRG will change depending on the number of contexts the device is configured for. A higher number of contexts means higher initiation intervals (II), reducing throughput. Generally, mapping of a DFG is started on the smallest II first, and if it cannot be mapped, the II can be increased, effectively increasing the number of execution slots for the functional units – time-multiplexing the hardware.

The upstream processes, such as \textit{compilation} from higher level languages (such as assembly or C) and downstream processes (such as generating the configuration for the CGRA itself along with the necessary control), while necessary, are not considered part of the mapping processing in our context.

5.2 Related Work

Mei et al. first proposed the Modulo Routing Resource Graph (MRRG) model in DRESC [67]. This novel work frames the constraints of the modulo scheduling problem, operator placement, and value routing, within the graph itself and subsequent works have capitalized upon this abstraction. The core algorithm within DRESC is simulated annealing. Following DRESC, SPR [68] uses a similar simulated annealing method with some additions.

More recently, many works have examined graph-based approaches to mapping. Park et al. [91] used a graph theory technique called \textit{graph embedding} to draw the target application onto a three dimensional target space representing functional units over time. Park et al. [92] followed on with edge-centric modulo scheduling. In this method, routing of values is prioritized, where operator placement is secondary. Chen et al. propose a \textit{graph minor} approach to mapping [83]. Their algorithm involves node reductions on the MRRG to test if an application graph is a is minor of the MRRG. Yet another graph-based technique is proposed by Ma et al. [87].

Lee et al. [94] presents two algorithms, not based on an MRRG approach, that are quite specific to their proposed architecture. The first is an architecture-specific ILP approach that optimizes for latency. The second approach is a list-scheduling and quantum-inspired evolutionary algorithm. Yoon et al. [99]
also incorporate ILP techniques into their work and Nowatzki et al. \cite{95} present an ILP formulation for three specific template architectures. The major difference between other ILP formulations and the ILP formulation within CGRA-ME is that the formulation is valid over any architecture from which an MRRG can be generated.

Formal techniques have been applied in other areas of CAD for reconfigurable hardware, for example, Boolean satisfiability has been explored as a direction for FPGA routing \cite{96, 98}. As is shown in Section \ref{5.4}, our ILP formulation is specifically a zero-one integer linear program or binary integer linear program formulation. These two methods are closely related as it has been shown that a zero-one integer linear program can be readily transformed into a Boolean satisfiability problem in conjunctive normal form (CNF) \cite{100}.

### 5.3 Simulated Annealing Mapper

The simulated-annealing mapper implemented within our framework allows for flexibility to map data-flow graphs to architectures modelled within CGRA-ME. Specifically, it handles a variety of inter-connectivity and varied functional units, demonstrated in Section \ref{4.7}. The overall algorithm is outlined in Figure \ref{5.4}. Mapping is considered complete as soon as a legal placement and routing is found.

```c
0: bool Map(DFG * dfg, MRRG* mrrg) {
1:     InitialPlaceAndRoute(dfg->nodes, mrrg);
2:     forever {
3:         for(i = 0; i < 100 * count(dfg->nodes); i++) {
4:             old_cost = getTotalCost(mrrg);
5:             old_mapping = getMapping(mrrg);
6:             op = getRandomOp(dfg);
7:             candidate_fu = getRandomFU(mrrg, op);
8:             current_fu = getMappedFU(op);
9:             SwapPlaceAndRoute(op, current_fu, candidate_fu);
10:            new_cost = getTotalCost(mrrg);
11:       if(!accept(old_cost, new_cost, temperature)
12:           restoreMapping(mrrg, old_mapping);
13:       }
14:     if(NoOverUse(mrrg))
15:         return true;
16:     else if (temperature < THRESHOLD)
17:         return false;
18:     else
19:         temperature = updateTemp(temperature);
20:     }
21: }
Figure 5.4: Pseudo-code for the simulated annealing mapper.
```
On Line 1 in Figure 5.4, we begin with initial placement. Every operation node in the DFG is placed onto a functional unit that is able to perform the operation specified by the operation node. For example, a multiply operation will never be placed on a functional unit that can only do additions. However, temporary overuse of functional units is permitted – multiple operation nodes may be placed on a single functional unit. Likewise, routes between operation nodes are allowed to be temporarily overused in a PathFinder-like fashion \[60\]. Overuse of resources and interconnect is penalized through costing during the annealing process and gradually removed.

