Improving the Energy Efficiency of Coarse-Grained Reconfigurable Arrays

Brian C. Van Essen

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2010

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Brian C. Van Essen

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of the Supervisory Committee:

_____

William H.c. Ebeling

_____

Scott Hauck

Reading Committee:

_____

William H.c. Ebeling

_____

Scott Hauck

_____

Luis Ceze

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Improving the Energy Efficiency of
Coarse-Grained Reconfigurable Arrays

Brian C. Van Essen

Co-Chairs of the Supervisory Committee:
Professor William H.c. Ebeling
Computer Science and Engineering

Professor Scott Hauck
Electrical Engineering

Coarse-grained reconfigurable arrays (CGRAs) are a class of spatial accelerators that combine advantages from both FPGAs and VLIW processors to efficiently exploit the computationally-intensive inner loops of an application. This thesis systematically explores a region of the design space for CGRAs and identifies architectural features that can improve the architecture's overall energy efficiency, without sacrificing performance. Using the results from this exploration we developed the Mosaic CGRA, which is an energy-optimized architecture that can meet the industry's demand for dramatic improvements in energy-efficient computing, by boasting a 10× reduction in its area-delay-energy product versus a baseline architecture.

We accomplish this improvement in energy efficiency by systematically isolating the impact of each architectural feature on both the architecture's physical characteristics and its runtime performance, rather than performing an ad-hoc exploration of the design space. As a result, the architect is able to compare the relative advantages of each feature and compose them more easily than has been traditionally possible with past architectural explorations. Not only does this evaluation provide the foundation of the Mosaic CGRA, but provides an infrastructure, and a reference design for future architecture development.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Graduate school has been a long and challenging journey for me, and I would like to express my gratitude and give thanks to everyone who helped me through it.

First and formost, I would like to thank my wife, LeAnn, both for encouraging me to pursue my Ph.D. and for supporting me through the journey. I would also like to thank my advisors, Carl Ebeling and Scott Hauck, for their guidance, patience, and mentorship. Both of you have given me every opportunity to succeed and taught me a great deal about being a researcher, from learning how to challenge my own convictions to how to defend them.

Furthermore, I would like to thank all of the members of the Mosaic research group, past and present: Allan Carroll, Stephen Friedman, Robin Panda, Aaron Wood, and Benjamin Ylvisaker. I could not have completed this without all of your hard work, intellectual stimulation, and camaraderie. In particular, Benjamin's path has paralleled mine for nearly eight years, through a startup, a shutdown, and two graduate programs; he has been a good friend throughout journey.

Along the way, there have also been a number of good friends, great teachers, and excellent support in the CSE and EE departments; in particular, Lee Daemon, who with the help of the VLSI/ACME support staff, was able to kept the clusters running, despite my best efforts. Also, I would like to thank both the Department of Energy and the National Science Foundation for their generosity in providing grants that allowed me to pursue this research.

Last, but not least, I would like to thank Jonathan and Anna for their patience and for understanding when daddy could not play games; and my parents, because without their unbounded support, generosity, and encouragement, this would not have been possible.

DEDICATION

I would like to dedicate this work to my family,
who have provided joy and companionship along this journey,
but especially to my father,
who has always been the consummate scientist.

Chapter 1

INTRODUCTION

Computationally intensive applications continue to grow in size, but most importantly in scope and distribution. From supercomputers to battery powered embedded systems, the ubiquity of computing is demanding increased performance, but also increased energy efficiency. Many of the most actively researched and computationally demanding applications, with wide commercial deployment, are drawn from the digital signal processing, streaming data, and scientific computing application domains. While Moore's law continues to provide increasing transistor density on a chip, commensurate increases in power density now limit how effectively these additional transistors can be utilized. Concerns with power density in data centers [1], battery life for unmanned vehicles and mobile devices [2, 3], and the ever rising cost of energy and cooling [4], have motivated computer architects to focus on specialized architectures as accelerators for performance- and energy-bound applications.

The challenge of energy-efficient computing has led to a resurgence in research on spatial architectures and parallel programming models for specialized application domains. Current solutions using multi- and many-core sequential processors are being challenged by general-purpose graphics processing units (GP-GPUs), field programmable gate arrays (FPGAs), coarse-grained reconfigurable arrays (CGRAs), and massively-parallel processor arrays (MPPAs). These specialized architectures are frequently integrated as co-processor accelerators for dedicated host processors, offloading an application's kernels rather than operating as standalone processors.

Spatial architectures offer a large number of concurrent functional units (*i.e* processing elements) that allow for a high degree of parallel computation by spreading out the computation across an entire chip. They differ from traditional sequential processors by communicating primarily through space via wires and distributed storage rather than through time

using centralized storage structures; their execution models favor concurrent operations over sequences of operations. In general spatial architectures are designed as arrays composed of repeated tiles that are simpler than traditional sequential processors, increasing computational capacity by minimizing complexity. The distributed nature of spatial architecture's computation, communication, and storage allow spatial processors to scale to larger sizes more efficiently than sequential processors. This scaling comes at the cost of more complex programming models, some of which will be discussed in this dissertation. The diversity of spatial architecture designs is considerable, but a unifying theme is that spatial computing favors a larger ratio of concurrency to control flow than sequential computing. They thrive on applications where there are many parallel operations for each basic block (or hyperblock) in the execution's trace; furthermore, they are especially good at repeating relatively short tasks and thus excel at executing an application's computationally-intensive inner loops.

The renewed interest in spatial computing results from its ability to utilize the vast number of transistors available on a single chip in a power-efficient manner, and traditional sequential processors' inability to effectively utilize the current abundance of silicon real-estate. Unlike sequential computer architectures (*i.e.* conventional processors), programmable spatial computing architectures provides exceptional performance and performance per watt within certain application domains [5, 6], approaching levels close to those offered by dedicated ASICs. This benefit is derived from the vast amounts of parallelism achievable by the large quantities of concurrent hardware and the relatively high ratio of functional units to control logic. For many spatial architectures, rather than than having an instruction fetch pipeline and branch predictor for 5-8 functional units, they amortize even simpler control structures (such as a modulo counter) across hundreds of functional units. Additionally, performance scaling is commensurate with the number of transistors utilized. Coarse-grained reconfigurable arrays (CGRAs) are an interesting sub-class of spatial computing, which have also seen renewed interest that stems from their superior energy efficiency over FPGAs for word-wide (datapath) applications and the ever increasing cost of developing new application-specific integrated circuits (ASICs).

Coarse-grained reconfigurable array architectures exist in the design spectrum some-

where between FPGAs and VLIW processors. They are composed of word-wide functional units (*i.e* ALUs and multipliers) and multi-bit buses. As a result, CGRAs are similar to an FPGA, but operate on words of data at a time; however, they still face many of the same CAD tool challenges (*i.e.* place and route) as FPGAs. Unlike commercial FPGAs, they are dynamically reconfigured with parts of the CGRA changing their functionality on a cycle-by-cycle basis. Many CGRAs are similar to a VLIW processor that has been extended to a 2-D array, with less instruction fetch / control logic; essentially they execute Very Short Very Wide programs. Furthermore, CGRAs have distributed (rather than centralized) storage and communication resources, and no global shared memory. These architectures are statically scheduled so that a compiler (or CAD tool) choreographs the execution of functional units and movement of data at compile time, which allows for fairly simple (and low overhead) runtime control logic. Therefore, they are do not directly handle dynamic, data-dependent control flow and thus are ill-equipped to execute general-purpose program branches. As a result, these architectures are a cross between being configured like an FPGA and programmed like a VLIW, and they are optimized for executing loop bodies.

CGRAs use dynamic reconfiguration (*i.e.* time-multiplexing) to increase the array's effective logic capacity by using small configuration memories to reconfigure and reuse expensive compute, communication, and storage resources. This technique is particularly effective when an application's data dependencies would have otherwise forced resources to be idle, a common case that results from a loop-carried dependence. In summary, time-multiplexing enables CGRAs to make efficient use of their compute resources by being able to 1) execute applications of varying size, and 2) utilize what would have otherwise been dead space in the architecture's static schedule.

Digital signal processing, streaming data, and scientific computing are domains that continue to demand improvements in energy efficiency as well as performance. These domains are interesting because the task- or thread-level parallelism (TLP), data-level parallelism (DLP), loop-level parallelism (LLP), and instruction-level parallelism (ILP) present in these applications is fairly predictable and not control dominated; this means that applications in these domains can often be efficiently accelerated with spatial processors, as illustrated in [2, 7, 8, 9, 10, 11]. My research, and the majority of CGRA research, focuses on accelerating

these application domains, although my research is focused on doing so in an energy-efficient manner. Common characteristics that make these applications efficient for spatial execution are described by Ylvisaker et al. [12] and presented here:

- There exists a small number of inner loops, or kernels, which dominate the runtime of the application.
- The behavior of the kernel is relatively regular and predictable.
- The ratio of computation to external communication is large enough to keep the computing resources busy.
- The active working set of each kernel is of a reasonable size that can fit within on-chip storage resources. Blocking and striping exemplify some staging techniques that are used to partition the full data set into reasonably sized working sets.

The demand for these applications and the focus on energy efficiency has also driven a resurgence in commercial research efforts focusing on spatial computing for these domains [13, 14, 15]. Loop-level parallelism is of particular interest to CGRAs as the computationally intensive inner loops (or kernels) of these applications frequently can be pipelined, thus implementing loop-level parallelism very efficiently as pipeline parallelism. The effectiveness of accelerating loop-level parallelism with pipeline parallelism and statically scheduled architectures is well established by the dominance of VLIW architectures in digital signal processors (DSPs). As the centralized control and storage structures of VLIW DSPs fail to scale effectively, CGRAs offer a scalable architecture that transfers the advantages of VLIW architectures to spatial computing. This in turn leads to the affinity between CGRAs and accelerating loop-level parallelism in the application domains of digital signal processing, streaming data, and scientific computing.

1.1 Advantages of specialized architectures

The advantages of spatial computing, over sequential computing, are the result of distributing computation, storage, and communication over the entire architecture. It is this

decentralization of resources that allows continued scaling of performance and energy efficiency with additional physical resources. In contrast to spatial architectures, sequential processors force all computation, storage, and communication through a relatively small number of centralized structures. Because the area and power of these centralized structures typically scales quadratically with the size of each structure (*e.g.* number of inputs), it is no longer profitable to build larger, monolithic sequential processors, thus leading to the renewed interest in architectures with distributed, and decentralized, structures.

Configurable computing is a subset of spatial computing where all of the resources are explicitly controlled by a single, or a small number, of large configurations. A traditional processor's instructions are small, densely encoded, words that dictate the configuration of a local part of the processor's pipeline at any given time: *e.g.* the opcode controls the functional units during the execution stage, and register fields index into the register file during instruction decode. The configurations used in configurable computing typically encode the global behavior of all resources simultaneously. Configurable architectures are either reconfigured during execution (*i.e.* dynamically) or configured once (*i.e* programmed) at load-time. FPGAs are a common example of a load-time configurable computing system, while CGRAs such as RaPiD [16] and PipeRench [17] are examples of dynamically reconfigurable computing systems. Therefore, processors and FPGAs represent two extremes for how to control a system: where processors execute a long sequence of very compact instructions, while FPGAs have a single, very large and explicit configuration to control its fabric; CGRAs fill out part of the spectrum in between these two ends, although closer to the FPGA side, with a small number of fairly large configurations that are sequenced over time.

1.1.1 Performance advantages of configurable computing

There have been several studies documenting the performance advantages of configurable computing, each of which show how these architectures achieve massive parallelism via many concurrent computing resources. One example is given by Ebeling et al. [18] for the RaPiD architecture. They show that for an OFDM application RaPiD has 6x the cost-performance

of a DSP, while an ASIC implementation is only another 7x better than RaPiD. Finally, they show that RaPiD has 15x the cost-performance of an FPGA implementation. They measure cost in terms of normalized area and observe that RaPiD, a CGRA, provides an interesting design point between the programmable DSP solution and an ASIC. Another example is provided by Singh et al. [19] who show the MorphoSys CGRA providing significant performance gains over both FPGAs and sequential processors. For an automated target recognition application MorphoSys was twice as fast and required 1/16th the hardware of an FPGA based solution. For a motion estimation algorithm from MPEG compression, MorphoSys achieved a 20x speedup over a high-performance DSP.

1.1.2 Energy efficiency for configurable computing

Despite providing significant advantages over sequential processors, significant improvements in the energy efficiency of configurable computing is still possible, by minimizing control logic, distributing resources, and not wasting energy moving instructions. In general, prior research on CGRAs has focused on improving performance and developing novel architectures (or architectural features), rather than energy efficiency. While commercial development of FPGAs focus on both logic density and techniques for energy efficiency, FPGAs still sacrifice energy efficiency for flexibility (when compared to CGRA spatial accelerators) by providing configuration bits for each wire and individual lookup-table. This is a reflection of FPGA's traditional focus on supporting general-purpose bit-wide computation instead of word-wide datapath computation. Recent developments in FPGA architectures have added monolithic, word-wide multiplier and block memory structures to the otherwise bit-oriented fabric. Despite these improvements, Kuon and Rose [20] determined that an application targeted to an FPGA typically suffers a ∼21-40x overhead in area, a ∼3-4x longer critical path, and a ∼12x reduction in energy efficiency versus an ASIC.

CGRAs typically aim to reduce the area and energy gaps between ASICs and configurable logic by an order of magnitude and halve the critical path penalty. They achieve this goal by using an architecture that is focused on datapath computations for DSP, streaming, and scientific computing. This means that the bulk of the computation and communication

resources will be designed to handle multi-bit data words. This transition from bit-wide to word-wide structures significantly reduces the overhead of the configuration logic. One of the challenges in this effort is that only including a word-wide datapath is inefficient for many sophisticated applications. Therefore, a bitwise control network may be selectively overlaid on the datapath to provide a good balance of control versus overhead.

Given the lack of comprehensive evaluations of the energy distribution within CGRAs, we turn to studies of other configurable architectures to identify interesting areas for research. Studies of the power consumption within an FPGA, as shown by Poon et al. [21], identify that 50% to 60% of the energy of an FPGA is dissipated in the routing fabric, 20% to 40% in the logic blocks, and anywhere from 5% to 40% in the clock network. Using these numbers as a guideline, we see that the interconnect offers the greatest chance for energy optimization, followed by the logic blocks.

1.2 Research overview

My thesis is that the energy efficiency and performance of traditionally designed CGRAs that use ad-hoc evaluation methodologies can be improved via the systematic evaluation and analysis of individual architectural features, and their subsequent composition. Previous research efforts have typically focused on improving performance and novel architectural features; those that have evaluated energy efficiency have done so in an ad-hoc manner that does not provide sufficient information to isolate the impact of specific architectural features. This dissertation focuses on optimizing architectural features for communication, storage, computation, and composition (topology) within a coarse-grained reconfigurable array. In addition to developing a CGRA architecture that is optimized for energy efficiency, by applying a systematic evaluation to each architectural feature, these results can be used to guide future architecture design. Since the focus of this dissertation is to identify what impact the architect can have on energy efficiency, this work leverages the state of the art transistor and physical design optimization techniques for energy efficiency.

The research goals for this work are to:

- Develop an energy-efficient CGRA that targets embedded and scientific applications.

- Explore the design space of a class of CGRAs by examining the interplay between performance, area, and energy for several architectural features, and isolate the contribution of each feature from the overall system.

1.2.1 Outline

In this dissertation I have focused on the optimization of architectural features for communication, storage, and computation. The experiments and results for each aspect of the architecture are presented in Chapters 6, 7, and 8, respectively. All of these individual results are then combined into the optimized Mosaic CGRA architecture, which is presented in Chapter 9. Chapters 1 through 5 provide both introductory and background material to support the experimental results. Finally, the attached appendix covers supplementary material about existing CGRA architectures. The following list is an outline for the dissertation.

Chapter 1: Introduction and motivation for the research as well as a brief overview of the entire dissertation.

Chapter 2: In-depth overview of a CGRA that defines key characteristics of CGRAs and how they differ from other spatial architectures.

Chapter 3: Defines the Mosaic CGRA architecture families. Describes the common architecture elements that are shared between experiments and what part of the CGRA architecture design space is covered by the Mosaic CGRA.

Chapter 4: Presents the Mosaic toolchain used for these experiments. Includes the CGRA architecture design tool, the SPR CAD tool, and the Macah language plus compiler used for writing applications.

Chapter 5: Describes the benchmark suite and general experimental methodology.

Chapter 6: Experiment on the structure of the top level interconnect.

Chapter 7: Experimental results of optimizing the distribution of storage resources to better manage short-lived and long-lived values.

Chapter 8: Experimental results for specializing the functional units.

Chapter 9: Presentation of an optimized Mosaic CGRA architecture that incorporates results from the previous three chapters. Additionally provides an evaluation that shows the net reduction in area-delay-energy from aggregating all optimizations over a baseline architecture.

Appendix A: Summary of key design principles that are common throughout the field of CGRA research.

1.2.2 Summary of results

This dissertation shows that architectural decisions can have a significant impact on energy efficiency as well as performance and area. Furthermore, using a systematic methodology for analyzing each architectural feature effectively isolates the impact that each feature has on the CGRA. Teasing out the individual contributions of each feature allows for the identification of architectural trade-offs when composing multiple features. In subsequent paragraphs we provide a high level summary of the optimizations examined in this dissertation, as well as detailed highlights of individual results. The optimizations focus on: communication, storage, computation, and composition.

When optimizing the communication resources, we show that time-multiplexing the top-level interconnect, so that it dynamically reconfigures along with the functional units, dramatically reduces the number of interconnect channels in the top-level interconnect and the amount of energy consumed in the interconnect. Furthermore, we illustrate that a single-bit interconnect nicely complements the word-wide interconnect by improving the global interconnect resource usage and reducing the complexity of the secondary, local interconnect between functional units.

Our work on optimizing the distributed storage structures in the CGRA identifies the unique challenges of managing values of different lifetimes in a time-multiplexed architec-

ture, as well as the advantages of dedicated storage structures for short-, medium-, and long-lived values. In particular, it highlights the benefit of using special rotating register files that were originally developed for VLIW architectures. These rotating register files provide dynamic register-renaming in a statically scheduled architecture that allow them to store long-lived values in an inexpensive manner. Additionally, we show the benefit of using private, short-term storage that is local to each functional unit.

Finally, we demonstrate that a heterogenous mix of computing resources provides a reasonable reduction in energy consumption by reducing the number of expensive, but rarely used, physical devices. This exploration also shows that composing multiple simple functional units into a single, compound functional unit is a valuable technique for reducing the amount of peripheral logic required. Furthermore, we show that inclusion of at least one compound functional unit that can perform all operations (*i.e.* a universal functional unit) reduces the chance for over-specialization and improves opportunities for spatial locality within individual computing resources.

Detailed highlights from each experiment are summarized here. Note that throughout these experiments we provide a baseline architecture as a point of comparison. For each experiment we tailor the baseline to reflect current state of the art design practices. Furthermore, the baseline architecture for each experiment incorporates results from previous chapters.

- Communication: Comparing statically configured interconnect versus scheduled, time-multiplexed, interconnect
  - A fully scheduled, 32-bit wide interconnect has an area-energy product that is 0.32× the area-energy product of a fully static interconnect.
  - A dedicated static, 1-bit wide, auxiliary interconnect complements the 32-bit datapath and reduces the implementation complexity of the compute cluster. Furthermore, it provides a 6% reduction in interconnect area-energy product over an architecture with just a 32-bit datapath.

- Storage: Managing short-lived versus long-lived values

  - Using three distinct storage structures that are optimized for short-, medium-, and long-term storage reduces the area-energy product to 0.61× the area-energy product of a baseline architecture.
  - Rotating register files provide a significant advantage, in terms of effective capacity and energy-efficiency for holding long-lived values, versus traditional register files.

- Computation: Specialization of functional units

  - Paring down hardware support of expensive and infrequent operations reduces the area-delay-energy (ADE) product of an architecture with two universal functional units to 0.86× a baseline architecture with four universal functional units.
  - A small number of universal functional units avoids over-specialization and improves spatial locality, particularly when executing applications with diverse operation mixes. Maintaining at least one universal functional unit per cluster, versus only having specialized units, reduced the area-delay-energy (ADE) to 0.89× the ADE of the more specialized architecture.
  - Fusing back-to-back multiply-add operations (without intermediate fanout) is a valuable optimization that takes advantage of a common design pattern to integrate an additional adder stage into the multiplier hardware, forming a MADD device. However, generalized opportunities for operation fusion are limited by diverse fusion patterns for a relatively low percentage of all operations.

- Designing an Optimized Mosaic CGRA:

  - Incorporating the best optimizations from each chapter produced an optimized Mosaic CGRA architecture that had an area-delay-energy product 0.10× a baseline Mosaic CGRA.

1.3 Synopsis

This dissertation focused on improving the energy efficiency of coarse-grained reconfigurable arrays, a type of spatial accelerator that is used for offloading an application's computationally intensive inner loops. CGRAs are a class of spatial architectures that are typically designed to exploit loop-level parallelism through loop pipelining, and have distinct similarities to a word-wide FPGA and a 2-D VLIW processor. The Mosaic architecture is a time-multiplexed and statically scheduled CGRA that leverages simpler control structures similar to VLIWs, by offloading the scheduling, placement, and routing of applications to the CAD tools. Minimizing the control and configuration structures reduces the overhead of executing an application's inner loops that are regular and predictable. Improvements in energy efficiency are achieved through systematic analysis that isolates the impact of individual architectural features on an architecture's area, delay, and energy, and the subsequent composition of multiple features. Finally, this dissertation presents an optimized Mosaic CGRA architecture that provides a substantial reduction in area-delay-energy product over a baseline CGRA.

Chapter 2

COARSE-GRAINED RECONFIGURABLE ARRAYS

This chapter describes CGRAs and how they differ from other spatial architectures. It defines four key properties, or trademarks, of a CGRA and identifies the different programming models and types of parallelism that are amenable to each type of spatial architecture. Appendix A provides an overview of several research and commercial CGRA and MPPA projects. Furthermore, Appendix A presents a supplemental survey of existing work in the design of CGRAs, that identifies key architectural features of individual research efforts.

Spatial computing offers high performance and power efficiency when compared to traditional sequential processors. This is a result of spatially distributing computation across concurrent resources and communicating primarily through wires rather than indirectly through memory. Two key advantages of spatial computing are that the number of available computing resources and the aggregate communication bandwidth scale efficiently with the amount of silicon resources used.