The cost of a mapping is the summation of all used routing nodes and all used functional unit nodes within the MRRG. Along with this base cost, an additional penalty is applied for each routing or functional unit node that is overused. Overall, the cost of any node in the MRRG is given as: $\text{Occupancy} + \text{PenaltyFactor} \times \text{ExcessOccupancy}$. Initially, we begin with a $\text{PenaltyFactor}$ of 1. Subsequently, this coefficient is exponentially increased as the temperature of the annealing falls.

After initial placement, we enter the main simulated annealing loop on Line 2 of Figure 5.4. On Line 3, for each temperature, we try ($100 \times$ the number of nodes in the DFG) swaps. For each swap, we record the old cost and mapping (Lines 4 & 5) and find a random operation node in the DFG, \text{op} and new candidate functional unit that is able to support the operation node (Lines 6 & 7). The functional unit where the operation node is currently placed is located (Line 8) and then we perform the swap (Line 9). \text{SwapPlaceAndRoute} places the \text{op} on the \text{candidate_fu}, then routes its associated inputs and outputs. Then, one of the operation nodes on the candidate functional unit is ripped-up, placed and routed on the \text{current_fu}, only if operation can be supported. Then the \text{new_cost} of the \text{mrrg} is calculated on Line 10. We accept the new state of the mapping if the new cost is lower than the old cost or probabilistically, dependent on the current annealing temperature \[31\].

After the swapping phase, we check to see if we have any overused routing or functional unit nodes in the \text{mrrg} (Line 15). True is returned if we have found a solution. False is returned if the temperature is under a set threshold. Otherwise, the temperature is updated and we do another round of swapping.

### 5.4 Integer Linear Programming Mapper

In this section, we describe the formulation of variables, constraints and the objective function for the ILP mapper.
Chapter 5. CGRA Mapping

Figure 5.5: Two DFG fragments to illustrate value routing/fanout.

Figure 5.6: Three illustrative MRRGs.
5.4.1 Definitions

We first define four sets of items; the first two relate to the CGRA device model, the second two relate to the application being mapped into the CGRA:

- **FuncUnits**: contains all execution slots of every functional unit within the architecture – each corresponds to one functional-unit node within the MRRG.

- **RouteRes**: contains all routing resources (wire, bus, multiplexer, or register) within the architecture: each corresponds to one routing resource node within the MRRG.

- **Ops**: all operations in the DFG to be mapped.

- **Vals**: all values produced by operations in the DFG to be mapped. Each Vals is split into SubVals. A sub-value represents a source to sink connection in a multi-fanout value. For example, there are two sub-values for the two fanout nets in DFG B in Figure 5.5.

We define three sets of binary variables. The first set of variables define the placement of operations onto functional units (a mapping from Ops to FuncUnits). The second set of variables define the use of routing resources by values (a mapping from Vals to RouteRes). The third set of variables are abstract, and define the routing path of values between functional units for each sink of a multi-fanout value. For each value, we create a new binary variable that associates routing resources, the value, and the termination point (sink). These variables are necessary for the formulation, as will be elaborated upon below. The three sets of binary variables are:

- **F_{p,q}**: functional unit node \( p \) in the MRRG is used for supporting operation \( q \) in the DFG.

- **R_{i,j}**: routing node \( i \) is used for routing value \( j \).

- **R_{i,j,k}**: routing node \( i \) is used for routing value \( j \) to value \( j \)'s sink \( k \).

As will be apparent in the formulation below, \( R_{i,j} \) will be constrained to 1 whenever \( R_{i,j,k} \) is 1 (i.e. for any sink \( k \)). The sink-specific variables, \( R_{i,j,k} \), are required to achieve routing connectivity to each sink; the sink-agnostic variables, \( R_{i,j} \), are used to ensure no routing-resource overuse and in the ILP objective function to minimize overall resource usage.
5.4.2 ILP Constraints and Objective Function

**Operation Placement:** Every operation in the DFG is placed on exactly one functional unit.

\[
\sum_{p \in FuncUnits} F_{p,q} = 1, \forall q \in Ops
\]  

(5.1)

**Functional Unit Exclusivity:** Each functional unit slot (represented by \(FuncUnits\)) is occupied by at most one DFG Operation (i.e. there exist no overlaps among operations on functional units).