There are currently several classes of programmable spatial architectures that are available: field-programmable gate arrays (FPGAs), coarse-grained reconfigurable architectures (CGRAs), and massively parallel processors arrays (MPPAs). Sequential processors, even superscalar out-of-order processors, are not considered spatial architectures because the bulk of their communication is through the register-file, instruction issue queue, or memory hierarchy, rather than the functional units' forwarding logic. Neither are their multi-core derivatives, such as traditional chip multiprocessors (CMPs), multi-core, and many-core processors, which still rely heavily on centralized resources. Additionally, application specific integrated circuits (ASICs) are not considered here because they are fixed function logic, rather than programmable platforms. The boundaries between these classes is delicate at best and specific architectures can blur the boundaries. A brief overview of the architecture and computing models for FPGAs and MPPAs will be presented in Section 2.2

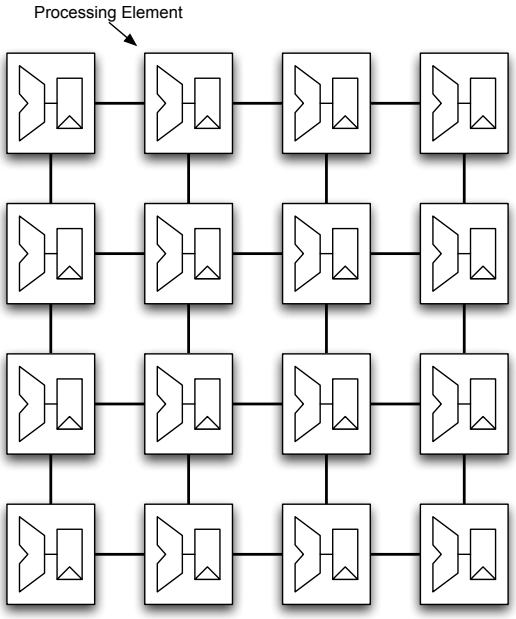
to help put CGRAs in context.

2.1 Coarse-Grained Reconfigurable Arrays

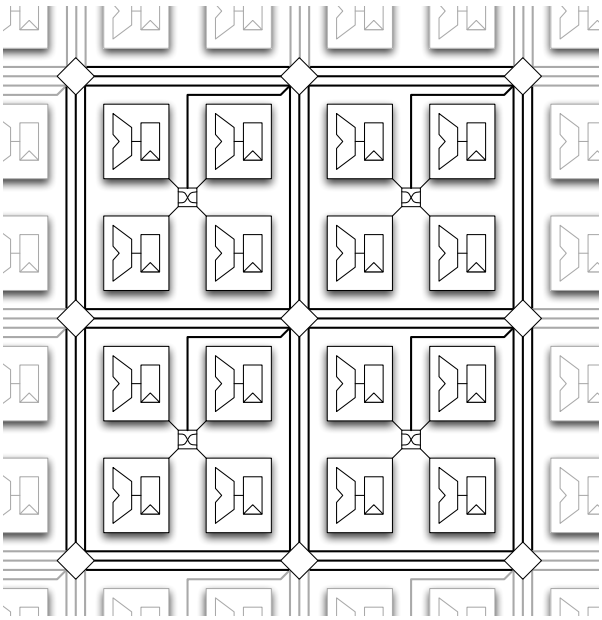
Coarse-grained reconfigurable arrays (or CGRAs) are an interesting class of spatial computing for efficiently exploiting loop-level parallelism within an application. As outlined in Chapter 1, a CGRA fits into the design space between an FPGA and a VLIW processor. They compute on words of data at a time, using multi-bit functional units like ALUs, multipliers, and shifters, and route values over word-wide buses. Many CGRA architectures are similar to a 2-D VLIW without the sequential fetch, decode, and issue instruction logic. Therefore, one way to envision many CGRAs is to imagine expanding the core functional units of a VLIW processor into a 2-D array, while stripping away most of the processor's instruction control logic (*e.g.* instruction fetch and branch prediction). Like a VLIW processor, the computation on these CGRAs is statically scheduled by a set of tools that choreographs both data movement and how the computation is distributed across the fabric. However, as with FPGAs, CGRAs use a suite of computer-automated design (CAD) tools that perform the scheduling and placement of operations and the routing of values, in addition to a front-end compiler.

Another perspective is that CGRAs are similar to a word-wide FPGA that reconfigures on a cycle-by-cycle basis. Like FPGAs, they are built from an array of fairly simple, repeated tiles. Unlike an FPGA, CGRAs are not designed for embedding any arbitrary circuit, but rather they are optimized for datapath computations that are regular and predictable. As a result, they are typically designed to operate at fixed clock frequencies and make heavy use of pipelining to achieve parallelism.

Overall, CGRAs contain a sea of functional units such as ALUs, shifters, and multipliers. These functional units are connected with wires, registers, and multiplexers to form topologies such as grids, meshes, trees, or linear arrays; techniques such as clustering provide multiple levels of communication resources. While running, a typical CGRA executes a single configuration that has a small number of contexts which are sequenced in a repeated pattern. Each context specifies the behavior of the functional units and the connection pattern of the interconnect for a single cycle of execution, and the configuration describes



(a) Fabric with mesh interconnect



(b) Fabric with clusters and island style interconnect

Figure 2.1: Example CGRA block diagrams.

the entire application kernel.

2.1.1 Anatomy of a CGRA

The behavior of a CGRA is governed by a configuration that represents how an application kernel was mapped onto the CGRA. Several components common to CGRAs are: processing elements, clusters of processing elements, embedded memory, interconnect, and configuration memory. There are also several common terms that describe key structures in CGRAs: context, datapath, configuration-path, and computing fabric. Throughout the literature, research projects use inconsistent terminology for most of these components and terms, but the common usage and definitions can be consolidated to the list presented below.

As a rule, the core of a CGRA is composed of a set of simple, repeated tiles or building blocks, illustrated in Figure 2.1. Furthermore, CGRAs typically have a fixed clock frequency and perform a fixed amount of work per clock cycle. By establishing critical paths and allowing complex structures to be optimized to minimize delay, a fixed frequency design allows the architecture to run at much higher clock rates than architectures such as FPGAs that are designed to run at variable frequencies, in general this improves performance as shown in [22].

The configuration for a CGRA specifies its behavior on a per clock cycle basis. Execution of an application kernel consists of sequencing through multiple contexts in the configuration. In general, data is fed and retrieved from the fabric by some external I/O or memory controllers, often in a streaming fashion. Common sources of I/O include a host sequential processor, some stream engines, or direct memory access (DMA) controllers. The design challenges of these I/O controllers is not unique to CGRAs, but is common to most spatial accelerators.

The definitions for the common components and terms are presented in the following list:

- **Processing elements** (PEs) are the basic tiles (or building blocks) within a CGRA. They typically contain a functional unit and peripheral logic, such as registers and / or distributed RAM, communication resources, and some configuration logic. Functional

units range from simple devices like arithmetic and logic units (ALUs), shifters, or look-up tables (LUTs), to complex devices such as multipliers, multiply-add units, or even domains specific logic such as a Viterbi add-compare-select unit or a FFT butterfly unit. The processing element is the basic computing resource for the CGRA. An abstract PE is shown in Figure 2.1(a) with an ALU and a register.

- **Clusters** of processing elements are hierarchical groupings of PEs, and are sometimes used to delineate between multiple levels of communication resources. Figure 2.1(b) shows a CGRA with four clusters, each with four PEs connect via a crossbar.

- **Embedded memory** provides storage for the working set of an application kernel and collectively refers to all of the on-chip storage. Logically, it is traditionally distributed between embedded RAM blocks, register files, and distributed registers. Physically, the storage elements are spread throughout the interconnect, clusters, and processing elements. Furthermore it is explicitly managed and not part of some cache hierarchy. Management techniques differ based on the type and usage of each structure. Large embedded block memories are frequently managed directly by the application kernel, while registers and register files are managed by the CAD tools.

- **Interconnect** is formed from a collection of wires, multiplexers, and registers, and is the primary spatial communication resource. Interconnects are either statically or dynamically configured, and the communication patterns within the interconnect is determined by the CGRAs configuration. Standard types of interconnects are the mesh, horizontal or vertical bus (HBUS or VBUS), grid (or "island" style), tree, and crossbar. Note that a mesh is sometimes referred to as a nearest neighbor interconnect. A key distinction between an interconnect and a network on a chip is that a network on a chip requires a routing protocol and routes communication traffic on a packet or circuit basis, while an interconnect is configured for a given communication pattern. Therefore, an interconnect has less dynamic flexibility but less overhead as well. Additionally, it is easier to construct a deterministic interconnect than a

deterministic network-on-chip, and deterministic communication makes it easier to statically schedule computation and avoid both deadlock and livelock.

- **Configuration memory** stores the configuration for the CGRA. Configuration memory can be globally centralized, distributed locally to each processing element, or some combination of the two. The capacity of the configuration memory is typically designed to hold tens to hundreds of contexts, and thus only requires a few hundreds of thousands of bits.

- **Context** or configuration plane is the set of configuration data needed to configure all components of the CGRA for a single cycle. A single context typically has several tens of thousands of bits, with one or a few bits per device / structure. The configuration for an application kernel is composed of one or more contexts. Typically, the capacity of the configuration memory is described by the number of contexts stored, rather than the total number of bits.

- **Datapath** is the primary set of resources intended to execute the core computation of an application's kernel. Most of the processing elements, memory, and interconnect are within the datapath. These resources are word-wide, reflecting the coarse-grained nature of a CGRA.

- **Control-path** is an optional and supplementary set of resources intended to execute single-bit, control-oriented computations in an application's kernel. Typical uses are for managing predicated execution and conditional data movement. Like the datapath, it contains processing elements and interconnect resources, although the processing elements use either lookup-tables (LUTs) or Boolean gates.

- **Configuration-path** is the auxiliary set of resources intended to choreograph or sequence the execution of the datapath. In general, it does not directly execute any part of an application kernel, rather it manages the reconfiguration of the compute fabric. For statically scheduled systems, it is responsible for the distribution (and decoding)

of the configuration data. In dynamically scheduled systems, it can feed signals from the datapath or control-path into configuration logic to incorporate runtime data into configuration decisions.

- **Computing fabric** or just **fabric** refers to the core set of computing and communication resources of the CGRA, *i.e.* the datapath, control-path, and configuration-path. It contains the processing elements, the interconnect, and can also include additional embedded memory. Abstractly, the fabric contains a sea of interconnected processing elements. However, the physical arrangement and topology of the interconnect is an important element of design that dictates communication patterns and latencies.Figure 2.1 provides examples of a 2-D mesh (Fig. 2.1(a)) and island style (Fig. 2.1(b)) array.

2.1.2 Trademarks of CGRAs

Coarse-grained reconfigurable arrays (CGRAs) can be characterized by a common set of properties or trademarks. These properties define an archetypal CGRA and can serve to differentiate them from other spatial computing architectures (discussed in Section 2.2). Some of these properties represent a hard boundary, while others are more of a spectrum of "CGRA-ness". The goal of these rules is to delineate the space of spatial computing and to create a common nomenclature for architecture discussion and design. The properties are enumerated below, followed by a more detailed description.

1. Coarse-grained datapath
2. Spatial communication via wires
3. Small number of independent domains of control
4. Reconfigurability

Property 1: Coarse-grained Datapath

The key distinction of a coarse-grained (versus fine-grained) datapath is that the computation and communication resources have a "native" word-width greater than 1 bit. Common

ranges for the word-width is between 4 and 64 bits[1], and is dependent on the targeted application domain(s). Coarsening the datapath allows for optimizations that substantially reduce the overhead of programmable logic and routing, improving the performance of each resource. The disadvantage is that the datapath will be underutilized if the native word-width of the application does not match the native word-width of the datapath. For this reason, a number of architectures use scalable datapaths (Appendix A.1.1).

Reductions in the overhead from the programmable logic come from a reduction in the number of configuration bits needed to specify the behavior of the CGRA fabric, and a reduction in the overall flexibility of the fabric. For example, in a word-wide datapath a single configuration bit controls the routing of multiple wires rather than a single one. Similarly, less flexible functional unit structures, like ALUs and multipliers are commonly used rather than word-wide lookup table based functional units, which require lots of configuration memory. However, the coarse-grained functional units such as ALUs and multipliers that typically replace single-bit lookup tables do not support the flexibility of embedding arbitrary circuits onto the datapath. Another benefit of using fewer bits to configure the fabric is that less time is taken to load a configuration into the CGRA.

Property 2: Spatial Communication

Communication within the datapath is primarily spatial, via wires, rather than temporal, via named registers or memory locations. Spatial communication routes data along a path from a producer to a consumer and is frequently pipelined. The alternative method is temporal communication, a building block of sequential processors, where a producer places a data into a named memory location so that the consumer can retrieve it at some future point in time. A central challenge of temporal communication is that the memory hierarchy creates a bottleneck as all communication traffic is funneled through it. This highlights the main advantage of spatial communication: communication is distributed, thus providing better performance (particularly increased aggregate bandwidth) as communication resources are

[1]One example that illustrates how these properties represent a spectrum of "CGRA-ness" is the GARP architecture. GARP uses a 2-bit datapath, which technically is a multi-bit word, but it does not provide many of the benefits of a coarse-grained datapath, and so we classify it as fine-grained.

added. The downside of spatial communication is that it is much harder to choreograph and efficiently use the resources. CGRAs primarily use spatial communication due to its scalability advantages.

Property 3: Control Domains

A control domain represents an autonomous region of an architecture that can execute an independent computation. It is similar to a thread of control as used in operating systems and programming languages, with the added meaning that it refers to a specific portion of a CGRA fabric. For traditional sequential processors the program counter (PC), associated control logic, and machine registers represent a thread; therefore, each processor represents a single control domain. Operating systems are able to virtualize these resources, allowing sequential processors to be viewed as having a virtually unbounded number of control domains.

Control domains are harder to virtualize on CGRAs, because the size of the local state is much bigger. To date, there is no known way to effectively virtualize the control domains on a CGRA as efficiently as a sequential processor and thus CGRAs are unable to present the appearance of having a virtually unbounded number of control domains. The most promising efforts to-date are multi-context FPGAs [23] and the Tartan project [24], which included the PipeRench CGRA [17]. PipeRench had a single control domain, but provided a virtualized hardware architecture that supported on-line reloading of the datapath and fairly compact contexts. While the virtualized hardware architecture (VHA) did nothing directly to emulate multiple control domains, it did set the stage for subsequent work on the Tartan project [24]. Leveraging the capabilities of on-line reloading and the programming model afforded by the VHA, the Tartan project developed "hardware paging", in which multiple, independent configurations could be swapped onto an array of PipeRench CGRA fabrics. In theory this combination could be used to virtualize the number of contexts supported beyond what would fit in on-chip memory, but such an approach would be unlikely to emulate an unbounded number of control domains due to the size of each configuration and the limitations of off-chip memory bandwidth. Multi-context FPGAs provide another

opportunity to emulate a large number of control domains because single-context FPGAs are already able to construct logic to emulate a reasonable number of control domains from the existing control- and datapath. Multi-context support would substantially increase the number of control domains supported, but is again limited by off-chip bandwidth and the inability to efficiently reload the offline contexts.

Within a CGRA it is common to have one control domain, with a single controller, that sequences through the contexts of a global configuration and choreographs the operation of all processing elements and interconnect resources. Two commonly employed mechanisms are (1) broadcasting a single context, a single instruction multiple data (SIMD) model, or (2) broadcasting a context pointer. Using a context pointer allows each processing element to index into a local configuration array, thus executing a unique context word, which looks like a distributed very long instruction word (VLIW). In a few architectures there are a small number of parallel controllers that control either independent (MATRIX [25] and Systolic Ring [26]) or overlapping (RaPiD [16]) regions of the fabric; examples of these are described in Appendix A.1.3.

Property 3.1: Impact on Programming Model

The most important consequence of the number of control domains is the impact it has on the programming model. Traditional parallel programming techniques that are used in scientific computing on sequential processors rely upon the abstraction that each processor can support a virtually unlimited number of threads of execution. The limit of one or a few control domains per CGRA changes how the programmer can exploit parallelism on these systems. Specifically, loop-level parallelism (LLP), data-level parallelism (DLP), and instruction-level parallelism (ILP) can all be exploited, but without some sort of programatic transformation, thread-level parallelism (TLP) cannot. Specifically, CGRAs have no effective method for supporting the parallelism of a large number of independent threads, and so they would have to be re-coded to utilize a different type of parallelism. The C-slowing technique would be one mechanism for collapsing some types of thread-parallel programs into data-parallel programs, which are more efficient description for CGRAs.

One way to evaluate how well an application, or kernel of an application, will map to a CGRA is to examine the application's control-dataflow graph. The control-dataflow graph (CDFG) is a common way to represent an application inside of a compiler or CAD tool; it has operations and program statements as vertices and represents both control and data dependencies as edges. In this representation, groups of vertices that are only joined by dataflow edges are basic blocks within the program and can be analyzed as independent dataflow graphs (DFGs); control edges join the basic blocks to show the program's possible paths of execution. Given the limited control capability of CGRAs, they typically focus on executing basic blocks, or more complex regions of the graph such as super-blocks or hyperblocks. As a result, it is common for only a single (or small number) of dataflow graphs to be mapped to the CGRA at a given time. Applications with high levels of LLP, ILP, and DLP typically have large regions of the CDFG with very few control edges, a good indication of regions (*i.e.* kernels) with low control complexity and potentially many concurrent operations. Applications with high levels of TLP will have multiple regions of the CDFG that are not connected (or only sparsely connected) via control edges, and thus can be factored into many threads. Therefore, each region could execute as a concurrent thread of control; a property that CGRAs with one or only a few control domains are unable to effectively exploit.

One reason that CGRAs have focused on supporting a small number of control domains is that applications in the domains of interest are typically dominated by only a couple of computationally intensive loops. Mapping a dataflow graph to a CGRA is simplest when the DFG is a directed acyclic graph (DAG); however, in practice many interesting DFGs are not DAGs due to the existence of feedback. Nadeldiner [27] presents some techniques for solving this problem through the use of pseudo operators, which make the DFG virtually acyclic, but mentions that embedding loops of variable length within a DAG can prevent pipelining. For this reason, many CGRAs choose to focus on executing a single loop body, rather than a loop embedded in a larger DFG. By moving the scope of execution to the loop body, it is often possible to pipeline dynamically terminating loops. As previously mentioned, pipelining a loop to transform loop-level parallelism into pipeline parallelism is a well studied problem [28, 29] that requires minimal control overhead and only a single

control domain, making it a very efficient method of execution.

CGRAs with multiple control domains have the ability to execute multiple DFGs simultaneously, but can also be used to support multiple levels of loop nesting or sequencing within a single DFG, as illustrated by the RaPiD [30, 31] architecture. Another possibility for mapping nested and sequenced loops onto a single control domain of a CGRA is for a compiler to perform loop flattening [32, 33, 34] and loop fusion [35] on the kernel. Loop flattening is a software transformation that takes a set of nested loops and turns it into a monolithic loop using conditional statements to guard the contents of the original loop bodies. Loop fusion transforms sequenced loops into a monolithic loop by matching the number of iterations of all loops and then fusing their bodies.

Another challenge for programming CGRAs is that they typically only support a small number of contexts for each configuration and thus can only execute short "programs". However, despite the limited number of contexts, the number of concurrent operations per context means that the configuration for a CGRA can contain hundreds to thousands of operations. In practice these restrictions do not limit the size or number of application kernels that can be accelerated; rather loop-carried dependencies, resource contention, or other aspects of the loop body limit the amount of parallelism that can be extracted.

Property 4: Reconfigurability

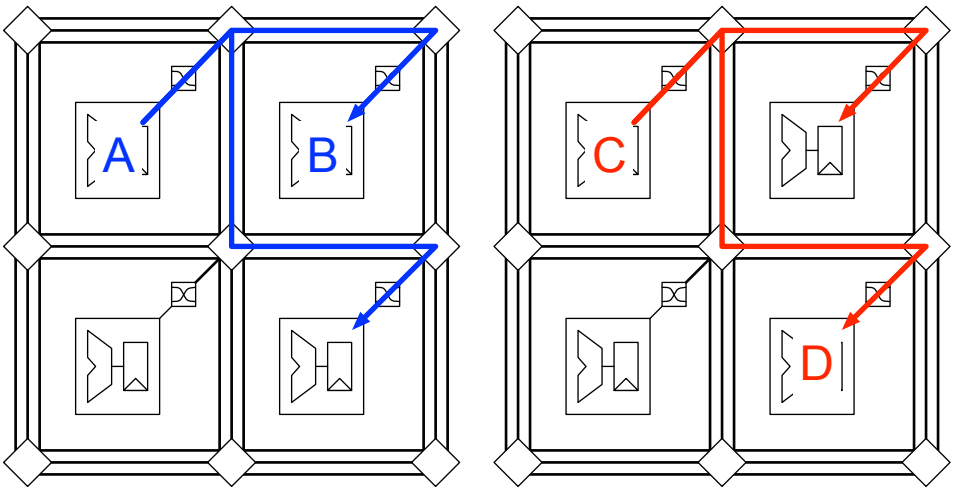
The ability to reconfigure the function of the fabric is what sets CGRAs apart from application specific integrated circuits (ASICs), making CGRAs multi-function devices. The components of a CGRA are partitioned into sets of statically configured and dynamically configured elements. Statically configured components are configured once per device reset, or until a new configuration is loaded. Dynamically configured components are reconfigured during execution as a configuration plane (context) is loaded from configuration memory into the fabric, *i.e.* it is made active. Dynamic configuration typically occurs on each cycle of execution and is also known in the literature as: runtime reconfiguration, dynamic reconfiguration, or just reconfigurable hardware. RaPiD [30] is an example architecture that contains both statically and dynamically configured components, but most CGRAs

are entirely dynamically configured. Dynamically configured hardware is typically statically scheduled, with all computation and data movement orchestrated at compile time by the architecture's CAD tools. The alternative, as exemplified by MATRIX [25] and Element CXI [15], is to dynamically schedule the reconfigurable hardware, which incorporates runtime data into the execution and configuration of the CGRA. The advantages and disadvantages of statically scheduling versus dynamically scheduling the dynamically configured hardware is discussed in Section 2.3.1. The precise mechanisms for dynamic configuration can be broadly broken down into three methods: context-switched, time-multiplexed, or pipeline reconfiguration. An overview of each mechanism follows. Context switched is by far the most common mechanism for dynamic configuration in CGRAs.

- Context-switched reconfiguration [36] has an active context and a set of inactive contexts. On each cycle of execution the domain controller selects a new context to be active, and thus changing the configuration of the entire fabric. As described in Section 2.1.2, the controller can broadcast either the entire context, or a context pointer that is used to index into local configuration memory.