\[
\sum_{q \in Ops} F_{p,q} \leq 1, \forall p \in FuncUnits
\]  

(5.2)

**Functional Unit Legality:** Ensure that operations are only placed on functional units that can implement the operation (applies to heterogeneous architectures). Where \(SupportedOps(p)\) is the set of operations that are supported by functional unit \(p\).

\[
F_{p,q} = 0 \quad \forall p \in FuncUnits, q \in Ops
\]  

where: \(q \notin SupportedOps(p)\)  

(5.3)

**Route Exclusivity:** Ensure that each routing resource is occupied by at most one value (i.e. multiple DFG Values cannot be routed on a single routing resource).

\[
\sum_{j \in Vals} R_{i,j} \leq 1, \forall i \in RouteRes
\]  

(5.4)

**Fanout Routing:** Guarantee continuity of values between adjacent routing resources. We ensure for each fanout of a routing resource used by a value, there is at least one downstream routing resource that is used by the same value, whether this be another routing resource or the sink of the fanout (which is itself a routing resource). So, at least one output of a routing resource node must be driven with the same value, or it must terminate at the input of the downstream functional unit node.

\[
R_{i,j,k} \leq \sum_{m \in \text{fanouts}(i)} R_{m,j,k}
\]  

\[\forall i \in RouteRes, \forall j \in Vals, \forall k \in \text{sinks}(j)\]  

(5.5)

**Implied Placement:** Ensure that Fanout Routing terminates at the input of a functional unit, thereby
implying a mapping of the downstream operation to the functional unit. Here, we use the \( \rightsquigarrow \) symbol to denote the existence of an edge between two nodes in a directed graph. This constraint means that if the route for sink \( k \) of value \( j \) terminates at functional unit \( p \), then operation \( q \) must necessarily be mapped onto functional unit \( p \). This constraint also accounts for operand correctness in the case of non-commutative operations.

\[
F_{p,q} \geq R_{i,j,k}
\]
\[
\forall p \in \text{FuncUnits}, \forall q \in \text{Ops}, \forall i \in \text{RouteRes}, \forall j \in \text{Vals}
\]
where: \( \exists (j \rightsquigarrow q) \land (i \rightsquigarrow p) \)
\[
\forall k \in \text{sinks}(j)
\]
(5.6)

**Initial Fanout**: Ensure that the routing resources at the output of a functional unit are set to the output value of the mapped operation. The binary variable \( R_{i,j,k} \) is set for each sink, \( k \), of the output value. This constraint is only applied when there is an edge from \( q \) to \( j \) in the DFG and an edge from \( p \) to \( i \) in the MRRG. Again, we use the \( \rightsquigarrow \) symbol to denote the existence of an edge between two nodes in a directed graph.

\[
R_{i,j,k} = F_{p,q}
\]
\[
\forall i \in \text{RouteRes}, \forall j \in \text{Vals}, \forall p \in \text{FuncUnits}, \forall q \in \text{Ops}
\]
where: \( \exists (q \rightsquigarrow j) \land (p \rightsquigarrow i) \)
\[
\forall k \in \text{sinks}(j)
\]
(5.7)

**Example 1**: Consider mapping DFG A in Figure 5.5 to MRRG A in Figure 5.6. If \( Op_1 \) is placed on \( FuncUnit_1 \), \( F_{1,1} = 1 \). Since the Value in DFG A has only one fanout, the Initial Fanout constraint ensures that \( SubValue_1 \) is associated with \( RoutingRes_1 \) with \( F_{1,1} = 1 = R_{1,1,1} \). The Fanout Routing constraint then ensures that at least one of \( R_{2,1,1} \) or \( R_{3,1,1} \) is equal to 1. Application of the Implied Placement constraint on \( RoutingRes_2 \) and \( RoutingRes_3 \) allows the routing to terminate at \( FuncUnit_2 \) or \( FuncUnit_3 \) setting \( F_{2,2} = 1 \) or \( F_{3,2} = 1 \), placing \( Op_2 \).