- Time-multiplexed reconfiguration dynamically configures a set of the components that interact with another set of statically configured components. The distinction between context-switched and time-multiplexed relates to the interplay between dynamically and statically configured components. Context-switched reconfiguration changes the entire CGRA fabric and thus changes the computing "context" of each component. Time-multiplexing, on the other hand, only changes a portion of the components in the CGRA fabric, each of which is essentially multiplexing between multiple sets of statically configured components. Time multiplexed reconfiguration was used in the RaPiD [30] architecture.

  Figure 2.2 illustrates how the functional units in three clusters are reconfigured around a static interconnect configuration. The communication pattern for both phases is initially configured. Then $A \rightarrow B$ communicates in phase 0, followed by $C \rightarrow D$ in phase 1.



(a) Phase 0: Operations $A \rightarrow B$ (b) Phase 1: Operations $C \rightarrow D$

Figure 2.2: Example of time-multiplexing functional units around static interconnect. The interconnect pattern is configured once, and functional units reconfigure from $A \rightarrow B$ to $C \rightarrow D$.

- Pipeline reconfiguration was developed in the PipeRench [37] architecture. The architecture is composed of a linear array of physical pipeline stages, and the application is represented as a linear array of virtual pipeline stages. At each stage of execution a virtual pipeline stage is mapped onto a physical pipeline stage. When all of the physical resources are configured, the oldest configured stage is reconfigured with the next virtual pipeline stage. Essentially it is a dynamic, cycle-by-cycle version of an FPGA's partial runtime reconfiguration. Due to its limitation to feed-forward pipelinable dataflow graphs, this approach has not been adopted beyond its initial work. Pipeline reconfiguration was primarily used to implement pipeline virtualization [38, 17], which is discussed further in Section A.1.4.

Another consideration is the ability of a CGRA to reload the configuration memory, either loading in more of a single configuration or loading a new configuration. Support for reloading configuration is either online or offline with respect to the array's execution. Online reloading can overlap computation with reconfiguration, while offline reloading forces

the CGRA fabric to halt execution. MorphoSys [19], Pleiades [39], and Colt [40] use online configuration reloading to allow a host processor to rapidly switch between application kernels or portions of a kernel.

2.1.3 Summary of CGRA Trademarks

The four properties presented—coarse-grained, spatial communication, small number of domains of control, and reconfigurability—characterize "CGRA-ness" and can be used to understand what makes them efficient for spatial computing. In the following section other spatial computing architectures will be presented. These properties will then be used to understand the advantages and disadvantages of these architectures.

2.2 Alternate Methods of Spatial Computing

FPGAs consist of a sea of fine-grained programmable logic units, typically look-up tables (LUTs), with each LUT surrounded by channels of interconnect resources. This structure is referred to as an island style FPGA. Each LUT and interconnect channel is designed to produce and transport a single bit of data, respectively. Additionally, FPGAs are statically configured (*i.e* programmable) as opposed to being dynamically configured (*i.e* reconfigurable). They are typically programmed once in the lifetime of an application, due to overhead to reload the configuration data. One key difference between FPGAs and CGRAs relates to the concept of control domains. Control domains occur in architectures like CGRAs and sequential processors because there is dedicated logic that manages the execution of the architecture's resources. In contrast, FPGAs provide programmers with access to the raw bits that control the architecture's programmable fabric, and thus there is no pre-defined control mechanism that governs an FPGA's execution. Furthermore, as a product of being configured only once, FPGAs do not have control domains since the configuration-path is nominally static.

However, unlike most CGRAs they have the ability to execute arbitrary circuits and thus can emulate a variable number of control domains, a situation that commonly arises when using multiple soft-core sequential processors to emulate a chip multiprocessor or system-on-chip. Other possibilities for emulating multiple control domains arise from combining

architectural features like partial reconfiguration with a unified datapath and control path, which allows dynamic interaction with the otherwise static configuration-path. This combination allows the FPGA to reconfigure parts of the array on the fly and allows programmers to construct control domains as necessary. For this reason they can be thought of as having many control domains, based on the application that is currently implemented, despite the fact that they lack any native control domains.

Modern FPGAs are pushing towards the space of CGRAs by incorporating several coarse-grained components such as 18x18 bit multipliers and large embedded RAMs as standard features. Additionally, some FPGAs have incorporate something similar to dynamic reconfiguration [41], in the form of Run-Time Reconfiguration (RTR) or partial reconfiguration, where small parts of the FPGA can be changed over the course of many clock cycles.

MPPAs are composed of a large number of independent sequential processors, each of which typically uses scalar, in-order issue processor pipelines that are fairly simple compared to modern CMPs. Additionally, MPPAs tend to lack sophisticated control structures, such as branch predictors or caches for the memory hierarchy, reflecting their intended use for accelerating regular and predictable datapath computations rather than executing control dominated applications. Furthermore, they often eschew a single unified memory space and instead offer partitioned memory spaces and software-managed scratchpad memories. Finally, their communication is primarily spatial and the topology differs from architecture to architecture, with examples ranging from the point-to-point FIFO Ambric channels [13] to the scalar operand network of RAW [42, 43]. Like CGRAs, MPPAs are coarse-grained spatial architectures that reconfigure their functional units on a cycle-by-cycle basis, but they do so using small processor cores, rather than the CGRA-style dynamic reconfiguration. The primary difference between MPPAs and CGRAs is in the number of control domains supported and the resulting effect on the programming model. CGRAs favor one or a small number of control domains and focus on accelerating sequenced and nested loops. MPPAs typically have many, if not hundreds, of control domains (1 per processor) that target tightly-coupled, highly multi-threaded applications.

2.2.1 Computing Models

In addition to the differences between each class of spatial architecture, there are important distinctions in what type of computing/execution models they support. The computing model influences what types of parallelism and application kernels are executed most efficiently, and the types of programming models that can be used to program them.

FPGAs support a wide range of execution models due to their ability to implement arbitrary circuits and their flexibility in implementing multiple control domains. Traditionally, FPGAs were intended to implement circuits, specified in hardware description languages (HDLs), for random "glue" logic, system controllers, or prototyping an ASIC. More recently they have been used to implement datapaths for DSP and scientific kernels, or multi-threaded control applications. The fine-grained computing elements and rich interconnect lend themselves equally well to all types of parallelism and application domains. A new vein of research has been to create CMP or MPPA systems on an FPGA using multiple soft processors. Despite all of this flexibility, the difference between the fine-grained fabric of the FPGA and the word-wide datapaths of many application kernels causes a substantial overhead in terms of area and energy efficiency. For this reason, FPGAs are relatively efficient for bit-level computations.

CGRAs are frequently tailored to exploit the loop level and pipeline parallelism available in many interesting application kernels. They are generally intended to execute one dataflow graph at a time very efficiently, where each dataflow graph is a loop body. Accelerating multiple kernels from a single application is possible by sequencing through and executing kernels independently, due to the rapid reconfiguration and the small configuration size of many CGRAs. Features such as online reloading of configurations further improves this ability.

MPPAs frequently focus on exploiting thread (or task) parallelism. Unlike traditional multi-core, parallel programming systems (*e.g.* chip multi-processors) that work best with large, independent, and coarse-grained threads, MPPAs are designed to efficiently execute applications with small, tightly knit, fine-grained threads. This focus arises from the relative cost of inter-thread communication, which is fairly inexpensive for MPPAs. Application

kernels that are primarily composed of loop level and pipeline parallelism can be accelerated efficiently by either mapping loop iterations or pipeline stages to independent threads, and thus to processors in the array.

2.3 Advantages of CGRAs

Exploiting loop-level parallelism requires running multiple loop iterations simultaneously and can be achieved using two techniques: pipelining and multi-threading. Pipelining a loop creates a single instance of the loop body that is shared between multiple iterations of the loop, with pipeline registers breaking it up into multiple stages. These registers allow multiple pieces of data to execute simultaneously on a single instance of the loop, one per pipeline stage. Multi-threading a loop creates multiple instances of the loop body, each of which processes a piece of data. In many cases both pipelining and multi-threading can be applied to exploit LLP.

For kernels with pipelinable loops, a single instance of the loop body concurrently executes multiple iterations of the loop, reusing the same hardware within a single control domain. Therefore, communication between two loop iterations is frequently confined to a single region of hardware that can be statically scheduled, thus choreographing all data movement and minimizing runtime overhead. Furthermore, communication can be optimized such that a device will communicate with itself at later points in time, thus keeping communication local to the device. By transforming inter-iteration communication into intra-pipeline communication values typically travel shorter distances in the spatial architecture, and is thus likely to be less expensive than communication between two instances of a loop. Pipelining a loop also allows the loop body to share loop-invariant data between multiple iterations (*e.g.* the tap coefficients of a FIR filter), thus reducing the number of copies of data in the architecture. Multi-threading a loop creates a unique instance of the loop body for each loop iteration, forcing inter-iteration communication to require external communication between loop bodies. Multi-threaded loop instances are then spread across a spatial architecture, which increases the cost of communication between loop iterations. Furthermore, any additional control domains required to implement multiple instances of the loop adds overhead to a multi-threading approach with respect to pipelining. There-

fore, if the application domain is composed of kernels that exhibit pipeline parallelism, then CGRAs are a compelling method for spatial computing because of their ability to implement application kernels very efficiently. On the other hand, if many of the kernels have LLP that cannot be easily pipelined but can be exploited using multi-threading techniques, then MPPAs are very compelling for their ability to implement fine-grained TLP efficiently.

2.3.1 Statically Scheduled Execution

One source of design complexity in modern microprocessors, and some spatial accelerators, is hardware support for dynamic execution of instructions. Design examples include superscalar, out-of-order (OOO), processors and massively parallel processor arrays (MPPAs) such as Ambric. Superscalar OOO processors use complex forwarding networks between their functional units and register files to accommodate dynamic instruction execution. The area and energy of these forwarding networks grow quadratically with the number of functional units and register ports. MPPAs, like Ambric, use flow control on the communication channels between each processor to synchronize their execution with each other. The resources required for dynamic scheduling/instruction execution consume a significant amount of an architecture's area and energy, but are able to exploit an application's parallelism using runtime information. For applications that are control dominated or have irregular control-flow the use of runtime information can substantially increase the number concurrent operations executed. However, for compute-intensive applications with regular control-flow, most of the application's parallelism can be identified at compile-time and exploited using statically computed schedules. Statically scheduling a computation is an alternative to dynamic execution and is commonly used in both CGRA and VLIW architectures.

In a system that is statically scheduled, the complier, CAD tool, or programmer are responsible for scheduling when all operations occur and ensuring that there are no resource conflicts. While this simplifies the hardware design, the system loses the ability to use runtime information to extract parallelism. However, applications in the domain of interest typically have loop structures that can be statically pipelined using software pipelining

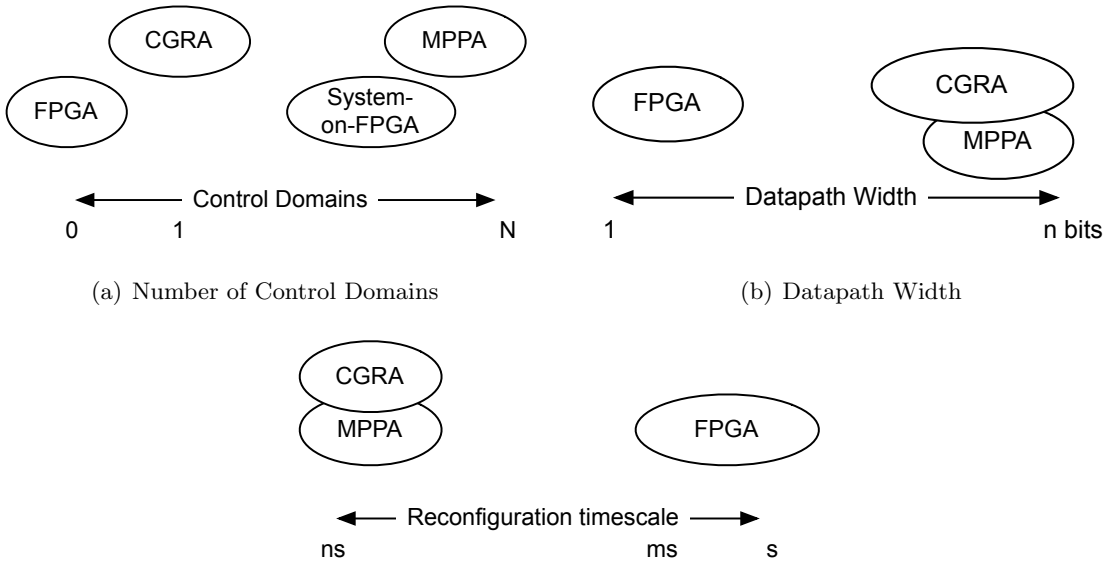
techniques, which allows for highly concurrent execution of the loop bodies. The structure of these applications make them well suited for achieving energy-efficient acceleration by using statically scheduled architectures.

It is interesting to note that while it is possible to design a CGRA that is dynamically scheduled and executed, this is not a common design choice, and it interacts poorly with the small number of control domain that are typical of CGRAs. The problem with combining dynamic scheduling and few control domains stems from the challenge of routing arbitrary runtime data from any part of the array to the domain controller in an efficient manner. Given a large array, where each domain controller covers a significant portion of the area, this would require a relatively complex, and expensive, combining interconnect that would need to be capable of arbitrary operations on the values input to the interconnect. Furthermore, the latency of this specialized interconnect would have to be designed pessimistically to allow values to be produced in any part of the array. The high latency required to incorporate the runtime data into the configuration logic and domain controller would create a long delay between when a result was computed and when it could affect the course of execution. Sequential processors refer to this phenomenon as branch delay slots, and modern microprocessors dedicate a significant amount of logic resources to mitigating this penalty. Other spatial architectures that have been discussed overcome these challenges by having a moderate to large number of control domains. In practice this means that runtime data can only affect local domain controllers, which effectively limits how far a value has to travel before being incorporated into the configuration logic, thus minimizing the delay between when a value is produced and when it affects the sequence of execution.

2.3.2 Programming CGRAs

CGRAs have the potential to be easier to program than MPPAs and many-core processors because of their limited number of control domains, although the relative immaturity of their programming languages and environments (especially for debugging) mean that this feature has yet to be effectively exploited. The intrinsic advantage CGRAs have is that in parallel programming, coordinating multiple threads of execution is hard; expressing an

application kernel as a single DFG should be easier than expressing it as multiple DFGs and coordinating between them. In the past, however, this has not been the case, because writing programs for CGRAs has typically been limited to the abstraction level of assembly languages or hardware description languages (HDLs), which are low level and fundamentally concurrent, and thus make tasks like pipelining difficult. There have been a number of research approaches that have sought to raise the level of abstraction for FPGAs, such as NAPA C [44], Streams-C [45], Impulse C [46], and CGRAs, with RaPiD-C [31], and DIL [17]. As language techniques for CGRAs improve and they incorporate features from FPGA-oriented languages, the expectation is that programming CGRAs should be simpler than targeting an MPPA.



(a) Number of Control Domains

(b) Datapath Width

(c) Reconfiguration Timescale

Figure 2.3: Approximate relationships in spatial computing landscapes.

A high level summary of the differences between FPGAs, CGRAs, and MPPAs is provided by Figure 2.3. For some set of application domains, including digital signal processing, streaming data, and scientific computing, CGRAs and MPPAs provide superior performance and lower overhead than FPGAs. The bulk of computations within these application domains are word-wide and align better with the datapath width of CGRAs and MPPAs,

as shown in Figure 2.3(b). The correlation between an application's operation width and the datapath's operator width leads to an improvement in performance and efficiency as resources are not wasted. The ability to quickly reload the configuration (Figure 2.3(c)) of a CGRA could also provide a performance advantage if multiple application kernels are to be serially executed.

It is clear that both CGRAs and MPPAs offer advantages for certain application domains of interest. To differentiate between CGRAs and MPPAs it comes down to the correlation between the available parallelism within the application and the capabilities of the CGRA or MPPA. As argued earlier, CGRAs offer the potential of a simpler programming environment for application kernels with pipelinable loop-level parallelism. As shown in Figure 2.3(a), MPPAs offer more control domains, and are tailored to express fine-grained thread-level parallelism. MPPAs can express loop-level parallelism as pipeline parallelism but incur the overhead of multiple control domains when compared to a CGRA. In general, MPPAs offer a potentially richer programming environment and the ability to exploit fine-grained TLP, thus targeting a slightly different set of applications from the domains of interest.

2.4 Summarizing CGRAs & MPPAs

Having examined the design space of CGRAs in detail and conducted an overview of MPPAs, it is clear that they have many structural similarities but differ primarily in their modes of execution. The primary difference is in the number of control domains available, which influences the computing model, programming model, and the types of parallelism that they are intended to exploit efficiently. Given that CGRAs are primarily focused on accelerating loop-level parallelism through pipelining, while MPPAs use fine-grained multi-threading to accelerate loops or task-parallel applications, CGRAs and MPPAs are only directly comparable when accelerating loop bodies. With proper tool support, CGRAs offer substantial advantages over MPPAs for accelerating loops. One advantage is in ease of programming: tools for automatically pipelining loop bodies are more robust and mature than those for partitioning loop iterations into multiple threads. The challenge of partitioning applications into fine-grained thread-level parallelism is similar to some of the place and route CAD problems faced by CGRAs, which is compounded with memory dependence analysis.

Ambric attempts to address this issue with the CSP-style programming model. A second advantage of CGRAs is that the additional control domains of MPPAs incur overhead that is unnecessary for most execution of pipelined loop bodies.

That being said, MPPAs offer a programming and execution model that is more flexible than CGRAs, making it capable of accelerating a wider range of applications. It seems clear that CGRAs and MPPAs are efficient tools for exploiting pipeline parallelism and fine-grained thread-level parallelism, respectively. What is unclear is which is the best tool for accelerating application kernels that lie somewhere in-between these two end-points and contain a even mix of pipeline parallelism and fine-grained TLP.

Chapter 3

DEFINING THE MOSAIC CGRA ARCHITECTURE FAMILIES

This chapter describes the class of CGRA architectures represented by the Mosaic CGRA family. Specifically, core elements of the Mosaic architecture are outlined, as well as the architecture's execution model. This dissertation explores the space of the Mosaic architectures to understand how architectural features affect energy-efficiency and performance, as well as to find a near-optimal solution. Subsequent chapters 6, 7, and 8 present an exploration of architectural features for global interconnect, storage, and functional unit composition, respectively. Chapter 9 presents an optimized version of the Mosaic architecture that incorporates many of the results from our architectural exploration.

3.1 The Mosaic Architecture

The Mosaic architecture is a coarse-grained reconfigurable array (CGRA). It is an application-accelerator and is coupled with a control (*i.e.* host) processor that will both initiate execution of the Mosaic array and stage data that feeds into and out of the accelerator. It exploits loop-level parallelism by pipelining an application's compute intensive loops. An application is statically scheduled onto the architecture as detailed in Section 2.3.1. Application loops are pipelined via software pipelining techniques and executed using a modulo schedule, which is discussed in Section 3.3.

As with most modern accelerators, the Mosaic CGRA architecture can be logically divided into the computing fabric array, the configuration logic, and some supporting logic that interfaces the Input / Output (IO) of the chip to the fabric. The design and evaluation of the computing fabric is the focus of this research. The execution model for the Mosaic architecture, as well as the Macah language, assumes that there are stream engines (similar to direct memory access engines) that shuffle data between external memory and the CGRA fabric. The stream engines provide independent threads of control, dynamic scheduling, and

dynamic execution to offload code that does not map well to the statically scheduled CGRA computing fabric. Examples of such code are address calculations and data fetching routines that typically are of low computing intensity and can be irregular, especially when dealing with locking routines / semaphores or dynamic data sets. As described in Section 3.4, the external memory is shared with a host processor, allowing for efficient communication between host processor and accelerator. This research assumes that the stream engines have sufficient resources to execute the stream accessor functions of the applications in the benchmark suite. Similarly, the external control and sequencing logic is assumed to provide the necessary flexibility for managing each application's configuration and data staging. Data staging is required at kernel startup and shutdown to move initial values that were live-in to the kernel, and final values that were live-out of the kernel, into and out of the CGRA fabric, respectively.

Like other spatial architectures such as FPGAs and GP-GPUs, the computing fabric of CGRAs is built from a small set of repeated tiled; additionally, the computing elements can be hierarchically clustered to create multiple levels of interconnect. This dissertation focuses on a baseline coarse-grained reconfigurable array architecture that is a 2-D array of repeated tiles, shown in Figure 3.1. Each tile consists of a compute cluster and a switchbox that connects to a global grid interconnect. Each compute cluster is composed of 4 processing elements, each of which contains an arithmetic functional unit and possibly some local storage. The size of the clusters was limited to 4 processing elements to balance the advantages of grouping functional units together against the increasing size of the intra-cluster crossbar. The clustering of processing elements is valuable for both energy efficiency and scaling the number of PEs in a fabric. Clusters provide natural boundaries for throttling bandwidth between processing elements, and for establishing clock cycle boundaries in a fixed frequency architecture. Additionally, application kernels typically have natural clusterings, such as feedback loops, in the dataflow graph that have more local communication within the clusters than between clusters. When the application's clusters have natural one-to-one mappings to clusters of PEs there is little need for significant communication bandwidth between clusters. Using a cluster design with 4 time-multiplexed processing elements provides an ample opportunity for a natural mapping with the application's clusters,

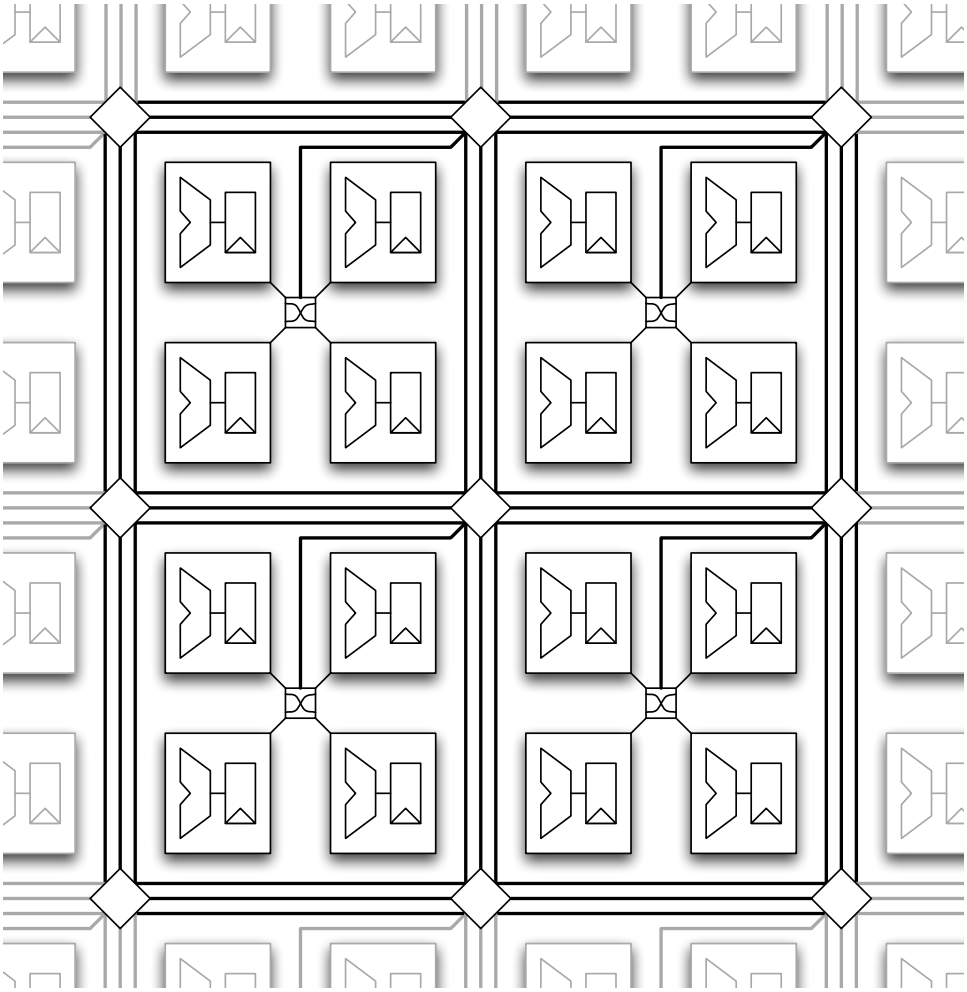


Figure 3.1: CGRA Block Diagram - Clusters of 4 PEs connected via a grid of switchboxes.

without over-provisioning the cluster's compute capacity. Given that the current design scales to hundreds of processing elements, it is unlikely that a single flat interconnect will be an optimal solution since the interconnect will be significantly over-provisioned for the application's clusters. Clustering PEs provides a natural way to add hierarchy to the interconnect, and enables each level of the interconnect to be optimized for different levels of latency, capacity, and energy efficiency.

Figure 3.2 highlights the logical connectivity of a cluster. A cluster contains four processing elements, each of which contains a functional unit, as well as some supporting logic

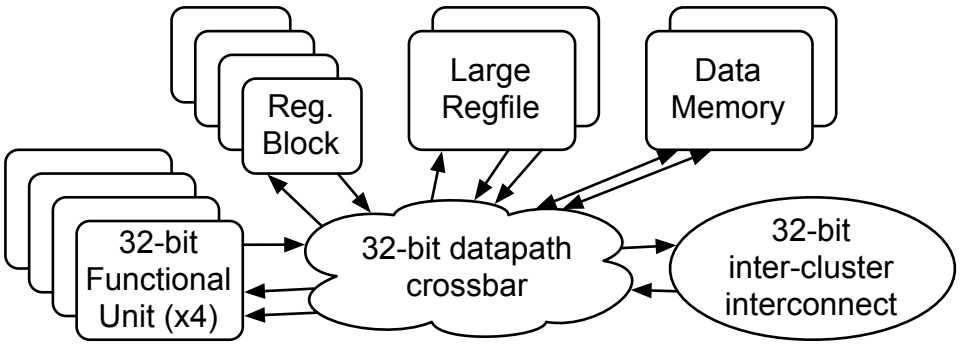


Figure 3.2: Compute Cluster Block Diagram - Contents of a cluster, including 4 PEs, connected via a crossbar.

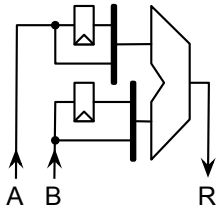
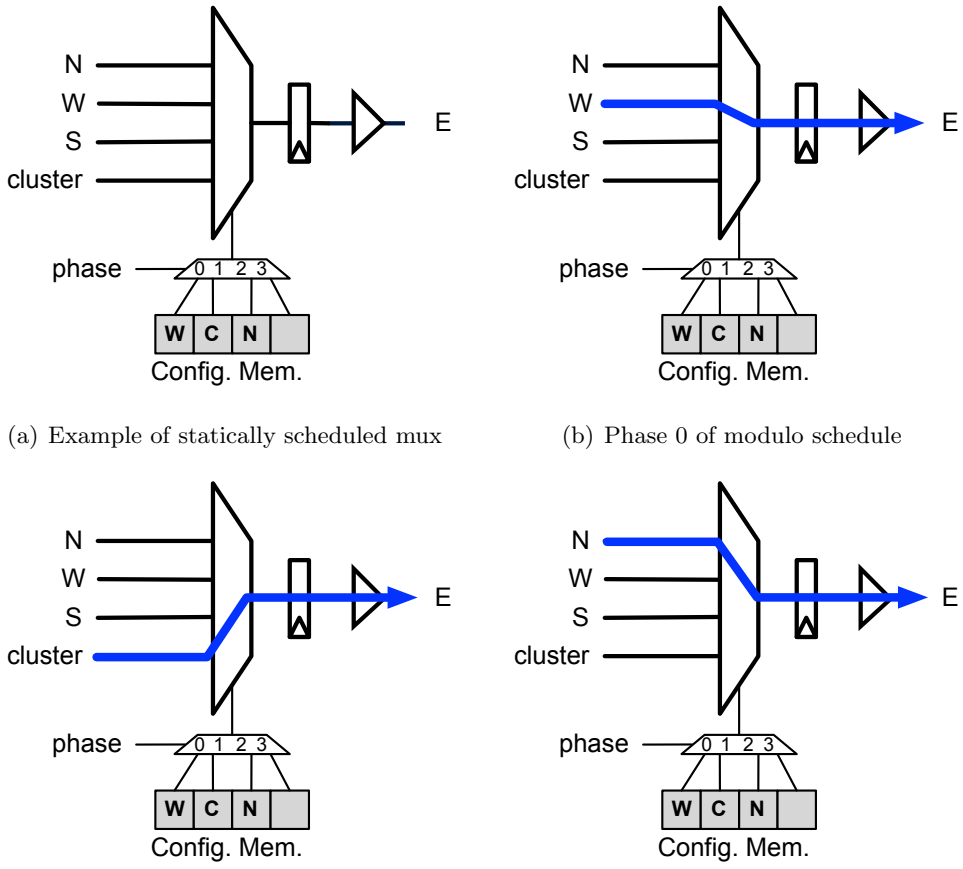


Figure 3.3: Example processing element with idealized functional unit and 1 register for input retiming.

and storage. Figure 3.3 shows an example processing element (including an idealized functional unit) from Chapter 7. More realistic and specialized functional units are explored in Chapter 8. The most notable feature in the processing element are the registers on the inputs of the functional units. Our functional units retime values on their inputs rather than their outputs, which was shown by Sharma et al. [47] to provide a 19% improvement in area-delay product. Other components in the cluster include registers, register files, and large data memories. Details about the cluster contents are presented in Chapters 7 and 8. Components that are not shown but are also attached to the crossbar are stream send and receive ports, which connect the computing fabric to the CGRA's external I/O via FIFOs that are managed by stream controllers.

3.2 Statically Scheduled Architectures



(a) Example of statically scheduled mux

(b) Phase 0 of modulo schedule

(c) Phase 1 of modulo schedule

(d) Phase 2 of modulo schedule

Figure 3.4: Example of a statically scheduled multiplexer. The 4 grey boxes are the configuration SRAM, supporting a maximum hardware II of 4. Text in each configuration box shows the connection pattern for each phase of an II=3 modulo schedule. Heavy blue lines indicate the connection pattern for each cycle of the schedule.

This dissertation focuses on CGRA architectures where the fabric is statically scheduled. Therefore, the majority of the core compute resources in the fabric are time-multiplexed and execute a static schedule. The architecture executes a cyclic schedule that is controlled by a modulo counter, which repeats itself after sequencing through the small number of

contexts in an application's configuration. Each resource has a configuration memory that contains N contexts, where N is the longest cyclic schedule supported by the architecture. By virtue of using a modulo schedule, as described in Section 3.3, N is also known as the maximum initiation interval (II) supported by the hardware. The initiation interval of a pipelined loop dictates how often the loop can accept new pieces of data, and is discussed in greater detail in Section 3.3. On each cycle of execution a global clock advances distributed modulo counters, which are used to index into the configuration memory (Section 2.1.2). Thus, the architecture is time-multiplexed in lockstep. Figure 3.4 shows an example for a statically scheduled multiplexer. In this example the application's II=3 and the hardware supports a maximum II of 4. The grey boxes represent the configuration memory, and the blue line indicates the connection pattern in each phase of execution.

3.3 Modulo Schedules

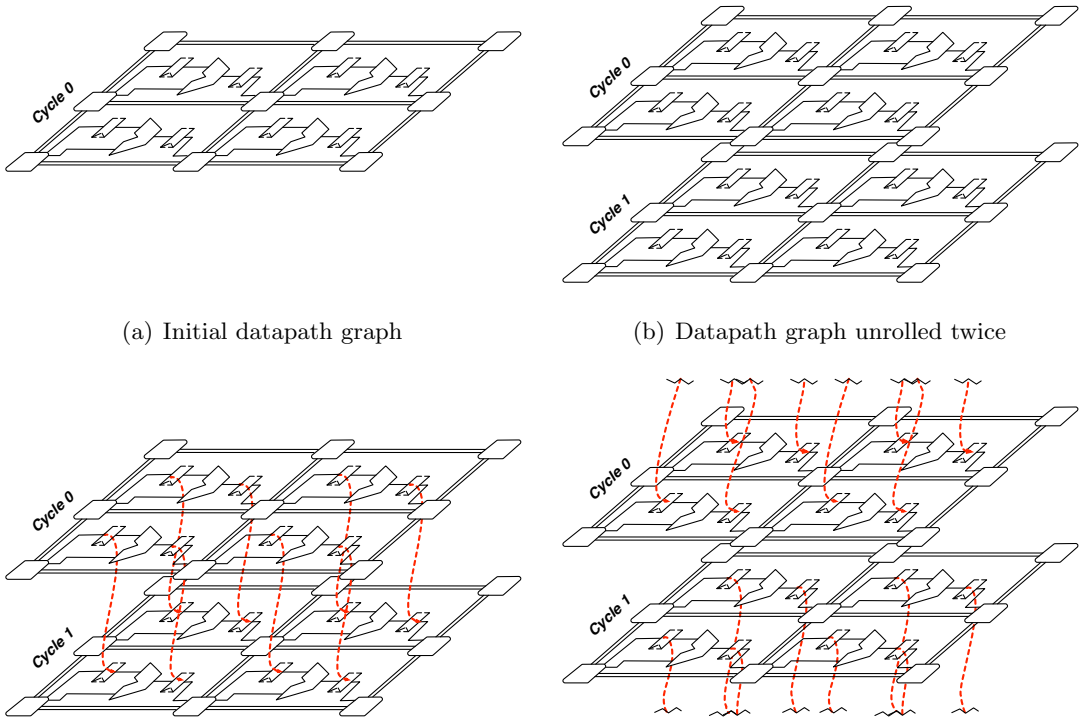
To generate the configuration for mapping an application to an architecture the Mosaic toolchain uses software pipelining techniques from the VLIW community [28], which generates a modulo schedule [29]. A modulo schedule describes a conflict-free allocation and scheduling of resources that permits overlapped execution of multiple instances of a loop body. As a modulo schedule is executed multiple times, successive iterations of a loop body are executed simultaneously. The key characteristic of a modulo schedule is its length, which dictates how frequently the schedule can be repeated, and is known as the schedule's initiation interval (II). The II is the number of cycles that a producer must wait between feeding successive pieces of data into the pipelined loop, *e.g.* an II of 2 means that the pipeline accepts new data on every other cycle. The II directly affects the throughput of the pipelined loop.

The latency of the pipelined loop is the number of cycles that it takes for a piece of data to traverse the pipeline, *i.e.* the pipeline's depth. Assuming that all operators take at least one cycle to complete, the lower bound on the latency of the application kernel is determined by the length of the traditional schedule, ignoring resource and placement constraints, which is also the height/depth of the dataflow graph. Examples of other factors that can increase the latency of the loop are having a limited number of shared resources,

or hardware operators that take multiple cycles to complete. The ratio of the length of the traditional schedule to the length of the modulo schedule is the number of times the modulo schedule is executed to complete the first iteration of the application's loop body.

The physical resources within the Mosaic CGRA are represented as a datapath graph in the Mosaic toolchain. Like the dataflow graph of an application, the datapath graph represents the physical devices (*e.g.* functional units, multiplexers, and registers) as vertices and wires as the edges within the graph. The datapath graph can be used to represent the the architecture in both space and time during scheduling and execution as shown in [48]. Time is represented by unrolling the datapath, *i.e.* creating multiple copies of the datapath that represent different instances in time. These unrolled copies are then connected through the registers, which route the data that is written to them in one cycle to the same register in the next cycle. These register-to-register connections illustrate the flow of data through time during execution, and turn the 3-D space-time problem into a 2-D spatial problem when scheduling an application onto the architecture. Details on how the Mosaic toolchain maps applications to architectures are presented in Section 4.3. A key difference between the datapath graph representation for scheduling versus execution is that when scheduling an application to the architecture the datapath graph is only unrolled a finite number of times and the register connections from the last unrolled copy are wrapped back to the first unrolled copy. These wrapped connections represent the resource hazards (for the registers) that arise from the cyclic nature of the modulo schedule. When a datapath graph represents the execution of the architecture, the graph is essentially unrolled an unbounded number of times.

An example of the unrolling process for scheduling is shown in Figure 3.5, and by Friedman et al. [48]. Figure 3.5 illustrates the stages of unrolling the datapath graph one time. Figure 3.5(a) shows the initial datapath graph and Figure 3.5(b) shows the graph being replicated for an II=2 application. Figures 3.5(c) and 3.5(d) shows the register connections (in red) from cycles 0 to 1 and from cycle 1 wrapping back to cycle 0. Figures 3.6 and 3.7 show how the datapath graph is unrolled to represent execution. These figures, and the following text, walk through several steps of executing an example application's dataflow graph that was mapped to an architecture's datapath graph with an II=1 and latency of



(a) Initial datapath graph

(b) Datapath graph unrolled twice

(c) Register connections, only from cycle 0 to 1

(d) Register connections, from cycle 1 back to 0

Figure 3.5: Stages of unrolling a CGRA's datapath graph.

4. The steps are broken down into: the initial unrolling of the datapath graph, mapping of the application's first loop iteration to the architecture's datapath graph, and mapping the subsequent three iterations (of the application's loop) to the architecture's datapath graph. Finally, the modulo schedule is presented, which is color coded to show how different parts of the schedule correspond to different iterations in flight for the example.

1. A representation of the datapath graph is read into a CAD tool that will perform the schedule, placement, and routing (SPR) of the application with II=1, on four consecutive cycles of execution. Note that if the II of the application were greater than 1 then the datapath graph would be unrolled, creating virtual copies, to provide sufficient resources in both time and space. Details of how SPR unrolls a datapath graph are shown in Figure 3.5, described in Section 4.3, and by Friedman et al. [48].

2. Figure 3.6(b) shows the mapping of one iteration of the application's loop body to the architecture as a blue highlight, for cycles i thru $i + 3$ of iteration i. Note how the loop body is spread across successive clock cycles of the datapath graph's execution.

3. As a single iteration of the loop body is visualized over multiple cycles of execution, it is important to remember that the loop body will be pipelined. Figure 3.6(c) shows that use of a device by one iteration of the loop (in one cycle of execution) indicates use of the same device by other iterations in all previous and subsequent cycles of execution. The red highlights on devices indicate which devices are actively used by other iterations of the loop. The green and red highlights on registers indicate when a register is being written and read, respectively.

4. Figures 3.7(a) and 3.7(b) show the activity of iterations $i + 1$ (orange), $i + 2$ (red), and $i + 3$ (green) in cycles i thru $i + 3$.

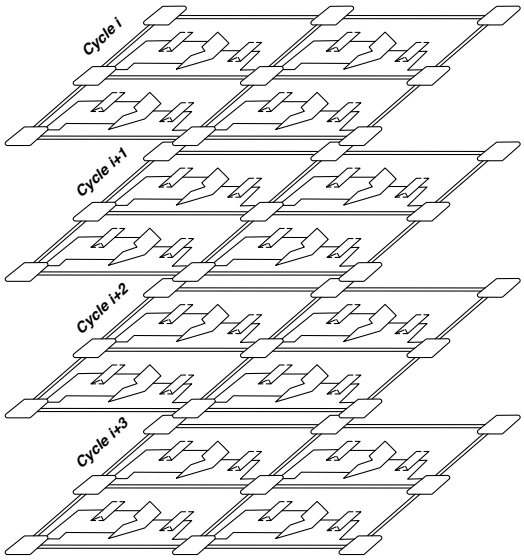
5. Finally, Figure 3.7(c) shows the modulo scheduled with $II = 1$. The activity from loop iterations $i, i + 1, i + 2, i + 3$ are shown in colors blue, orange, red, and green, respectively. Note that the modulo schedule can be observed in the execution of cycle $i + 3$.

For most applications within this domain throughput is more important than latency, and so the architecture and tool chain are designed to optimize II. In CGRAs it is possible to trade-off II with resource utilization. If the II of a pipeline is 1, then each resource performs the same operation on each clock cycle. If the II is k, then each resource performs k operations in a cyclic schedule, thus effectively increasing the capacity of the fabric by k times. Therefore, the capacity of the fabric can be increased by decreasing its throughput.

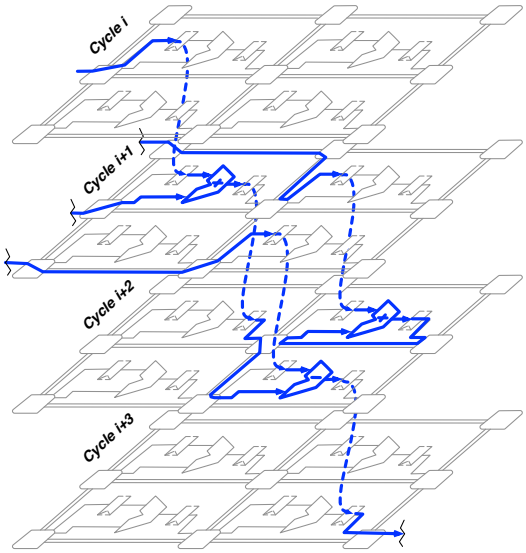
3.4 External Interface and Streaming I/O

To feed data into and out of the CGRA there is typically a collection of interface logic that is on-chip, but outside of the CGRA fabric. The Mosaic architecture uses a set of stream engines that move and/or shuffle data to and from a main system memory, which

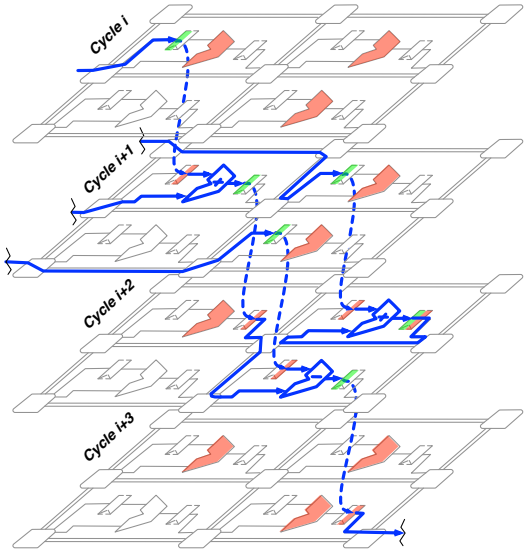
is shared with the host processor. These stream engines are similar to traditional DMA engines, but have the capability to execute independent threads of control that are tuned for looping access patterns, which are irregular and have low computational intensity. The stream controllers execute small stream accessor threads (or programs) that are generated by the Macah compiler from stream accessor functions in the application code. These functions specify the intended access patterns, and can execute small amounts of control code. This control code can be fairly sophisticated, and even include semaphores to interact with locking data structures. Stream accessor threads run in the shared address space with the host processor, pulling (or pushing) data from main memory into FIFO queues that are accessed inside of the CGRA fabric via stream receive and send operations. Stream accessor threads range in complexity from simple loops that walk over a data structure with either a regular or variable stride, to more complex iterative control flow that uses semaphores or other locking mechanisms to synchronize data access. Implementation, exploration, and optimization of the stream engines is beyond the scope of this dissertation.



(a) Datapath graph visualized for 4 cycles of execution

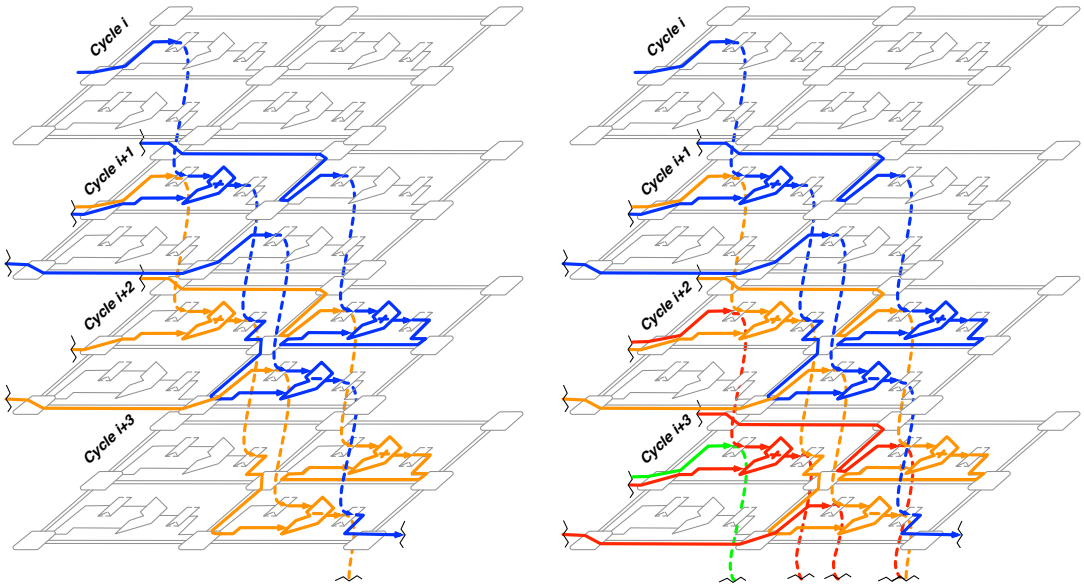


(b) Mapping iteration i, cycle i thru (i+3) in blue



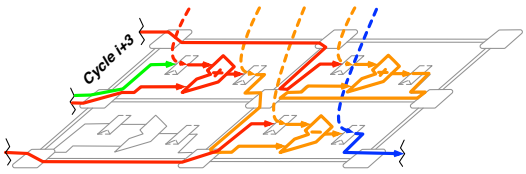
(c) Red highlights indicate which functional units will be utilized by other loop iterations when executing a modulo scheduled. Green and red highlights on registers indicate production and consumption of values.