**Routing Resource Usage**: Since routing is formulated on a sink-by-sink basis using sub-values, the
following constraint ensures that routing resources are associated with the corresponding values.

\[ R_{i,j} \geq R_{i,j,k} \]
\[ \forall i \in \text{RouteRes}, \]
\[ \forall j \in \text{Vals}, \]
\[ \forall k \in \text{sinks}(j) \] \hspace{1cm} (5.8)

**Multiplexer Input Exclusivity**: Prevents self reinforcing routing loops that would terminate fanout routing within the loop instead of the required sink of the route. By disallowing multiplexer inputs from having the same value, loops are prevented.

\[ R_{i,j} = \sum_{m \in \text{fanins}(i)} R_{m,j} \]
\[ \forall i \in \{\text{RouteRes} \mid |\text{fanins}(i)| > 1\}, \]
\[ \forall j \in \text{Vals} \] \hspace{1cm} (5.9)

**Example 2**: Consider mapping DFG A in Fig. 5.5 to MRRG B in Fig. 5.6. If \( Op_1 \) is placed on \( \text{FuncUnit}_1 \) \( (F_{1,1} = 1) \), Initial Fanout and Fanout Routing would set \( R_{2,1,1}, R_{4,1,1} \) and \( R_{5,1,1} \) to 1. Routing can now continue through more routing resources in a cloud of connected routing resources in \( C_1 \) or \( C_2 \). Without the Multiplexer Input Exclusivity constraint, routing through \( C_1 \) and setting \( R_{1,1,1} = 1 \) is feasible. Now, the Fanout Routing constraint for \( \text{RouteRes}_1 \) is already satisfied since \( R_{4,1,1} = 1 \) and \( \text{SubValue}_1 \) has not been routed to any \( \text{FuncUnit} \). By applying the Routing Resource Usage constraint and Multiplexer Input Exclusivity constraint, \( R_{1,1,1} = 1 \) is infeasible because \( R_{2,1,1} = 1 \). Fanout Routing at \( \text{RouteRes}_5 \) can now only be satisfied by routing through \( C_2 \), ultimately to the sink at \( \text{FuncUnit}_2 \).

**Example 3**: Consider mapping DFG B in Figure 5.5 to MRRG C in Figure 5.6. \( \text{Val}_1 \) has two fanouts – one to \( \text{Op}_2 \) and one to \( \text{Op}_3 \). Consider applying our routing constraints to \( \text{Values} \), instead of \( \text{SubValues} \). \( \text{Op}_1 \) is placed on \( \text{FuncUnit}_1 \) \( (F_{1,1} = 1) \), the Initial Fanout constraint would set \( R_{1,1} \). Going through \( C_1, R_{2,1} \) could be 1 fulfilling the Fanout Routing constraint and with the Implied Placement constraint, mapping either \( \text{Op}_2 \) or \( \text{Op}_3 \) to \( \text{FuncUnit}_2 \). If \( \text{Op}_2 \) is mapped to \( \text{FuncUnit}_2 \) \( (F_{2,2} = 1) \), \( F_{3,3} = 1 \) due to the Operation Placement constraint. Now, there is no other constraint that guarantees a connection of the value through \( C_2 \) and \( \text{RouteRes}_3 \) to \( \text{FuncUnit}_3 \). Hence, each sink is assigned a distinct \( \text{SubValue} \) for routing.
The objective function we use is minimizing the number of routing resources used by the mapping.

\[
\text{Minimize } \sum_{i \in \text{RouteRes}, j \in \text{Vals}} R_{i,j}
\]  \hspace{1cm} (5.10)

However, it is straightforward to apply alternative objective functions, where, for example, specific types of resources have unique costs. For example, one can imagine that resources such as long wires, registers, register files or other data value routing structures contribute significantly to power consumption and these nodes could be weighted to optimize for power.