Figure 3.6: Stages of executing a modulo schedule: mapping iteration i in blue.



(a) Adding execution of iteration i+1 in orange

(b) Adding execution of iterations (i+2) & (i+3) in red and green



(c) Modulo Schedule - $II = 1$

Figure 3.7: Stages of executing a modulo schedule: adding iterations i+1, i+2, i+3 in orange, red, and green, respectively. Display of the final modulo schedule.

Chapter 4

TOOLS

This dissertation is focused on exploring the architectural design space for coarse-grained reconfigurable arrays represented by the Mosaic architecture families. The two key tasks involved in this research were the development of CGRA architectures that isolated key design features, and the analysis of each feature using a suite of benchmark applications to simulate and characterize the features.

The development of architectures and applications for CGRAs is a complicated process that requires the support of specialized electronic design automation (EDA) and computer aided design (CAD) tools and algorithms. To provide this support, the Mosaic research group has developed a suite of tools, shown in Figure 4.1, that enable:

1. Architecture design through programmatic/scripted architecture generation and visualization.

2. Rapid application development.

3. Automatic mapping of applications to architectures.

4. Capture of simulated energy consumption, application timing, and signal activity.

The goals of the tools in the Mosaic toolchain are to facilitate architecture design and evaluation, and application development and mapping. Architecture design is accomplished with a plugin that I developed, which extended the Electric VLSI CAD tool that was developed by Sun Research Labs. Architecture analysis used both the ModelSim and Synopsys Verilog simulators, coupled with a collection of monitoring routines that I wrote in C and interfaced with the Verilog simulator using the Verilog Procedural Interface (VPI) set of the Programming Language Interface (PLI). The simulator monitoring routines used area,

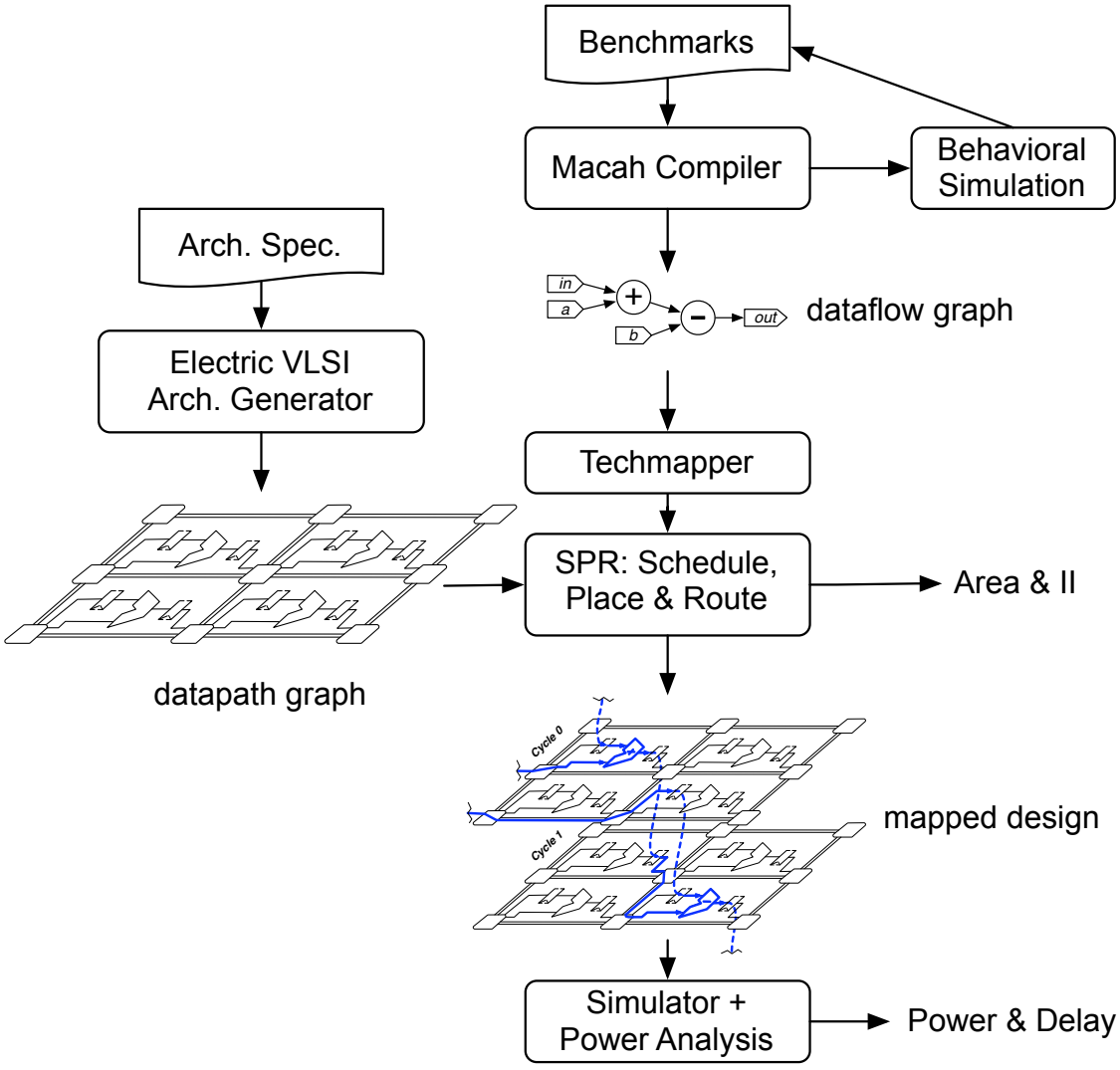


Figure 4.1: Mosaic Toolchain

delay, and energy statistics from physical models that were developed by other members of the Mosaic research group. Application development and programming of our CGRAs is done with the Macah language and compiler, and the SPR CAD tool. Both Macah and SPR were developed as part of the thesis research of other members of the Mosaic group, and the benchmark suite was collaboratively developed by all members of the Mosaic group. Finally, I developed the techmapper to fine-tune the applications in the benchmark suite to leverage architecture-specific features.

4.1 Developing application benchmarks

One recurring challenge for CGRA research, and a barrier to commercial success, has been a very low level of programming abstraction for developers, who are often required to write applications in Verilog HDL or VHDL. With these programming languages, the programmer has precise control over parallelism, scheduling, clocking, and resource management. However, this control typically buries the programmer in a sea of minutia. Several of these tasks, particularly scheduling and the pipeling of loops, could be managed effectively via tools using well-studied algorithms. With this in mind, the Mosaic research group has developed the Macah language and compiler that alleviates the need for the programmer to explicitly pipeline, schedule, and coordinate clocking for an application's computation kernel.

In Macah, the primary task of the programmer is to express parallelism and manage resource allocation. Macah [49] is a C-level language where the programmer expresses the core computation of the application as a set of nested and sequenced loop bodies. Currently, Macah is intended for applications that primarily contain loop-level parallelism, providing the programmer with a programming model that is suitable for pipelined execution. In the current programming model task parallelism is typically relegated to memory and stream accessor helper threads. Resource allocation is accomplished through explicit declaration of Macah operators, memory structures, and streams. Parallelism and resource consumption can be managed explicitly through unrolling of key loop bodies, a task that is simplified through the use of special "`FOR`" loops that are unrolled at compile-time.

Macah uses two key language constructs, kernel blocks and streaming I/O, to make

applications more suitable to pipelined execution on CGRAs. Streaming I/O provides FIFO communication channels between Macah tasks. They are a good mechanism for decoupling execution, and for moving data into and out of a kernel block by offloading the explicit address calculation to concurrent helper threads. Minimizing the number of address calculations in the kernel required for communication helps produce a more regular dataflow graph that is easier to pipeline. A stream is accessed by one or more send and receive operations and contains a bounded amount of storage that holds in-flight data. Streams simplify compiler analysis because they are not part of the memory hierarchy, and each steam is independent of other streams. For the purposes of the experiments in this dissertation the stream engines are modeled as C code that interacts with the Verilog PLI to service the CGRA simulation infrastructure.

We are focusing on CGRAs that use software pipelined loops to overlap execution of multiple iterations of a single loop. To designate regions of an application that are suitable for software pipelining Macah introduces kernel blocks that are demarcated by the `kernel` keyword and curly braces. Within a kernel block, the language and compiler do not guarantee that accesses to independent streams occur in program order. By allowing this relaxation in the traditional C execution model, accesses to the same stream are ordered, but it is possible to have multiple reads (receives) from a given stream before a single write (send) is ever done to another stream. This allows for the pipelined execution of the application's loops, with multiple iterations of the loop executing concurrently on the CGRA.

Application kernels written in Macah are transformed into dataflow graphs by the Macah compiler, and then mapped to specific architectures by the SPR CAD tool [48]. The dataflow graph is saved in Verilog HDL, and uses libraries of operators that are recognized by SPR as fundamental components of a dataflow graph. The dataflow graph libraries contain primitive operators, multiplexers, stream sends and receives, runtime and compile-time constants, and delay elements that forward data from one iteration to the subsequent one. Finally, there are several simulation-specific modules, such as data/token collectors and distributors that allow the DFG to be simulated in a dataflow manner. It is important to note that because Verilog HDL is the format used for a dataflow graph, it is possible to write applications directly in Verilog, or other languages that target Verilog, bypassing the Macah language.

The Macah compiler then performs a source-to-source translation on the application code that is outside of the kernel blocks, generating standard C code. The Verilog simulator is then interfaced with the remainder of the application's C code, which executes the remainder of the application and generates the applications test vectors.

4.2 Preparing applications for specific architectures

After an application has been transformed into a dataflow graph, our technology mapping tool (or techmapper), is used to perform optimizations on the dataflow graph prior to mapping to an architecture. The purpose of a techmapper is to optimize a generic dataflow graph when targeting a specific architecture or architecture family. Our techmapper performs the following phases.

1. Dead Code Elimination - Initial Pass
2. Strength Reduction
3. Commute Operator Inputs
4. Fold Constants Into Immediate Field
5. Dead Code Elimination - Final Pass
6. Fuse back-to-back Multiply-Add operators
7. Pack Boolean logic onto 3-LUTs

Dead code elimination, which is performed in phases 1 and 5, removes operations that perform no useful function on a given CGRA. One common source of unnecessary operations, when targeting the Mosaic architecture, are type-conversions in Macah. Mosaic CGRAs use a uniform 32-bit wide datapath, with operators that properly truncate and sign extend wide and narrow operands. Therefore, these operations can be safely removed from the dataflow graph. Another source of useless operations are arithmetic operations where one operand

is a constant value that forms an identity operation, *e.g.* addition or subtraction of zero. Identity operations typically arise through constant folding, where a constant value operand is folded into an immediate field of an arithmetic operation. Identity operations are removed by the last phase of dead code elimination.

Strength reduction is a common technique in compiler design that transforms expensive operations that have one constant operand or immediate value into less expensive operations. The Mosaic techmapper transforms division and modulus operations by a power of two into arithmetic right shift and bitwise AND (mask) operations, respectively.

The 3rd phase of the techmapper swaps operands of commutative operators if the left-hand operand is a constant value, since the architectures considered support an immediate field on the right-hand side of operators. Additionally, this phase will "reverse" and commute operands of comparison operations with a constant input. Table 4.1 shows the list of commutative operators and Table 4.2 shows the "reversed" comparison operators.

Operator	Symbol
Addition	+
Multiplication	×
Equality	==
Inequality	!=

Table 4.1: Commutative Operators

Operator	Symbol	Reversed Operator	Reversed Symbol
less than	$<$	greater than or equal	$\geq$
less than or equal	$\leq$	greater than	$>$
greater than	$>$	less than or equal	$\leq$
greater than or equal	$\geq$	less than	$<$

Table 4.2: Transformation of comparison operators

The next phase of the techmapper folds small constant values into an immediate field in many opcodes for the ALU, shifter, or MADD devices. Constant values 0 and 1 appear frequently in many dataflow graphs and are inexpensive to encode as extensions to the set of opcodes required by the Macah language. To minimize hardware overhead, only the right hand side (or B) operand can be replaced with an immediate 0 or 1 value. The design of the shifter that we use provides two modes of operation: shift by arbitrary values, or shift by 2^x. This allows the shifter to efficiently encode shifts by common small powers of two, namely 1, 2, 4, or 8.

The last two phases of the techmapper fuse specific patterns of binary operators into a single ternary operator. The first fuses back-to-back multiply-add operations, where there is no fanout of the intermediate value, into a MADD operation. Fused multiply-add (or MADD) operations are desirable because multipliers can be cheaply extended to MADDs as part of their partial product tree. Further details about MADD units are provided in Section 8.2. The last phase fuses multiple single-bit logical operations into a 3-input logical operation that fits into the 3-LUTs in the Mosaic architecture's control path. The list of supported transformations is shown in Table 4.3.

First operation	Second operation	Fused operation	Example expression
AND	AND	3-AND	$a \,\&\&\, b \,\&\&\, c$
OR	OR	3-OR	$a \parallel b \parallel c$
AND	OR	AND-OR	$(a \,\&\&\, b) \parallel c$
OR	AND	OR-AND	$(a \parallel b) \,\&\&\, c$

Table 4.3: Back-to-back logical operators that can be fused to form 3-LUT operations.

4.3 Mapping applications to architectures

Once the dataflow graph has been optimized for the target architecture, the Mosaic SPR tool performs scheduling, placement, and routing of operators and operands in the dataflow graph onto the architecture's datapath graph. SPR is iterative in nature and repeats mul-

tiple phases to find a mapping that minimizes the initiation interval (II) of an application loop's modulo schedule.

The initial scheduling phase of SPR is based on the iterative modulo scheduling algorithm developed by Rau et al. [29]. To determine the length of the modulo schedule, the scheduling phase estimates both the minimum resource and recurrence II of the loop body. The minimum resource II is how many times the CGRA will have to be unrolled in time to provide sufficient resources to execute the loop body. The recurrence II is the number of clock cycles required to execute the longest loop-carried dependence. Note that in the current generation of the tools and architecture each operation takes one cycle, so this translates to the loop-carried dependence with the largest number of operations (*i.e.* the longest critical path). A loop-carried dependence indicates that the data in one loop iteration depends on one of its previous values, which was computed in some earlier iteration (*i.e.* a feedback path). Therefore, the dependent operation cannot start executing until the source operation is complete, and so the longest feedback path in the application loop limits how much each loop iteration can overlap. The larger of the resource and recurrence II is used as the initial modulo schedule length; SPR then schedules each operator into a time slot of the modulo schedule. Once an initial schedule is formed, each operator is pseudo-randomly placed onto an available device in the CGRA architecture.

Placement and routing an application onto an architecture faces a particular challenge for time-multiplexed spatial architectures like CGRAs. The problem is that an operation has to be placed on a device in both space and time. Because SPR targets statically scheduled architectures, operands have to arrive at the operator inputs in the same cycle that they are to be consumed since the operators are unable to stall or store values until the proper cycle. Therefore a route between source and sink devices must include enough registers to provide the necessary latency between production and consumption. This compounds the already challenging 2-D placement problem by adding a 3rd dimension to the solution space. To simplify the task of placing and routing in both space and time simultaneously, SPR starts with the physical datapath graph that represents the architecture and makes multiple copies of it that represent different instances in time, as described in Section 3.3. By connecting these unrolled copies through the registers (Figure 3.5), the datapath graph

shows how data will be routed through time: data that is written to a register in one cycle is available from the same register on the next cycle. The cyclic (back) edges that connect the registers of the last unrolled copy to the first unrolled copy also show the structural hazard that occurs when the modulo schedule is executed repeatedly, where subsequent iterations of the schedule have to contend with data movement from previous iterations. As described earlier, these register-to-register connections turn the 3-D space-time problem into a 2-D spatial problem, which allows the placement and routing tasks to be solved with known CAD algorithms.

SPR uses the simulated annealing algorithm [50] to optimize the placement of operators to devices. Simulated annealing is modeled after the annealing phenomenon found in material science, particularly in the formation of crystalline structures. The idea is that the placer is allowed to move and swap operations throughout the fabric, and the quality of each move is judged by a distance function that quantifies how good or bad an operator's position is with respect to operators that produce and consume its operands. The probability that a bad move will be permitted is proportional to the "temperature" of the annealing phase, which starts off high and is cooled during the placement process. Because SPR has transformed the 3-D space-time problem into a 2-D spatial representation, moves by the placer can change when an operation occurs as well as where it occurs in the fabric. Therefore, the placer actually performs limited rescheduling as it anneals the placement, and will increase or decrease the latency between a value's production and consumption to allow sufficient time to accommodate the revised placement and scheduling. SPR's simulated annealing algorithm is based on the VPR tool for FPGAs; a more detailed discussion of the algorithm, including the cooling schedule, is provided in [51]. The distance (or cost) function used by the annealer is architecture-specific. To maintain SPR's architecture-independent design, the distance function is customized by a plugin generated by the architecture generator, which is detailed in Section 4.3.1.

The last stage of SPR routes signals between producers and consumers. The challenges for SPR differ from traditional FPGA routing algorithms because the CGRA is fixed frequency, highly pipelined, and uses both time multiplexed, scheduled channels and shared static channels. The core algorithms used are based on the Pathfinder [52] and Quickroute

[53] algorithms. The algorithm for sharing static channels and further details on SPR are described in [48].

4.3.1 Specializing SPR algorithms for an architecture

SPR is an architecture-independent tool that uses architecture specific plugins to tailor the algorithms and behavior to specific architectures. Each of these plugin modules, described below, is customized by the architecture generator to convey key details of the architecture to the various CAD algorithms.

- Base cost initializer: establishes how expensive each device and routing resource is to use during both placement and routing.

- Configuration writer: converts the internal SPR representation that maps application-to-architecture into a programming file for the architecture. It also can be used to optimized (or customize) SPR's usage of specific devices in the architecture.

- SPR technology mapper (techmapper): establishes a set of mappings from each dataflow graph operator to one or more physical devices in the architecture's datapath graph. It allows for one-to-one, one-to-many, and many-to-one style operation-to-device mappings.

- History initializer: sets the Pathfinder algorithm's default history cost for each routing structure. It is also used to selectively enable and disable routing paths in specialized routing structures, particularly rotating register files.

- Placement distance function: determines the estimated number of multiplexers between any two points in the datapath graph at a given latency.

- Placement selection function: returns the set of legal placement locations for a given operation. Allows the set of possible placement locations to be customized for each operation, rather than being a generic set for each operator type. This optimization

can improve both the placer's runtime and quality of result by reducing the search space.

- Routing distance function: determines the fewest number of multiplexers between any two points in the datapath graph at a given latency and identifies pairs of points that have no legal path between them. It is used to calculate the estimated path length of a route for the $A^\star$ algorithm.

- Verilog import module: maps each device type in the architecture's datapath graph to SPR's internal data structures. In particular it is used to identify routing structures, establish and form multiplexers from distributed structures in the datapath graph, and merge multiple logical nets that correspond to the same physical wire.

Distance functions

Both the placement and routing algorithms in SPR use cost-based algorithms to optimize the mapping of an application to the architecture. The algorithms try to minimize the average path length between producers and consumers; the placer works with estimated path lengths, while the router searches and negotiates for the shortest path. The path length is determined by the number of multiplexers used between producer and consumer. Not all multiplexers have equal cost; more expensive ones represent resource bottlenecks or devices that consume more energy.

Placement distance functions

In the placer's distance function, the cost of a path represents the estimated distance (*i.e.* separation) between any two points in the architecture with a specific latency. A placement is illegal when a source and sink device are unable to communicate, or unable to communicate in the appropriate amount of time for a given producer / consumer pair. As mentioned in Section 4.3, operands have to arrive at precisely the right time for the computed schedule. Therefore if a signal cannot traverse the distance between the source and sink within the scheduled latency, it is an illegal path. Similarly, if the signal cannot find a sufficient number

of registers along a path between source and sink, that is also an illegal path. This second case only occurs in very constrained parts of the architecture where the distance estimate can prove that no longer path exists, which is quite uncommon.

The placer's distance estimate is a composite value that combines both the logical distance across the fabric and the base cost of each resource used along the predicted path between two points. The logical distance is defined by the topology of the available communication resources (*i.e.* interconnects). For example, in a grid topology (like the Mosaic CGRAs) the logical distance is approximately the physical, Manhattan distance; other topologies (*e.g* a tree or fat-pyramid) do not have a similar correspondence between physical and logical location. Combining both logical distance and resource cost creates a search space that favors topological-locality, optimizing for short low-cost paths, and promotes trade-offs between long-distance paths with cheap resources and short-distance paths with expensive resources.

The base cost is used to indicate which resources are more expensive (or cheaper) than others, thus biasing some communication paths or patterns. For example, the Mosaic architecture is a clustered architecture, which means that there is more intra-cluster communication bandwidth than inter-cluster bandwidth. Therefore, the multiplexers in the cluster-to-cluster interconnect have a higher base cost than the intra-cluster multiplexers, as shown in Table 4.4. This allows an application to spread out across the fabric, but does encourage the tools to use intra-cluster communication when possible. In general, when the distance function is unable to precisely predict the number of mandatory (*i.e.* unavoidable) registers along a path, it is better for it to overestimate the number of registers than to underestimate them. Therefore, the placement's distance function is considered pessimistic with respect to the number of registers along a path (*i.e.* the path's minimum latency). When the distance function is unable to compute the correct number of mandatory registers available along a speculative path, guessing that there are more of them gives the router the opportunity to correct the overestimation by bypassing any unnecessary ones. If the placer were to underestimate the number of mandatory registers, then the path would require more clock cycles than the scheduled lifetime of the value, and thus the value cannot possibly show up at the consumer on time.

Device Type	Cost
Multiplexer in a processing element	1
Intra-cluster Multiplexers	5
Inter-cluster Multiplexers	10
Idealized functional unit	1
Universal functional unit	5
S-ALU functional unit	1
ALU functional unit	1
MADD functional unit	5

Table 4.4: Base costs for placeable and routable devices.

Routing distance function

The routing algorithm is based on the $A^\star$ algorithm [54], which computes a cost to represent the remaining distance of the forward part of the routing path; the distance is similar to the placement algorithm's, although it is required to be optimistic rather than pessimistic. Using an optimistic cost is important to prevent the $A^\star$ algorithm from incorrectly pruning viable paths, and thus the routing cost function has to be correct or underestimate the number of multiplexers on a given path. This $A^\star$ forward path cost (*i.e.* distance) is then combined with both a present sharing cost and a history cost to determine the cost of the path so far, and to estimate the remaining portion. The present sharing and history costs are part of the Pathfinder algorithm that represent current and historical congestion (overuse) of a resource. The present sharing cost indicates how many signals are currently using a given routing resource. Allowing values to share routes is an illegal condition for the final configuration, but it is very useful during routing to let different signals fight over which really needs to use a given resource, and which have the possibility of choosing other, less congested routes. The history cost allows the routing algorithm to avoid resources that are historically congested. A secondary use for the history cost is to selectively disable resources in complex architectural structures, namely the Cydra rotating register file. Using

this technique, routing resources can be disabled on a per phase basis by initializing the history cost to infinity.

Cost functions

The four main methods for customizing the various costs used by SPR are the base cost initializer, the history cost initializer, and the placement and routing distance functions. The base cost initializer sets the base cost for each device in the architecture. In the Mosaic architecture the standard base cost for many routing and placeable devices is shown in Table 4.4. Devices are classified as either routing or placeable devices, depending on their function and how they are managed by the tools. Placable devices are the hardware resources that perform specific computations or operations: the functional units, stream send and receive ports, memory read and write accesses, and registers that are initialized with a specific constant value. Routing devices are the multiplexers in the architecture. Other devices in the architecture such as unnamed multiplexers and stream or memory ports have unit base cost. Details of the different functional units are provided in Chapter 8. The base cost for the placable devices is used to bias the placement algorithm so that it favors devices that consume less energy, or are more common, just like the base cost for the routing resources is used to bias the placer and router towards communication paths with less expensive resources.

The history cost initializer resets each routing structure in SPR to its default values after each full placement and between complete iterations/attempts of the schedule, place, and route phases. For most structures, the default value is 0, but is infinity for selectively disabled structures. The initializer can be customized for each set of connections and each phase of the schedule.

Abstractly, the placement and routing distance functions compute the distance between any two locations in the grid. This calculation is complicated by the fact that communication not only happens between two points in space, but also two points in time. Due to the pipelined execution of the application's dataflow graph, each operation occurs at a fixed time during the schedule's execution. Because the system is statically scheduled, data

must arrive at the destination at the correct cycle (*i.e.* when the consumer is "listening" for it). As described earlier, the distance function determines (or estimates) how many multiplexers are traversed when any two points in the grid communicate at a specified amount of latency. The challenge for the router is to find the shortest path through the topology that also traverses the correct number of registers. Because the Mosaic architecture runs at a fixed frequency, there are registers that appear at regular intervals, bounding the longest combinational path in the cluster or interconnect. One method for increasing the number of registers on a path is to traverse a loop in the architecture that has at least one register on it.[1] These types of loops occur most frequently in storage structures such as register files or shift registers. Typically, the distance function calculates the minimum number of registers along the shortest path between two points at any latency, and then estimates the number of muxes required to pick up additional registers to meet the latency requirement.

Example of calculating the placer's distance estimate

Table 4.5 and Figure 4.2 provide several examples for estimating the distance between source-sink pairs at different latencies. Figure 4.2 presents an abstract block diagram of a Mosaic architecture, with 4 clusters joined by a grid. Within each cluster are 4 processing elements that contain the functional units, a register file, and a crossbar that connects them. Processing elements are labeled with their cluster coordinate. Table 4.5 shows the calculated cost for several source-sink pairs at different latencies. To simplify the example, the cost is reported as the number of muxes, without incorporating any non-unit base costs. Furthermore, assume that a signal must cross one mux to get off of the cluster crossbar and two muxes to go between clusters across the grid. Notice that source-sink pairs that have insufficient latency to traverse the forced registers report infinite cost.

To enable the distance function to compute the logical distance between two points in a topology, and calculate how much forced latency there is along a given path, the distance

[1]Note that unregistered loops in the architecture present a particular problem for the tools because they do not increase the latency of the path; they are not supported by the tools and considered an illegal design pattern.

Signal Class	Source	Sink	Latency	Path Cost
Intra-cluster	(0,0)	(0,0)	1	2
	(0,0)	(0,0)	2	2
	(0,0)	(0,0)	3	5
Inter-cluster	(0,0)	(0,1)	1	inf
	(0,0)	(0,1)	2	5
	(0,0)	(0,1)	3	5
Inter-cluster w/turn	(0,0)	(1,1)	1	inf
	(0,0)	(1,1)	2	inf
	(0,0)	(1,1)	3	6
	(0,0)	(1,1)	4	6
	(0,0)	(1,1)	8	9

Table 4.5: Examples calculations of placer cost function for an architecture shown in Figure 4.2, with 2 forced registers between clusters. Source and sink coordinates are presented as (X,Y) and latency is in clock cycles.

function is specialized for that topology and requires a coordinate system to determine the source and sink locations. The Mosaic CGRA uses a hierarchical coordinate scheme to reflect its hierarchical interconnect. For the top-level grid interconnect, each cluster is labeled with an (X,Y) coordinate, which can be used to compute the Manhattan distance between clusters. Inside of each cluster the topology is flatter due to the crossbar, thus sources and sinks are much closer to each other. When evaluating the distance within the cluster for placement, the only distinction between locations are if a producer and consumer are on the same functional unit or different functional units. To determine how many multiplexers remain on a path between producers and consumers, there are several layers of routing within the cluster that include multiplexers: directly attached to consumer devices (*e.g.* within the processing element), for the read ports within the crossbar, on the outbound path from local producers, and on the inbound path to the cluster from the

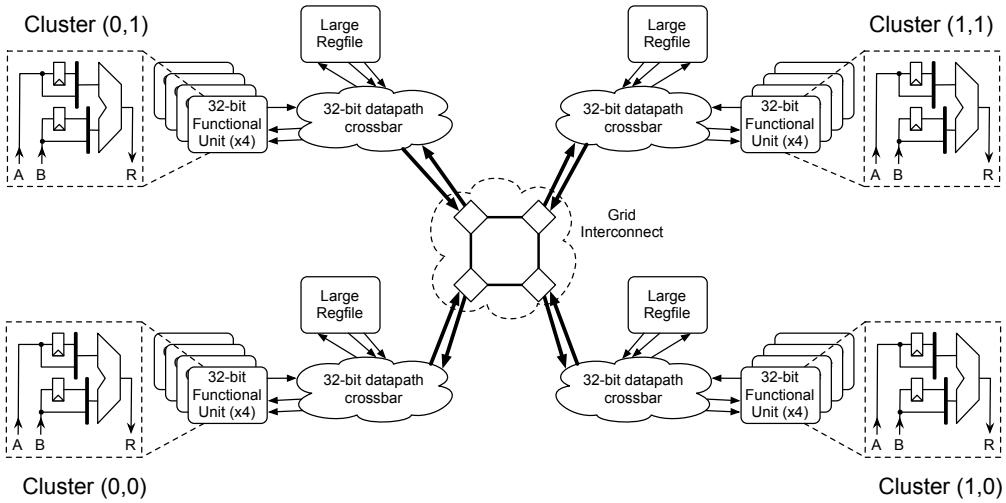


Figure 4.2: Placement example: DPG with 4 clusters and grid coordinates (X,Y).

grid interconnect. For the internal cluster coordinates, placeable devices such as functional units, stream ports, and memories are labeled with a device ID. The local intra-cluster crossbar read muxes are labeled with a routing ID, which also incorporates the ID of the device attached to the multiplexer's output, and muxes within a processing element inherit a routing ID directly based on the local device ID. Using this coordinate scheme the plugin is able to compute both the distance in terms of multiplexers and the latency between any two points in the architecture.

The first three examples from Table 4.5 are intra-cluster pairs between PEs 0 and 1. The first two paths cross muxes on the feedback path and retiming register in the functional unit. The third example requires one additional register, which it picks up in the register file; this requires one additional trip across the crossbar, plus two muxes in the register file itself. The next three examples are inter-cluster signals. The first of these signals only has a latency of 1 cycle, but there are two forced registers between the clusters and so there is no legal path. The next two paths require one mux as they leave the first cluster, two in the interconnect, another one in the sink cluster, and finally the mux on the functional unit's retiming path. The last five signals are inter-cluster and require a turn in the grid interconnect. This turn requires an additional forced register and one more mux. As a

result the first two examples do not have sufficient latency to get from source to sink. The cost for the last three signals can be computed from previous example paths.

Optimizing the placement function

A method for filtering the set of devices that the placer will examine for a given operator, is an ancillary plugin to the placement distance function. This filter method can be used to eliminate illegal placement positions (as opposed to bad ones) during each step of the simulated annealing algorithm. The primary use for this is to help manage constant value operands. In the Mosaic architecture, constant values can be placed in globally accessible register files or in local, private, register files. A problem occurs when a constant value is placed in a local register file that is not local to its consumer, since there is no path between source and sink. This filtering method is used to constrain the set of available placement positions to the globally accessible register files or the local register file that is private to the sink operator. Limiting the placement of the constant values can increase the rate at which simulated annealing converges to a good solution since the simulated annealing algorithm spends less time exploring illegal parts of the solution space.

Mapping operators to devices (SPR Techmapping)

Internal to SPR is a plugin called the techmapper. Unlike the techmapper that has been previously discussed, SPR's internal techmapper establishes the mapping of operators in the dataflow graph to devices in the datapath graph. For example, in an architecture that contains both ALU and MADD functional units, the techmapper indicates that an addition operation can be mapped to either the ALU or MADD functional units, but that a multiplication can only be mapped to the MADD functional units. The techmapper is customized by the architecture generator based on the physical devices available. Architectures with multiple devices that provide overlapping functionality can adjust each device's base cost to bias the placement algorithm's device selection. For example, the experiments in Chapter 8 test architectures with both ALU and MADD functional units. While both devices can perform addition, using the MADD hardware consumes more energy than the ALU. There-

fore the placer is setup to favor ALUs over MADDs when possible by making them have a lower base cost, as shown in Table 4.4.

Generating the configuration

Once SPR has completed mapping the algorithm to the architecture it is necessary to generate a configuration that is loaded into the architecture for simulation. In general the configuration for any device, *i.e.* multiplexer or functional unit, consists of a single bit that marks the device as configured, and one or more arrays of configuration words that describe the function of the device for each phase of the modulo schedule. The configured bit not only enables the device, but also can be used to indicate that power mitigation techniques such as clock gating could be applied to unconfigured devices, reducing energy consumption. The array of configuration words is stored locally in each device and is indexed by the global modulo counter that synchronizes execution of the architecture.

There are three structures used in the Mosaic architecture that deviate from the simple method of configuration: compound functional units, register files, and variable shift registers. Compound functional units combine several primitive functional units (*i.e.* ALUs, shifters, and MADDs) into a single logical functional unit and are described in detail in Chapter 8. As a single logical functional unit, compound functional units are presented to SPR in a similar manner as primitive functional units: they are placed and scheduled as an atomic unit rather than as their constituent components. Examples of compound functional units are the universal functional unit and the Shifter-ALU (S-ALU) functional unit that are described in Section 8.2 and Figures 8.2(b) and 8.2(a), respectively. When an operator is mapped to a compound functional unit, the configuration writer customizes the configuration to map to the proper sub-device and programs the compound functional unit's output mux appropriately.

Register files present a particular challenge for SPR. The problem relates to how SPR handles a decoder structure. In a register file, on each write port a value enters the structure and then is steered to one physical register or row of an embedded SRAM array. The problem is that SPR's router does not contain a mechanism to enforce that a value cannot fan out

in a decoder and must be routed to only one output of a decoder structure. A possible solution to this problem is to use a data structure and algorithmic technique similar to the way that time-multiplexed routing structures are controlled, which is discussed in [48]. This has not been implemented in the current version of SPR; as a result decoders do not exist in SPR's representation of a datapath graph. Instead, they transform into many distributed multiplexers on the "output" signals of the decoder.

As a result of this technical limitation, it is possible for a signal that is written into a register file to fan out to multiple registers or rows of an SRAM array. However, this operation is not permitted by large register files that use SRAM arrays to back their logical storage. To correct this problem, the configuration writer post-processes the results of routing and merges all concurrent writes of a value to a register file into a single write. To ensure a correct configuration, concurrent writes are collapsed onto the register file entry (row) with the longest lifetime. The lifetime is the number of cycles between when a value is written and when it is last read from that location. In practice, this requires finding the last read of the value, moving all earlier reads to use the same register (or row) as the last read, and then removing extraneous writes of that value. The process for consolidating concurrent writes is a safe transformation and straightforward to implement.

The last challenge for the configuration writer is managing the configuration for shift registers. Shift registers are characterized by a common control signal that either holds a value in place or advances all entries in the structure simultaneously. Like the decoder for register files, SPR lacks the infrastructure to manage a shared control signal that is used for the shift enable signal. As a result, each register in the shift register structure is independently controlled, and SPR can (and will) arbitrarily advance signals deeper into the shift register than is strictly necessary. These superfluous writes to registers can lead to a significant increase in energy consumption. The configuration writer is able to change the control signals in a shift register to provide densely packed signals that only go as deep into the shift register as strictly necessary. To rewrite the configuration (*i.e* routing) of the shift register, the configuration writer determines the lifetime of each value and the total number of values written to a structure. During this process, the arrival times and departure times for each value are fixed by the rest of the application's configuration, and

the only requirement for a shift register is that each value shifts forward when a new value enters the structure. Using these conditions, the configuration writer is able to achieve a dense packing of values that helps minimize energy consumption.

4.4 Building Coarse-Grained Reconfigurable Array Architectures

Our automatic generation of CGRA architectures simplifies the process of creating architecture families, where specific architectures differ only in the presence / absence of specific features, the ratio of resources dedicated to a given architectural feature, or their distribution. To enable this automatic generation, we have created a plugin to the Electric VLSI CAD tool [55] for describing and creating a specific CGRA design. Electric VLSI is an open source EDA CAD tool that provides tools such as schematic capture, Verilog block design, physical layout, layout versus schematic validation, and design rule checking. Hardware designs can be described using mixed schematic, register transfer level (RTL), and block level design techniques, and then be output in a hierarchical Verilog RTL representation.

Because the tool is written in Java and open source it was possible for us to create tasks and routines that would manipulate and programmatically assemble various hardware designs that had been manually created. The two main thrusts of our work for the architecture generator was the development of a library of basic structures that served as the building blocks for full CGRA designs, and writing the algorithms and program code that would stitch them together. Key features of the architecture generator include the ability to specify the topology of the architecture, the composition of the CGRA tiles (or clusters), and automatically wire together all of the components.

Another attractive aspect of using Electric as the foundation for the architecture generator was that it has a GUI and is fundamentally designed for interactive use. This made it well-suited for creating visual representations of the CGRA architecture, and for showing how an application was actually mapped onto a specific architecture. The visualization of the application to architecture mapping showed not only signal activity, but also resource congestion, which proved very useful for debugging architecture resource utilization.

4.5 Simulating CGRA Behavior

Simulation of the application mapped to a specific CGRA architecture is used for both validation and performance measurement. To bring up new applications, architectures, and CAD algorithms, simulations are used to validate the entire Mosaic toolchain. Once an architecture is validated, simulation is used to determine performance and energy consumption of an application mapped to an architecture. We used Synopsys VCS and Mentor Graphics ModelSim, both industry standard Verilog HDL simulators, to provide the cycle accurate results.

4.6 Power Modeling

Traditionally, power modeling of a digital design in CMOS technology focused only on the dynamic power contribution of the architecture, as the static power contribution was minimal. However, advances in modern VLSI CMOS processes have dramatically increased the leakage current of transistors. This shifted the balance of power consumption to an almost even distribution between static and dynamic sources. When the only cost for quiescent logic was its area, architects could dedicate large swaths of a design to specialized logic that was only sporadically used, taking advantage of the vast amounts of silicon real estate available. As quiescent logic starts to consume a nontrivial amount of energy, the advantage of time-multiplexing reconfigurable logic starts to outweigh its overhead, especially since the configuration memory required for time-multiplexing is typically better optimized to minimize transistor leakage when compared to general purpose transistors. This creates a new inflection point in the trade-off between discrete specialized logic blocks and the flexibility of time-multiplexing reconfigurable logic.

Currently there is no standard method for performing power analysis of a digital design at the architecture level. One possible approach is to use tools such as PowerMill on a complete circuit layout to estimate power; however this is extremely slow and does not scale well with larger designs. Previous research efforts into power modeling at the architectural level of complex systems are the Wattch project [56] and the work by Poon et al. [21] on FPGAs. Wattch provides an operation-centric, activity-based study of modern microprocessors, fol-

lowing the activity of each macroscopic stage of the processor's pipeline while simulating the execution of some application. Poon et al. created a power analysis tool that fit into the VPR FPGA framework and conducted a survey that provides an analytical exploration of a Xilinx FPGA. Both of these approaches are similar, basing their analysis of signal activity and requirement for using another tool that provides either application simulation or mapping. A key difference between Wattch and Poon's work is that energy expenditure in spatial architectures depends on the relative locations of operations as well as the relative execution time of operations. For this reason, it is sufficient for Wattch to tie into the SimpleScalar processor simulator, but Poon's work requires placement and routing data as well as simulation, so it ties into the VPR place and routing tool for FPGAs. Wattch uses SimpleScalar to provide both the architecture description and the activity estimation for each structure during execution. Poon's work uses the mapping from VPR to describe the spatial relationship of the application and a synthetic activity model to stimulate the architecture's physical models. Similar to Poon's work with FPGAs, effective modeling of energy consumption in CGRAs requires placement and routing data in addition to simulation data.

To perform power analysis for a CGRA at the architectural level, we used an approach closest to Poon's methodology. However, we were unable to directly retarget either of the two approaches, given how closely both tools are tied to their simulation and mapping infrastructure. Instead we used their work as a guide for our experimental methodology, for characterization of physical devices, and to extrapolate what areas of a CGRA will have the largest impact on energy consumption. A key difference between FPGA and CGRA analysis is the coarse-grained and time-multiplexed nature of the CGRA. Time-multiplexing the coarse-grained devices allowed us to move to an operation-centric analysis over a suite of applications (similar to Wattch), rather than a signal level analysis of a synthesized hardware design. Therefore, energy modeling and analysis was tied into a Verilog HDL simulation of the Mosaic CGRA fabric. Like Poon's work with VPR, architecture specific extensions, such as the base cost initializer presented in Section 4.3.1, are added to SPR to help guide the mapping of application to architecture to a more energy-efficient solution.

Research by Chandrakasan and Brodersen [57] has shown that the dynamic power consumption of arithmetic units can vary substantially based not only upon the value of the

inputs, but also on the history of the inputs. For example, in architectures that preserve data correlation during computation, a correlated data source may switch 80% less capacitance than an uncorrelated, random data source. Similarly, a time-multiplexed adder may switch as much as 50% more capacitance than a non-shared adder.

These examples are of particular interest given both the application domain of interest and the time-multiplexed nature of CGRAs. Within the DSP, streaming data, and scientific computing application domains it is common to find data sets with a large amount of correlation within the data. For these applications, finding an activity model that properly captures the interplay between correlation in data and the potentially destructive interference of mapping these applications to a time-multiplexed architecture is particularly challenging. Another complicating factor is that the probability of a transition on the output of a functional unit is highly dependent on the logic function being implemented, a problem that is compounded when groups of combinational logic are fused together. As a result, it is unrealistic to assume that the inputs to each physical structure are modeled by a independent random activity function, and thus we would have to use a much more sophisticated model. For this reason a much simpler approach to energy modeling is to collect simulation data for real applications of interest and use that to predict the future energy consumption of applications mapped to that CGRA. One advantage of using a data-driven modeling environment is that it allows us to capture and measure the advantages of architectural features or mapping algorithms that preserve any existing data correlation.

To address the challenges of accurately modeling the correlation within a data stream, the transition density of a computation within a functional unit, and the potentially destructive interference of a time-multiplexed architecture, we used an empirical, trace-based analysis. This resulted in a key difference between our work and previous work by Wattch and Poon. Specifically, Poon's work relied entirely on analytical activity estimation to generate per structure / wire signal activity. Alternatively, the Wattch project generated per-device activity by capturing a simulation trace and then aggregating the results on each device into a single activity number. This approach is sufficient to capture data correlation on simple structures, such as interconnect drivers and registers, but it is unable to capture the transition density and complex interplay of operands on more complex structures such

as ALUs. For simple structures, where the energy consumed by each bit-slice of a device is independent, our empirical approach is similar to the one used by Wattch. However, we used a more sophisticated approach for the functional units that accounts for the unique structure required to compute each operation, as described in a subsequent section.

4.6.1 Power-aware Architectural Simulation

To provide an empirical analysis of energy consumption, we need an infrastructure that allows us to study the activity levels of a CGRA executing a real application. To accomplish this we have used the Verilog Programming Language Interface (PLI) to annotate a CGRA's datapath with C routines that track activity and power directly. One main advantage of using the PLI is that it provides visibility into the simulation and permits the monitoring functions to examine the data values of a processing element's inputs.

The energy aggregation code uses native Verilog code and tasks to track the structural activity (*e.g.* memory reads or writes) or number of bits that transition during the execution of a benchmark, and PLI methods to aggregate the results across the architecture. Energy consumption is monitored on either a per bit, or per structure basis, depending on the type of structure. Functional units and memories track their energy for each transaction (*e.g.* operation, read, or write) executed by that structure. Multiplexers, buffers, registers, LUTs, etc. track energy for each bit that toggles in the structure.

To track the number of bit transitions in a logic block, each block maintains a local, internal copy of the previous value for each input. A power monitoring task computes the Hamming distance between the previous and current inputs for both combinational and registered logic blocks. Combinational blocks record the number of bits that toggle for each input signal that changes, and registered blocks record bit toggles on a per clock cycle basis. Once the simulation is complete, a PLI method is called to compute the total dynamic energy per structure by multiplying the estimated energy cost per bit and the number of bit toggles. The PLI then combines the dynamic energy with the energy consumed by transistors leaking over time, the costs to reconfigure logic, and clock the register structures.

The two types of structures that record energy on a per transaction basis are the func-

tional units and the memories. The memory compiler used provides the dynamic energy per read or write, and the static power for each design. The Verilog description of a memory or large register file records the accumulated energy for each read and write operation performed by the memory array. Functional units record the energy consumed for each operation performed and each transition on its inputs and outputs. The energy consumed per operation or due to signal activity is determined via a table lookup that is customized for each hardware device and operation. This methodology was derived from [58] and [59]; validation tests showed that on average our table based approach was within ∼5% of PowerMill simulations, with the worst case estimates being 2× off. Signal activity for the inputs and outputs is approximated using the Hamming distance between current and previous values. This approximation captures the number of bit transitions between values while compressing the lookup tables to a reasonable size.