## 5.5 ILP Mapper Evaluation

### 5.5.1 Test Architectures and Benchmarks

To test our formulation, a number of benchmarks and architectures were considered. The architectures were chosen to be representative of CGRAs proposed in the literature with varying flexibility. Since higher degrees of flexibility generally increases hardware costs, it is interesting to see how much flexibility is enough to map a set of benchmarks. Each architecture comprises a $4 \times 4$ 2D-array of functional blocks with 32-bit-wide bus-based interconnect. Each block within our test architecture, shown in Figure 5.3, contains a single functional unit ALU, an output register, and block I/O. The functional unit within the block can perform RISC-like operations such as \texttt{add}, \texttt{mul}, \texttt{shl}, etc. Each row within the array shares a memory port as shown in Figure 5.7. This port is modelled as a special functional unit that can only perform \texttt{load} and \texttt{store} operations. The array is also surrounded by I/O blocks that are also modelled as special functional units performing \texttt{input} and \texttt{output} operations. We consider two functional block architectures and two interconnect architectures with multiple contexts. One set of functional block architectures, Homogeneous, has full fledged ALUs including a multiplier, whereas the Heterogeneous architectures, only half of the ALUs in the architecture contain a multiplier. One style of interconnect architecture is Orthogonal, with each block having connectivity with the four ordinal directions. This connectivity is also depicted in Figure 5.7. The other style of interconnect architecture is Diagonal, with each block having additional connectivity to diagonal blocks. For Diagonal interconnect, the size of each functional block’s input multiplexer was increased to accommodate the additional inputs. Additionally, we model these architectures with 1 and 2 execution contexts, enabling mappings with II=1 and II=2.

The benchmark data-flow graphs were chosen to have varying number of operations, number of multiply operations and routing requirements. Table 5.1 shows detailed characteristics of each benchmark. The benchmarks reflect different mapping difficulties for the target architectures and they demonstrate
Chapter 5. CGRA Mapping

5.5.2 Mapping Flow

The overall flow of mapping is shown and described in Figure 5.8. CGRA-ME was used for generic specification of architectures. Detailed functional block and routing structures are constructed and modelled with higher-level connectivity. From this architecture model, the framework automatically generates a corresponding MRRG model for the mapper for a specified number of contexts.

To solve the ILP formulation, the Gurobi solver was integrated into the overall framework using their C++ API. This sophisticated solver is free-to-use for academic use.

5.5.3 Mapping Results

The ILP solver was able to determine feasibility/infeasibility for all formulations of benchmark/architectures except four that timed out after 24 hours. However, more than 80% of the runs completed within one hour.
Table 5.1: Benchmarks

Table 5.2 shows the full results of the ILP mapper over the eight CGRA architectures and 19 benchmarks. The single context architectures on the left of the table exhibit varying degrees of success to map each benchmark. The Heterogeneous Orthogonal architecture, the most constrained architecture, was only able to accommodate 5 benchmarks, whereas the most flexible, Homogeneous Diagonal, was able to map 14. It is also interesting to note that quite a few benchmarks were not able to be mapped in a single context (cos_4, cosh_4, sinh_4, and extreme). The Heterogeneous Diagonal and Homogeneous Orthogonal architectures show interesting trade-offs between reduced multipliers with extra routing and many multipliers with constrained routing. The mult_14 and mult_16 benchmarks both require many multipliers and routing, and are both unmappable. However, the smaller mult_10 benchmark is able to map to the Homogeneous Orthogonal architecture.

All benchmarks were able to be mapped in dual context architectures aside from the 4 solver timeouts. This shows the flexibility that a second context gives at the price of halved throughput (since $II = 2$) and extra hardware (and power) to support the second configuration context. If strict mappability for the given benchmark set is the architect’s concern, a Heterogeneous Diagonal architecture (that contains half the multipliers of a Homogeneous architecture) with support for two contexts may be sufficient, since all benchmarks can be mapped to this architecture. Additionally, 9 of the benchmarks could still be mapped with higher throughput ($II = 1$), while the other 10 would need to be mapped with two configuration contexts ($II = 2$). But if performance is a concern, a Homogeneous architecture with Diagonal routing will be the best option since it able to map the most benchmarks with $II = 1$. 
## Chapter 5. CGRA Mapping

### Table 5.2: Mapping Results for the ILP mapper

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Single Context (II=1)</th>
<th>Dual Context (II=2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mac</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>add_10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>add_14</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>add_16</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>mult_10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mult_14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mult_16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2x2-f</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2x2-p</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cos_4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cosh_4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>exp_4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>exp_5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>exp_6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sinh_4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tan_4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>extreme</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>weighted_sum</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total Feasible</td>
<td>5</td>
<td>9</td>
</tr>
</tbody>
</table>

1 signifies a feasible mapping, 0 signifies an infeasible mapping. T, signifies a solver timeout where the solver was unable to find a feasible solution or prove infeasibility.
Chapter 5. CGRA Mapping

Figure 5.8: The mapping flow within our framework for ILP-based mapping and simulated annealing-based mapping. The architecture description and the number of contexts is input into the framework, where it generates an MRRG. For ILP-based mapping, the ILP formulation is created from the input DFG benchmark and the MRRG. The formulation is then solved by the ILP Solver, in our case Gurobi. The simulated annealing-based mapper takes the DFG and MRRG directly as inputs to map the benchmark to the architecture.