An alterative approach we originally tried to computing the energy consumed during execution was to instrument each register, buffer, and multiplexer in the architecture with a PLI routine that computed the Hamming distance between a structure's current and previous outputs. While it was simpler to implement than using Verilog tasks, the overhead of making a call into the PLI routines on a per event basis in Verilog proved to be prohibitively expensive, both in runtime and in simulator memory required. One source of this inefficiency was that the heavy use of PLI routines disables many of the optimizations performed by the Verilog simulator. Overall, moving the aggregation of bit transitions from the PLI routine to native Verilog code, and the reduction in PLI calls, lead to approximately a 10× improvement in runtime and 10× reduction in memory required for simulation.

4.6.2 Creating Power Models

Creating power models for the Mosaic CGRA architectures required two different approaches, one for complex structures and the other for primitive devices. The first approach uses automatic synthesis, module, and memory compilers for specific components (such as ALUs and memories) that are well studied for power-efficient designs. The memory compiler reports area, delay, and power characteristics for the memory structures. Behavioral and

structural functional units are synthesized to physical designs, and the power characteristics are extracted using PrimeTime from Synopsis. The lookup tables that provide energy consumption based on a functional unit's activity are generated by running test vectors through PrimeTime and aggregating results that have matching input / output Hamming distances. The second approach, which is well-suited to modeling the interconnect, was to characterize the behavior of certain small structures in the CGRA (such as wires of different length, buffers, multiplexers, and switchboxes) on a per bit basis, using full custom design in a specific VLSI process and characterize them using HSPICE. Because CGRA designs are repetitive by nature, we were able to characterize all of the individual components using these two methods. These individual components were then composed to form the CGRA clusters and interconnect.

All of the physical design was done using a 65nm IBM process. Our initial work on the full custom models started with IBM's 10SF process node. These models were used for the experiments in Chapter 6. Standard cell libraries and memory compilers were released to the university much later and only for IBM's 10LP process, not the 10SF process. The 10LP process had lower leakage and thus lower static power than the 10SF process, but similar delay and dynamic energy consumption. Given the timing of experiments, we were unable to migrate the full custom models to the 10LP process for the experiments in Chapter 7. Therefore, results in Chapter 7 used the 10SF process for the individual muxes, registers, drivers, and interconnect components and the 10LP process for the SRAM arrays used in the large register files. The full custom models were migrated to the 10LP process for the experiments on specialization of the functional units and optimized Mosaic architecture in Chapters 8 and 9, respectively. Due to differences between the 10SF and the 10LP processes we expect that the final results of experiments in Chapters 6 and 7 would change if rerun, but that the relative differences between architectures would remain the same, and thus the conclusions would not change. This stability of the results is of particular concern for the experiments in Chapter 7, because they were run with physical models from both the 10SF and 10LP processes. These concerns are directly addressed in Section 7.3.2, but in summary only the long-term storage structures used the 10LP process, while all of the short-term storage structures used the 10SF process. Since our experimental methodology isolated

the evaluation of long-term storage from short-term storage, the results and optimization of each type of storage was performed with a consistent process technology, and thus are internally consistent. For this reason, we are confident that the relative performance and conclusions drawn from the experiments in Chapter 7 will remain unchanged.

Chapter 5

EVALUATION METHODOLOGY AND BENCHMARKS

This dissertation presents the results for three main experiments that explore and optimize architectural features for interconnect, storage, and specialized functional units of CGRAs. The goal of each experiment is to isolate the impact that a feature has on the energy efficiency, performance, and area of a CGRA design. This chapter describes the overarching processes and evaluation, while specific details about the interconnect, storage, and functional units are presented in Chapters 6, 7, and 8.

5.1 Benchmarks

As described previously in Chapter 3 the Mosaic architecture is designed as an accelerator for a host processor. As such, it is designed to execute an application's computationally expensive loops in a pipelined fashion, thus exploiting loop-level pipelining. To evaluate the effectiveness of different architectural features, the Mosaic research group has developed several applications with computationally-intensive inner loops and isolated computational kernels to form a benchmark suite. The benchmarks were selected to be representative of computation and communication patterns for the following application domains: streaming data, digital signal processing, and scientific computing. The isolated computational kernels are used to test that the CGRA performs well on key mathematical operations, transformations, or idioms from the application domain. Additionally, kernels provide a mechanism for comparing the performance of a CGRA to other spatial accelerators and are frequently found as the computationally-intensive inner loops in applications. The applications that were included in the benchmark suite are used to test system performance more broadly, including system I/O, irregular (or less regular) control flow, and more esoteric operations. Using a combination of isolated kernels and whole applications for architecture evaluation helps avoid overfitting an architectural feature to a single type of kernel or synthetic bench-

mark. The list of isolated kernels and applications used in the benchmark suite are given below, along with references for overviews of the algorithms that the kernel or application compute. Additionally each benchmark has a short description that indicated its structure, and some motivation for being included in the suite.

Kernels: isolated kernels that are standard algorithms within computer science and electrical engineering.

- Finite Impulse Response (FIR) filter - a feed forward algorithm that performs mostly multiplication and addition. It is commonplace in DSP applications.

- Matched filter [2] - an algorithm for matching hyperspectral data against one or more spectral signatures.

- 2D convolution - a kernel with multiple nested loops that is easily blocked or striped to balance computation versus local storage. It is a very common algorithm in DSP applications.

- Dense Matrix Multiplication [60] - provides a nice example of an algorithm that can be arranged such that computation scales with the square of the input size, while the communication scales linearly.

- CORDIC (COordinate Rotation DIgital Computer) [61] - a math library for performing trigonometric functions without the use of dedicated hardware functions or hardware multipliers.

Applications: these applications are both representative of their respective domains, and have computationally-intensive inner loops that are different than the isolated kernels in the suite.

- K-means clustering [62] - a dimensionality reduction algorithm that can be used for compressing and segmenting hyperspectral image data.

- Heuristic block-matching motion estimation [63] - used for video encoding and decoding, and contains a moderate amount of irregular control-flow and unpredictability that surround the execution of the computationally-intensive loops.

- Smith-Waterman approximate string matching [64] - an approximate string matching algorithm that uses dynamic programming to compare a query string versus a reference string. It is frequently used in computational biology.

- Positron Emission Tomography (PET) event detection [65] - a memory bound application that stresses the bandwidth to embedded memory and contains numerous, short, `if-then-else` blocks that are predicated.

The heuristic block-matching motion estimation application illustrates the need for spatial architectures to efficiently execute algorithms with irregular control-flow and unpredictability. This algorithm also illustrates a common design tradeoff, which is that an algorithm can be designed using a style that is: brute force, efficient, or fast. Brute force algorithms are typically simple, repetitive, and easy to accelerate. Efficient algorithms, such as fast Fourier transform (FFT)[1] and dynamic programming, use sophisticated algorithms to minimize data movement and computation. They still tend to be repetitive and predictable, but have more complex implementations. Fast algorithms use input-dependent heuristics to avoid computation, sacrificing some accuracy for speedup. Therefore, the control flow and data movement is less repetitive and predictable.

The naive version of block-matching motion estimation is extremely regular, and spatial accelerators can typically achieve several orders of magnitude speedup. However, heuristic variants of the algorithm can reduce the amount of computation by 95%-99% without significant loss in quality. Therefore, to be competitive with high quality sequential versions of the algorithm it is necessary to accelerate the more complicated heuristic variants of block-matching motion estimation.

---

[1] FFT was not included in the benchmark suite due to a lack of available programming resources.

5.2 Methodology

The focus of this research is on improving the energy efficiency of CGRAs without significant loss of performance. The emphasis on maintaining performance is a product of a CGRA's typical role as an accelerator. Most applications within the domains being explored favor high throughput over low latency, and for applications with pipelined loops the throughput is dominated by the loop's initiation interval (II) and the architecture's clock frequency. The recurrence II of a pipelined loop is the upper bound on the application's performance and is dictated by the number of cycles required to compute the longest loop-carried dependence, or 1 cycle if there is no dependence. In most of my experiments the tools were constrained to map the application to the architecture at (or near) the recurrence II, or to report a failure if that could not be achieved. For experiments in Chapters 6 and 7 applications were required to map at a consistent II. For reasons detailed in Chapter 8 the experiments on specializing the functional units could not be similarly constrained.

To understand the impact of the II on an application's performance it is necessary to examine the overall performance of an application, which is determined by architecture-independent and architecture-dependent components. When an application is mapped to a modulo-scheduled architecture, the architecture-independent performance is $P_A = II * N_L + L$; which is dictated by the mapping's initiation interval II, schedule length L (*i.e.* latency), and number of loop iterations N_L, as described in Sections 3.3 and 4.3. When the number of loop iterations to be performed is large compared to the latency, then the architecture-independent performance is dominated by the initiation interval. Long-running loops are common for the application domains of interest, and so the II of the modulo schedule is a reasonable proxy for architecture-independent performance. The architecture-dependent performance is $P = P_A * f$; it is dictated by the clock frequency f, which in turn is established by the critical path of the design. For many of the experiments that were conducted the architectural features being tested did not heavily influence the architecture's critical path. Therefore, while the overall performance of an application is $P = (II * N_L + L) * f$, the II remains a good approximation of overall performance.

This research uses a simulation-driven methodology for testing the efficacy of architec-

tural features in improving energy efficiency. We selected applications and computational kernels that are representative of an application domain to form a benchmark suite. Many of the computational kernels within these domains represent a family or class of computations and can be parameterized to form larger or smaller instances of the computation. Some examples of parametric kernels and applications are: FIR filters, dense matrix multiplication, or K-means clustering. A key parameter for a FIR filter is the number of taps in the filter, and changing the number of taps fundamentally changes the filter's response. However, as a representative of part of an application domain the number of taps is irrelevant, except that it determines the computational intensity of the FIR filter.

To create a flexible benchmark suite, each application is instrumented with tuning knobs, which are algorithmic parameters that can be adjusted to determine the size of the computation [66]. Tuning knobs are a special Macah language construct that allows the programmer to make an application parametric and specify a range of legal values for each tuning knob. Then the Macah compiler can auto-tune certain properties of the application, such as the computation to communication ratio for a kernel or a blocking factor of an inner loop, to maximize its throughput for a given number of architectural resources. The auto-tuning is accomplished by simulating the application while it is instrumented with sensors that allow the compiler to evaluate the quality of a particular application-to-architecture mapping. Specific settings for each application's tuning knobs are provided in the experimental chapters 6, 7, and 8.

The benchmarks are compiled and mapped onto families of CGRA architectures that only differ in a single architectural feature. These mappings are then simulated to collect area, energy, and performance measurements. Finally, the results for different architectures are compared to extract the influence that an architectural feature has on the overall architecture's statistics. A more detailed sequence of the testing steps is provided below. These steps are repeated for each application and kernel in the benchmark suite and for each instance in the architectural family.

1. Use an application's tuning knobs to customize the benchmark and to set the size of the application's dataflow graph.

2. Calculate the number of resources required by the application's dataflow graph, and the minimum recurrence initiation interval (II).

3. Construct a CGRA that has sufficient resources for the application to map at its recurrence II. For any critical resource, ensure that the architecture has 20% more resources than necessary to avoid over-constraining the CAD tools. (Note that 20% is a generally accepted guideline, and is used in the VPR [51] research CAD tool.) All other non-critical resources will be dictated by the architecture family's ratio between all physical resources.

4. Map the application to the architecture using multiple random seeds for the placement algorithm. For many of the experiments on interconnect and storage structures it is possible to share an application's placement between multiple instances of the architectural family, since the instances differ in features not considered by the placer; most of the interconnect and storage structures are only relevant to the routing algorithm and are ignored by the placement algorithm. By reusing the placement for multiple parts of a test we can reduce the variability in the CAD algorithms, thus reducing the amount of tool noise in the simulation results.

5. Simulate the mapped design on each architecture, using the configuration produced by the CAD tools. During simulation, record bit- and structure-level activity of the relevant parts of the CGRA. Combine these activity numbers with the simulated runtime of the application, the energy models that characterize the individual devices, and the components to compute the total energy per benchmark.

6. Collect the metrics from the simulation and the results of mapping the application to the architecture. The most important metrics are:

    - Actual II achieved when mapping the application to the architecture, and intrinsic recurrence II of the application
    - Schedule length (*i.e* latency) of the mapped application

- Total simulated runtime
- Total simulated energy consumption
- Area of the CGRA
- Clock frequency of the CGRA

7. Select the instance of the application where the random placement seed produced the best result. The quality of the result is judged by selecting the instance with the lowest II, then total energy, then total area. The selection criteria is designed to favor improvements in performance and energy over area. Recall that II is being used for performance because latency is a secondary contributing factor for long-running jobs, and that the architectures are running at a comparable clock frequency.

After following the above steps for each application, and for each instance that is being tested within the architectural family, we have a characterization of the architectural design space with respect to a single feature. The use of multiple benchmarks provides multiple data points that are then averaged across all benchmarks for each individual architecture within the architectural family. Using the average of the benchmark suite it is possible to quantify and compare the impact of the architectural feature under test on the architecture's reported statistics (*e.g* area, delay, and energy) and predict the architecture's performance for the entire application domain(s).

5.2.1 Constructing an architecture family

One key step in this methodology was to create a CGRA architecture template that could vary the quantity of an individual feature without perturbing the structure or resource allocations for the rest of the design (*i.e.* in isolation). Each instance of the template was then evaluated by running a suite of benchmark applications and collecting power and performance numbers. Varying the presence or the quantity of resources dedicated to each feature demonstrated how a given feature contributed to the architecture's energy efficiency and performance. This entire process was then repeated for each feature tested. The class of Mosaic CGRAs, which was described in Chapter 3, defines the family of CGRA architectures

explored in this work, and serves as the starting point for the sequence of experiments. The broadly defined Mosaic CGRA family of Chapter 3 is refined into a baseline architecture for each experiment by filling in the unexplored aspects of the architecture with a reasonable set of resources and simplifying assumptions. The goal of the baseline was to provide a context within which each feature was tested; then throughout the experiments in Chapters 6, 7, 8, the architecture is refined as features are tested and optimizations are identified.

To facilitate isolation of architectural features, and simplify the experimental methodology, aspects of the architecture not being tested were carefully constructed to avoid interfering with the feature under test. Typically this meant that some resources were designed to be unconstrained or idealized, to highlight the behavior of the primary features under test. One example of this is the crossbar in the compute cluster that ties the functional units and support logic together. When testing the effects of wider or narrower interconnect channels in Chapter 6 the cluster crossbar was designed so that it was not a bottleneck. Paring down from a full to a depopulated or partial crossbar would throttle the bandwidth inside of the cluster, which is not part of the architecture feature being tested, and would perturb the results in unforeseen ways. Another example are the functional units used in the experiments from Chapters 6 and 7. Since these experiments were focused on the interconnect and storage system, respectively, they used idealized functional units that executed all operations in a single cycle to minimize the effect that operation placement would have on the experimental results.

5.3 Overview of Experiments

Chapters 6, 7, and 8 provide the detailed experimental setup, results, and analysis for architectural features for the interconnect, storage, and functional units, respectively. The subsequent paragraphs give a brief overview and motivation for each experiment.

5.3.1 Optimizing Communication

A hallmark of CGRAs and other spatial architectures are communication topologies and resource distributions that provide a scalable amount of aggregate bandwidth and bandwidth per computing element. This scalability enables designers to efficiently add communication

resources to keep pace with the increasing number of computing resources. Examples of communication structures with both scalable aggregate and per-node bandwidth are global interconnects that form a grid, mesh, tree, rings, or segmented tracks. The arrangement of compute resources for these types of interconnects typically use 2-D or 1-D repeating patterns. Throughout this work we have used a 2-D grid interconnect because its layout and characteristics correlates well with the physical limitations of current silicon CMOS design processes, and is a common design for spatial architectures. The focus on the interconnect stems from the observation that it consumes a significant portion of a spatial processor's energy; Poon et al. [21] show that an FPGA's interconnect dissipates 50% to 60% of its energy.

In a grid topology each processing element or cluster is attached to the interconnect at regular spacings, and the interconnect's resources are independent of the processing element. This partitioning makes it easy to vary the number of channels in the interconnect with respect to the number of channels between each processing element (or cluster) and the interconnect. Specifically, the number of switchbox-to-switchbox channels is independent of the number of cluster-to-switchbox channels. Therefore, as applications become larger and spread farther across the fabric, the interconnect can scale to provide additional capacity without affecting the capacity to any single PE. For example, a grid interconnect could have 12 channels between switchboxes and only 8 channels connecting the cluster to the switchbox. This flexibility allows the architect to maintain a constant bandwidth to and from the cluster while scaling up the aggregate chip bandwidth.

Given that the functional units within the fabric are time-multiplexed, the primary design decision for the interconnect, aside from topology, is the nature of the interconnect channel. For each channel, are the multiplexers in the switchboxes of the grid statically configured, time multiplexed, or a combination of both? Furthermore, do the advantages or disadvantages of each type of interconnect channel depend on the maximum initiation interval supported by the architecture, or the ability of the SPR tool to share a statically configured channel between multiple signals while not changing the communication pattern? These questions, as well as the value of adding a single-bit control network to augment the word-wide datapath, are evaluated in Chapter 6.

5.3.2 Optimizing Storage

Efficiently managing values between their production and consumption is one of the more challenging aspects of designing a CGRA. The challenge arises because producers and consumers can be spread throughout the CGRA's fabric, which leads spatial architecture to favor distributed storage over centralized structures. A further challenge is that the cyclic execution of a modulo schedule limits the effectiveness of traditional register files, which are common in past CGRA research projects. Chapter 7 presents an exploration of different storage structures and their distribution throughout the CGRA fabric.

5.3.3 Specializing the Functional Units

Applications have a diverse mix of operators, ranging from simple shift or arithmetic and logic operations, to multiplication and fused multiply-add operations, and even datapath select / multiplexer operations. Simple CGRA fabric designs could use a homogenous set of universal functional units that perform all operations; however, this approach ends up over-provisioning the functional units by providing the resources required for expensive and rarely used operators. Specializing functional units will reduce their area and energy costs when compared to a universal functional unit, but can introduce resource bottlenecks both locally (within a cluster) and globally. The functional units forming these bottlenecks are referred to as critical resources, and typically degrade the quality of an application's mapping by decreasing throughput or increasing latency and static energy consumption. Two examples of when these types of resource bottlenecks can occur are where there are only a limited number of multipliers within a cluster, or there is only one type of device that supports both the multiply and shift operation. The impact of scheduling operations on a critical resource manifests differently based on the interaction between the application and the resource bottleneck. For example, an architecture with a single multiplication unit per cluster can exhibit both local and global bottlenecks. If an application had two multiplications that execute in parallel, and the results of both feed into a pair of cross-coupled critical loop-carried dependences, then the single multiplier per cluster creates a local bottleneck, as those multiplies would either have to be sequenced or spread out. Either solution would

add latency to at least one of the critical loops and thus increase the application's mapped II. Alternatively, if an application had many multiplications and very few other operations the single multiplier per cluster would be a global bottleneck; the application would require many clusters, but each cluster would be quite poorly utilized. In general, local bottlenecks can force a cluster of operations in the application's dataflow graph to artificially spread out physically (across clusters in the CGRA) or in time, either of which can increase the mapped II. Meanwhile, global bottlenecks typically force the application to require a larger CGRA fabric, but leave a large part of that fabric underutilized. These tradeoffs are examined in Chapter 8.

5.4 Related Work

Few other research efforts have evaluated the energy advantages and tradeoffs of architectural features in CGRAs. By and large these efforts have focused on holistic system analysis that examine multiple architecture features simultaneously. The disadvantage of this approach is that it is difficult to isolate the impact of a single architectural feature on the architecture's overall metrics, thus making it hard to integrate features from multiple projects into a single architecture. The most notable example of previous architecture explorations was the work of Bouwens et al. [67, 68], which explores the effects of interconnect topology and connectivity for the ADRES architecture. Their approach is similar in many respects to ours, using simulator driven activity analysis, with the exception that their results do not isolate particular architectural features.

Several efforts did isolate the impact of specific architectural features, but only examined area and delay metrics. Work by Sharma et al. [47] explored the effect on area and delay of input versus output registering for functional units, and the design and connection patterns of segmented buses within the RaPiD architecture. Kim et al. [69] looked at the tradeoffs for what they call primitive versus critical functional unit resources in a MorphoSys-style CGRA. Their study explored the area and delay tradeoffs involved with limiting the number of multipliers in their architecture. They found that pipelining and sharing multipliers with other processing elements substantially reduced the area and delay of the architecture. These results align well with the results presented in Chapter 8, although our results started

out with a pipelined multiplier and were conducted on a much larger scale than the experiments in [69]. Wilton et al. [70] explore how much connectivity is required for point-to-point interconnects in an ADRES architecture with a heterogenous mix of functional units, when optimizing for area and performance. Others, such as [71, 69, 72], were smaller evaluations that confirmed benefits from state of the art design practices such as: pipelining functional units, sharing critical and expensive resources, segmenting long and shared bus structures, and clustering multiple functional units together.

As discussed in Section 4.6, both the Wattch project [56] and the work by Poon et al. [21] provided tools and studies for microprocessors and FPGAs, respectively. These works were used as guidelines for the methodology used in this dissertation. Looking further afield, Nageldinger [27] briefly explores the design space of CGRAs but mainly focuses on the development of the Xplorer framework for the KressArray architecture, which provides design guidance using approximate reasoning. Furthermore, the brief amount of exploration that was conducted by this work focused on the architecture's functionality (*e.g* interconnect topologies, compound functional units, and dual-output functional units) without considering the impact on of performance, area, or energy metrics.

5.5 Customizing the benchmarks for the experiments

The benchmarks were written in the Macah language and use tuning knobs to scale the size of their computations and their resulting dataflow graphs. For the experiments in this dissertation we have tuned the benchmarks so that they run through the Mosaic toolchain, particularly the place and route phases of SPR, in a reasonable amount of time but are representative of real-world problem sizes. For each experiment, a datapoint is generated by mapping an application to an architecture and simulating that mapping in Verilog. We have limited the size of the applications, architectures, and the length of the simulation so that the combined time to map an application to an architecture and simulated it takes at least several CPU-hours, but no more than a few CPU-days, per datapoint. To achieve this bound, we have to limit the size of the application's dataflow graph (DFG) to several thousand routes and between several hundred to one thousand nodes. Within this range, SPR's place and route stages rarely take more than 30 hours. Verilog simulator runtimes

depend on the number of test vectors, but take between several hours and several tens of hours. Once simulation-specific nodes are removed, dataflow graphs of this size translate into applications that contain several hundred operations and target architectures with 64 to ∼200 functional units.