For contrast, the simulated annealing based mapper was run with moderate parameters (number of inner-loop iterations, penalty factors, temperature schedule, etc.) on the same set of benchmarks and architectures. A comparison of the results of both mappers is shown in the bar graph in Figure 5.9. The ILP mapper is able to find more mapping solutions for all eight architectures even after accounting for the 4 ILP solver timeouts. This highlights one of the drawbacks of the simulated annealing mapper, as it is not able to map in many of the cases.

5.6 Summary

In this chapter we considered mapping for generic CGRAs using an MRRG as an architecture model. Two mappers built within CGRA-ME were presented. The first using a simulated annealing approach...
and the second using an ILP formulation. To our knowledge, this is the first ILP formulation of the CGRA mapping problem using a generic MRRG model. The presented ILP formulation allows one to provably determine mapping feasibility or infeasibility and produce an optimal mapping. Architects are able to evaluate the ‘mappability’ of the architectures for sets of domain-specific benchmarks, allowing them to tune the flexibility of the architecture, while staying isolated from CAD tool effects. CAD experts are more easily able to evaluate and improve upon the effectiveness of heuristic methods, since the ILP formulation forms a bound on what is achievable. This ILP formulation was shown to be an effective way of determining mapping feasibility of applications to architectures through an evaluation study.
Chapter 6

Conclusions

6.1 Summary

As the silicon industry is faced with continuous transistor-count growth and increasing power density, traditional CPU scaling has not been able to match computing demands. Specialized hardware and reconfigurable computing pose one solution to dark silicon [15], owing to their greater efficiency as compared to traditional processor centric computing. Reconfigurable hardware, specifically Field-Programmable Gate Arrays (FPGAs), has long been an effective means of implementing specialized hardware. As transistor counts have increased, the amount of logic gates available on FPGAs has also increased. These larger FPGAs have now been deployed in data-centres as workload accelerators. Though more efficient than processors, FPGAs still are 3−5× slower and 20−40× less area-efficient than ASICs [1]. Another type of emerging reconfigurable computing architecture, Coarse-Grained Reconfigurable Arrays (CGRAs), also pose a solution to diminished CPU scaling and dark silicon. CGRAs being more specialized than FPGAs have the potential to be more efficient and closer to Application Specific Integrated Chips (ASICs) on the spectrum of programmability. Less studied and developed as compared to FPGAs, the prospect of CGRAs poses its own challenges and opportunities for innovation. Targeting improvements to reconfigurable architectures and related tools, this dissertation is summarized below.

Attacking the gap between FPGAs and ASICs, Chapter 2 presented an architectural study on hybrid logic structures for FPGAs. An alternative logic element to Look-Up-Tables (LUTs) that costs ~10% the area of a 6-LUT was proposed, along with improved technology mapping. This element was integrated into non-fracturable and fracturable LUT architectures in a hybrid fashion alongside traditional LUTs in a Configurable Logic Block (CLB). Architecture sweeps were performed with varying ratios of hybrid
Chapter 6. Conclusions

elements using the new technology mapping algorithm to evaluate architectures. The preliminary study focused on non-fracturable architectures and was published at the IEEE International Conference on Field-Programmable Technology (ICFPT) [26]. The comprehensive study that considers fracturable architectures was published in IEEE Transactions on Very Large Scale Integration Systems (TVLSI) [27].

Chapter 3 illustrated the potential of CGRAs as a more efficient platform for computing, giving background on CGRA design and on prior CGRA architectures. While other types of computing architectures have open-source tools available for modelling and evaluation, prior to CGRA-ME there were no other frameworks publicly available for CGRAs. Chapter 4 presented this new open-source CGRA-ME framework for the modelling and evaluation of CGRAs that was developed to further research within the CGRA field. In this chapter, the components and design of the framework were discussed with the use of the framework demonstrated through two architecture studies. The two studies demonstrated the framework’s ability to model a variety of architectures, compile and map benchmarks, and produce RTL designs of hypothetical CGRA architectures. This framework was first published in the proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP) [29]. And a follow-up study was presented in the proceedings of the ACM/SIGDA International Symposium on Physical Design (ISPD) [30] that included the modelling of more complex architectures and the implementation of CGRA overlays using FPGAs.

As with FPGAs, CGRAs require support from CAD algorithms to target the architectures. Chapter 5 presented the mappers developed for CGRA-ME – one based on simulated annealing and another that utilizes a new ILP formulation. The architecture-agnostic ILP-based mapper utilizes the generic Modulo Routing Resource Graph (MRRG) model, allowing it to be flexible over differing architectures. Moreover, the ILP-based mapper provides a definitive mappable or unmappable answer. This property of the mapper allows architectures to be developed in isolation of mapper noise in the case of probabilistic algorithms and also non-optimal heuristics. The ILP-based mapper formulation will be presented at the Design Automation Conference in 2018 [32].

6.2 Future Work

While FPGAs continue to be developed and produced, we are optimistic that CGRAs will gain more traction in the coming years. CGRAs seem promising as a computing platform, though considerable architectural questions are still open. There is much further research to be done though application studies in the up-and-coming area of machine learning, as well as wireless applications – two domains that will shape technology in the coming decade. It is also unknown where CGRAs may find a niche,
whether in low-power applications such as in the area of the internet-of-things, or alongside FPGAs as high-performance accelerators in commercial data-centres or baseband towers. For these applications and areas, a number of design parameters need to be considered. Within the microarchitecture of the CGRA, there are many trade-offs between having larger arrays or having smaller arrays with many configuration contexts. Larger arrays will require larger area, but more contexts also require more configuration memory. Parameters for the amount of routing and storage required for devices also change depending on the application area and overall CGRA design and ought to be tuned. Functional unit optimizations exist to be explored. Optimizations such as pipelining of the functional units for increased throughput could be beneficial, though it does pose interesting CAD mapping challenges.

Within CGRA-ME, scheduling is currently assumed to be static with no branches in the data-flow. One avenue for further research is the predication of operations, which allows for divergence while fitting well within static scheduling schemes. There are also architectures that include dynamic control flow and the modelling of these styles of architecture is also left to be explored.

The media for implementation will also affect these parameters, whether it is through standard cells or as an FPGA overlay. With the recent widespread adoption of FPGAs within data-centres, the path using FPGA overlays is conceivable. In this context, overlays would aim to reduce compilation times and allow users to deploy applications that rapidly utilize the FPGA hardware. Especially when considering FPGA overlays, pipelining of functional units becomes a priority as the DSP units on FPGAs perform best when pipelined.

Currently, CGRA-ME focuses on the microarchitecture of CGRAs but real-world systems need global infrastructure. Memory access and data movement accounts for much of the energy spent and latency of applications and require due consideration.

6.3 Conclusion

Facing the challenges of silicon efficiency and dark silicon, reconfigurable architectures in computing have become widely adopted into commercial data-centres. Towards the future, we are confident that the uptake of reconfigurable architectures will continue, requiring further innovation and development by researchers.

In this dissertation, the developments on FPGA architectures show that hybrid architectures push the boundary of what traditional FPGA architectures can achieve with respect to FPGA logic density, thereby closing the gap towards ASICs. CGRAs, already positioned closer to ASICs than FPGAs for some time now have been considered...
as an alternative to FPGAs, however their general uptake has been much slower in both industry and academia. There is considerably less research on CGRAs as compared to FPGAs. There are many factors that could contribute to this lack of research – too many for speculation. Though, one identifiable reason is the lack of a consistent evaluation framework within the research community. Promoting further CGRA research, CGRA-ME fills the gap of a platform for CGRA development.

The development of the architecture-agnostic ILP-based CGRA mapper within CGRA-ME first, furthers CAD tools within the field and second, demonstrates that CGRA-ME is flexible enough to be extended with new mappers. Specifically, the new mapper allows researchers to develop architectures in isolation of mapper noise and allows for an upper-bound on mappability for other heuristic mappers.

Through this dissertation, we have made headway towards more efficient reconfigurable architectures through architecture design and related CAD and are optimistic that these contributions will have positive impact on further research and industrial application of reconfigurable architectures.
Bibliography


[50] Personal communication with Altera.