In addition to the tuning knobs, which can scale the size of an algorithm, the following factors determine how many operators and resources are required: compiler version, algorithm variation, and techmapper version and settings. As a result, different experiments were run using slightly different variations of the benchmark suite. While the details of an application's dataflow graph changes between experiments, the overall structure and nature of the applications remained the same. Furthermore, for all of the tests within each experiment (as delineated by chapters) the toolchain and application versions were kept consistent.

The quality and sophistication of the Macah language and compiler have changed dramatically during the course of this research. Initial versions of the Macah compiler were unable to flatten nested or sequenced loops efficiently, so algorithms were hand flattened to achieve efficient implementations. Current versions of the Macah compiler allow for sophisticated loop flattening, with programmer annotations that help guide the compiler to an efficient implementation. As a result of these improvements in compiler technology applications have been rewritten to use more natural loop structures and control flow. Additionally, in general, we have been able to add more tuning knobs to customize the balance of computation to communication within an application.

Algorithmic variations are another factor that can produce very different dataflow graphs for the same kernel or application. One example of an algorithmic variation is demonstrated in the FIR filter kernel. One of the two main implementations scalarizes the data structures that hold the tap coefficients and old input values. This approach is very easy to pipeline, but requires a large amount of local register storage for all of the scalarized variables. The other approach used by us for implementing the FIR filter is to put tap coefficients and old input values in banks of embedded memory. This allows the application to scale up the number of taps supported, but requires time-multiplexed access to the embedded memories. As a result of the noticeable differences in computation, communication, and

storage patterns, both of these variants of the FIR filter are included in the benchmark suite as individual benchmarks.

The last factor that really changes the application's dataflow graph is the Mosaic techmapper that executes between the Macah compiler and SPR. The techmapper, as described in Section 4.2, is capable of folding constant values into an immediate field and fusing back-to-back multiply-add operation sequences or packing logical operations onto 3-LUTs. The inclusion of immediate values in the functional unit's operand code reflects an increase in the size of the CGRA's configuration memories. The impact of the additional configuration memory is accounted for during the evaluation of the functional units in Chapter 8, but does not directly influence the experimental results since all of the architectures in both Chapters 7 and 8 supported this feature.

As mentioned, these variations between versions of the tools and variations of the algorithms can lead to differences in the number of operations or constant values in the application's dataflow graph (*i.e.* its size). The details about algorithm variations and tool versions are provided in each experimental chapter. Additionally a summary of the number of operations and resources used, as well as the minimum recurrence initiation interval, are provided for each benchmark.

Chapter 6

INTERCONNECT

This chapter addresses the question of how flexible the interconnect should be in CGRAs. We focus on the interconnect first because it accounts for a large portion of both the area and energy consumption in spatial architectures. At one extreme, FPGA-like architectures require that interconnect resources be configured in exactly one way for the entire run of an application. At the other extreme are architectures that allow every interconnect resource to be configured differently on a cycle-by-cycle basis. In this chapter we investigate this tradeoff, comparing architectures at both extremes, as well as mixtures that combine each of these styles of resources.

The optimal tradeoff between scheduled and static resources will likely depend on the word-width of the interconnect, since the overheads associated with some techniques may be much larger in a 1-bit interconnect than with a 32-bit interconnect. To explore this possibility we consider the area, power, and channel width at different word-widths and maximum hardware supported initiation interval (II), as well as investigating hybrid bitwidth architectures that have a mixture of single-bit and multi-bit interconnect resources.

6.1 Baseline Mosaic CGRA Architecture

Chapter 3 presented the Mosaic CGRA architecture, which defined the class of coarse-grained reconfigurable arrays examined in this dissertation. To establish the baseline architecture for the family, it was necessary to fill in the aspects of the Mosaic CGRA that were left unspecified in Chapter 3. In particular, the contents of the cluster and the intra-cluster communication mechanism require definition. The block diagram of the CGRA fabric is shown in Figure 6.1. For these experiments we have focused on the interconnect that allows the clusters to communicate. Spatial arrangements for processing elements (PEs) and clusters are most commonly meshes or grids [73, 24, 19], which we explore; other patterns,

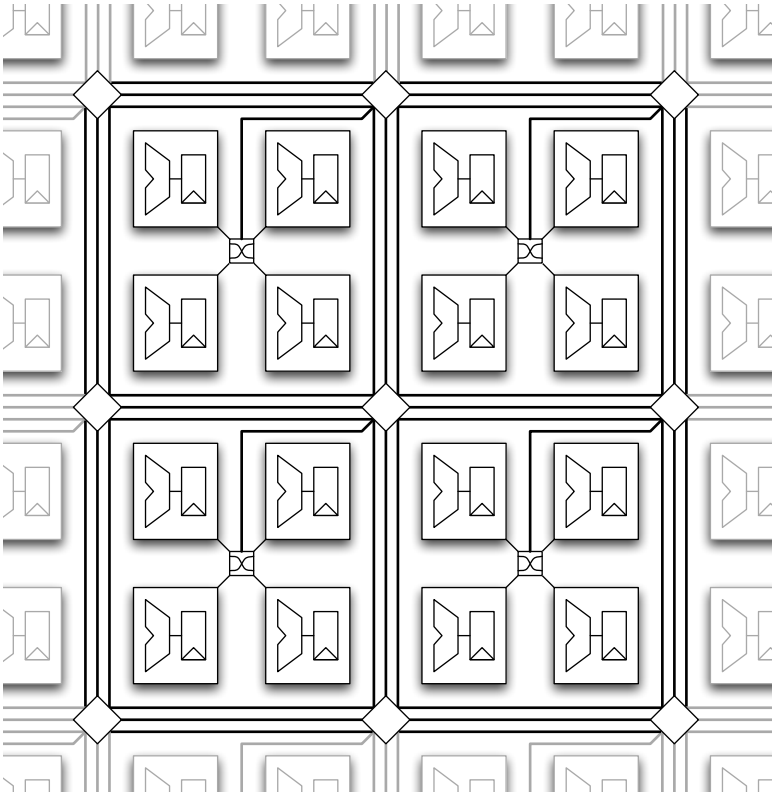
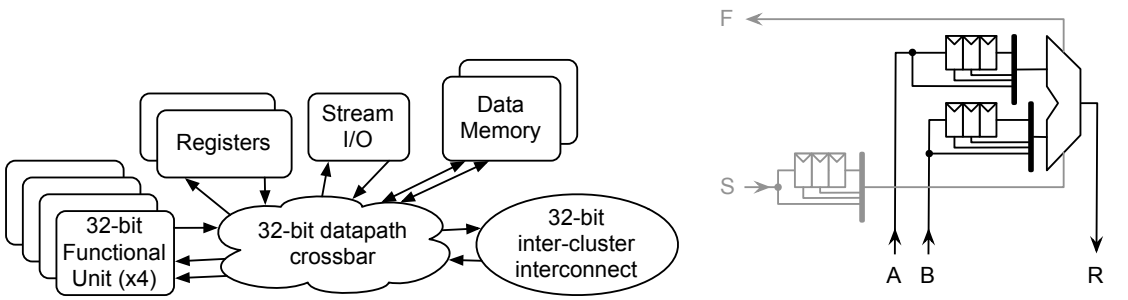


Figure 6.1: CGRA Block Diagram - Clusters of 4 PEs connected via a grid of switchboxes.

such as linear arrays [74] and fat-pyramids [22], have also been studied. As a starting point for these experiments, the clusters are arranged in a grid and composed of 4 processing elements with idealized functional units, data memories, registers, and stream I/O. Components within the cluster are interconnected via a crossbar that is time-multiplexed. Block diagrams of the cluster and processing element are shown in Figure 6.2, with greater detail provided in Section 6.5.1.

6.2 Related Work

Recent FPGA architectures include coarse-grained components, such as multipliers, DSP blocks and embedded memories. This trend blurs the line between FPGAs and CGRAs, but even the most extreme commercial architectures still devote most logic resources to



(a) Baseline Cluster

(b) Baseline PE w/Functional Unit

Figure 6.2: Baseline cluster and processing element with idealized functional units and 3 registers for input retiming.

single bit components, and to our knowledge no current commercial FPGA has word-wide interconnect or a scheduled (time-multiplexed) configuration system.

The interconnects of some early CGRA architectures, such as RaPiD [74] and Matrix [25], combined the concepts of static configuration and scheduled switching to some extent. However, in neither project was there a systematic study of the optimal mixture of the two kinds of resources.

The ADRES [73] and DRESC [75] projects are very similar to the Mosaic project. They provide architecture exploration and CGRA-specific CAD algorithms, but ADRES studies were limited to time-multiplexed interconnect in much smaller CGRA fabrics and primarily focused on the integration between the CGRA fabric and a VLIW host processor. In [76], the area and energy tradeoffs between various interconnect topologies are studied. The experiments presented here instead focus on the balance of scheduled and static resources, rather than topology.

6.3 Benchmarks

The benchmarks used for the tests in this experiment were introduced in Section 5.1. Table 6.1 provides the characteristics and sizes of the benchmarks as they were used for this chapter, and Table 6.2 lists the tuning knob settings for each benchmark. As described in Chapters 4 and 5, we have carefully constructed inner loops in Macah so that each applica-

Table 6.1: Benchmark Applications and Simulated Architectures

Application	Min II	# DFG operations				# PE clusters	CGRA Grid Size
		ALU	Memory Accesses	Stream IO	LiveIns		
64-tap FIR filter	2	199	0	1	195	64	10x10
240-tap FIR filter (Banked Mem)	6	284	48	1	177	30	7x8
2D convolution	4	180	6	1	189	30	7x8
8x8 Matrix multiplication	4	361	0	64	218	36	8x8
8x8 Matrix mult. (Shared Streams)	9	402	0	16	224	16	6x6
K-means clustering (K=32)	7	525	32	33	189	25	7x7
Matched filter	4	193	30	13	88	20	6x7
Smith-Waterman	5	342	11	5	109	25	7x7
CORDIC	2	178	0	5	46	30	7x8
8×8 Motion estimation [63]	5	465	16	9	189	30	7x8

Application	Tuning Knobs
64-tap FIR filter	Num. Coefficients = 64
240-tap FIR filter (Banked Mem)	Num. Coefficients = 240, Num. Memory Banks = 24
2D convolution	Kernel Size = 7x7, Vertical Stripe Width = 15
8x8 Blocked Matrix multiplication	Blocking Factor = 8x8
8x8 Blocked Matrix mult. (Shared Streams)	Blocking Factor = 8x8
K-means clustering (K=32)	Num. Clusters = 32
Matched filter	Num. Parallel Channels Processed (Block Size) = 2, Num. Signatures = 13, Num. Channels = 16
Smith-Waterman	Stripe Width = 11
CORDIC	Num. CORDIC iterations = 3
8×8 Motion estimation [63]	Image Block Size = 8x8, Num. Parallel Searches = 8

Table 6.2: Tuning Knob settings for Benchmark Applications

tion has an interesting kernel with significant parallelism, which were then mapped through the Mosaic toolchain. The inner loops are software pipelined [28], and the Min II column indicates the minimum number of clock cycles between starting consecutive loop iterations, which is determined by the largest loop-carried dependence. LiveIns is the number of scalar constants loaded into the kernel. Four of the benchmarks used for this experiment represent two permutations of the same key kernels: the FIR filter and dense matrix multiplication. The top two benchmarks listed in Table 6.1 are variants of a FIR filter. The first of these is the most straight-forward implementation with multiplications and tap coefficients spread through the array. The second variant is a larger, banked FIR filter where the tap coefficients are split into banks, stored in memories, and sequenced through during execution. The fourth and fifth benchmarks are variants of a blocked dense matrix multiplication; the first uses one input stream per element of the blocking window, while the second version shares input streams and has one per row and column.

6.4 Tools

As presented in Chapter 4, the Mosaic toolchain is able to rapidly explore a variety of architectures by programmatically composing Verilog primitives to fully specify an architecture instance. The resulting composition is denoted a datapath graph and contains all information needed to perform placement, routing, and power-aware simulation. Note that for this experiment the techmapper described in Chapter 4 was not available yet, and thus there was no optimizations performed on the constant values in the benchmark suite. As a result, there were more producers and consumers within the dataflow graph, but there was no significant impact on the distribution of value's lifetime, or communication patterns.

The compiled Macah program is a dataflow graph representing the computation that will be mapped to a particular hardware instance. Mosaic maps the dataflow graph onto the CGRA's datapath graph with the SPR CGRA mapping tool [48]. A key feature of the datapath graph representation and SPR's routing tool is that the architecture's routing resources are represented via multiplexers rather than wires. By focusing on the control resource (*i.e.* the multiplexer), rather than the wire, the Pathfinder algorithm is able to negotiate the input-to-output mappings of the routing muxes, as illustrated in Figure 3.4

and detailed in [48]. This feature also allowed multiple values to share statically configured interconnect channels in different clock cycles, by negotiating to have the multiplexers statically configured to a single mutually compatible input-to-output mapping. Furthermore, this approach enabled SPR to handle a mixture of time-multiplexed and statically configured resources, which allow us to evaluate our different architectures. The resulting CGRA configuration is then executed in a Verilog simulator.

6.4.1 Circuit Area and Energy Modeling

Recognizing that the energy consumption of a logic structure can be highly dependent on the input data sequence [57], we use simulation-driven power modeling, as discussed in Section 4.6. We characterize the fundamental interconnect circuits from full-custom layouts in an IBM 65nm 10SF process. With this approach, we provide realistic values for a particular process and, more importantly, comparable results for different permutations of the interconnect, allowing us to assess architectural decisions.

We use the Verilog PLI to account for energy consumption as the simulation progresses. The values accumulated are derived from our transistor-level circuit models. The high level of detail for these simulations requires long runtimes. Currently, simulating a large highly utilized architecture for a few tens of thousands of clock cycles can take as long as 20 hours. To reduce this burden we scale the input sets down to small, but non-trivial, sizes. Applications that are well-suited to CGRAs tend to be regular enough that the behavior of a kernel on a small set of inputs should be representative of the behavior on much larger set of inputs.

6.5 Experimental Setup

The channel width of our CGRA, like an FPGA, is the number of independent communication channel pairs between clusters of PEs. Thus, on a 32-bit architecture one channel (or "track") contains 32 wires, and a channel pair has two unidirectional tracks, oriented in opposite directions. A primary goal of this study is to understand the tradeoff between required minimum channel width and the flexibility of each channel (is it statically configured or scheduled). A statically configured channel is like an FPGA interconnect track; it

is configured once for a given application. A scheduled channel changes its communication pattern from cycle to cycle of execution, iterating through its schedule of contexts within a configuration. In our architectures the scheduled channels use a modulo-counter and a cyclic schedule.

To explore the benefits versus overhead of these scheduled interconnect channels, we mapped and simulated the benchmarks described in Table 6.1 to a family of CGRAs. These architectures vary in the ratio of scheduled to statically configured interconnect channels.

6.5.1 Architecture Setup

For this study we explore clustered architectures with a grid topology. Within each CGRA family there are a common set of compute (or PE), IO, and control clusters. The clusters are arranged in a grid, with PE clusters in the center and IO and control clusters on the outer edge. Computing and interconnect resources within the cluster were scheduled, and all scheduled resources supported a maximum II of 16 unless otherwise stated.

The datapath of the PE compute cluster contains 4 arithmetic and logic units (ALUs), 2 memories, 2 sets of loadable registers for live-in scalar values, and additional distributed registers. The control path contains 2 look-up tables (LUTs), 2 flip-flops for Boolean live-in values, and additional distributed flip-flops. Given our focus on the interconnect, the PE cluster is simplified by assuming that the ALU is capable of all C-based arithmetic and logic operations, including multiplication, division, and data dependent multiplexing. The components in the cluster are connected to each other and to the global interconnect switchbox via a scheduled crossbar. Refining the intra-cluster storage and logic resources will be the subject of Chapters 7 and 8.

The IO clusters are designed solely to feed data to and from the computing fabric, offloading any necessary computation to adjacent compute clusters. Each IO cluster contains 4 stream-in and 4 stream-out ports, one set of loadable registers, a set of word-wide and single-bit scan-able registers than can transfer live-out variables to a host processor, and distributed registers for retiming. As with the PE cluster, all components are connected via a scheduled crossbar that also provides access to the associated switchbox. Four of

the peripheral clusters are control clusters, which are the superset of a PE + IO cluster combined with auxiliary logic that provides loop control.

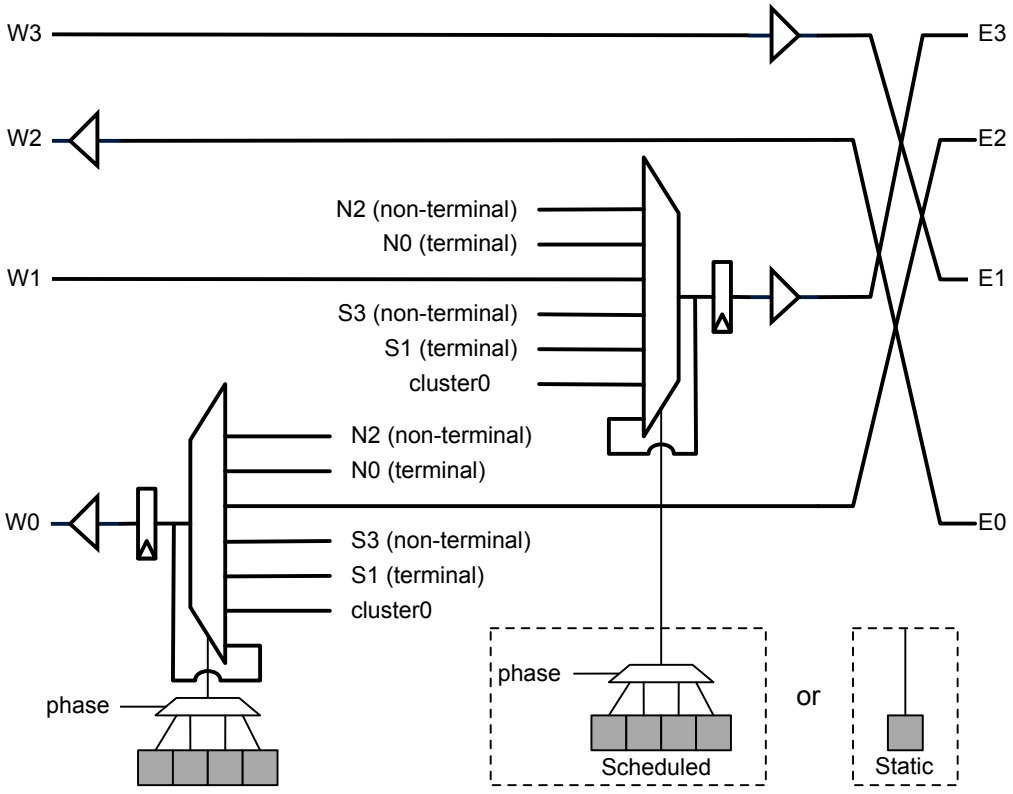


Figure 6.3: Block diagram of a horizontal slice of a switchbox with channel span 2. Shaded gray boxes represent configuration SRAM.

Each cluster is connected to a switchbox, which are themselves connected to form the global cluster-to-cluster grid. Communication between clusters and switchboxes, and between switchboxes, use single driver, unidirectional channel pairs [77]. The switchbox topology follows a trackgraph-style pattern. All terminal (*i.e.* sourced) tracks are registered leaving the switchbox. Additionally, connections that go into the cluster, or between horizontal and vertical channels, are registered leaving the switchbox. All interconnect channels had a span of 2 – they were registered in every other switchbox. The switchbox layout is shown

in Figure 6.3, based on Figure 5 from [77]. The statically configured channels replace the configuration SRAM bits shown in the bottom of Figure 6.3 with a single SRAM bit.

In order to determine the length of wires in the global interconnect, we use a conservative estimate of $0.5mm^2$ per cluster, based loosely on a CU-RU pair in the Ambric architecture [13]. The area consumed by the switchbox is computed from our circuit models.

6.5.2 Benchmark Sizing

Our benchmarks were parameterized with tuning knobs to adjust the degree of parallelism, the amount of local buffering, etc. Each parameter was then adjusted to a reasonable value that balanced application size versus time to place and route the design and time to simulate the application on an architecture. The circuit was mapped to the smallest near-square (NxN or NxN-1) architecture such that no more than 80% of the ALUs, memories, streams and live-ins were used at the min II of the application. The resulting compilation and simulation times for applications were 30 minutes to 20 hours, depending on the complexity of the application and the size of the architecture.

6.5.3 VLSI Circuit Modeling and Simulation

Our VLSI layouts of components used to build the interconnect topology provide realistic characterization data for our study. Each component provides area, delay, load, slew rate, inertial delay, static power, and dynamic energy for modeling purposes. Area and static power estimates are based on a sum of the values for the individual components required. The delay for the longest path between registers is used to estimate a clock period for the device. Registers that are unused in the interconnect are clock-gated to mitigate their load on the clock tree: statically for all word-wide static channels and single-bit channels, and on a cycle-by-cycle basis for word-wide scheduled channels.

For simulation of the 8-, 16- and 24-bit interconnect, the architecture was regenerated with these settings. However, our ALU models remained 32-bit, and we did not rewrite the benchmarks for correct functionality with 8-, 16-, or 24-bit widths. Instead, we only recorded transitions on the appropriate number of bits, masking off the energy of the upper

bits in the interconnect. This approximated the interconnect power for applications written for narrower bit widths.

6.5.4 Static versus Scheduled Channels

Scheduled interconnect channels can be reconfigured on a cycle-by-cycle basis, which carries a substantial cost in terms of configuration storage and management circuitry. Static configuration clearly offers less flexibility, but at a lower hardware cost. We explore the effects of different mixtures of scheduled and static channels by setting the scheduled-to-static ratio and then finding the minimum channel width needed to map a benchmark. For example, with a 70%/30% scheduled-to-static split, if the smallest channel-width an application mapped to was 10 channels, 7 would be scheduled and 3 static.

We measure the area needed to implement static and scheduled channels, and the energy dissipated by those circuits during the execution of each benchmark. At a given channel width, lower scheduled-to-static ratios are better area-wise. However, as we decrease the scheduled-to-static ratio we expect that the number of tracks required will increase, thus increasing the area and energy costs. Note that our placement and routing tools [48] support static-sharing, which attempts to map multiple signals onto static tracks in different phases of the schedule; the signals have to configure the track in the same way so that a single configuration works for all signals that share a track. This means that if multiple signals share a track, that track must be configured to reach the superset of all destinations of all sharing signals.

6.6 Results and Analysis

We divide our results into four sections, each focusing on a different issue in the interconnect. We start with a 32-bit interconnect and adjust the scheduled-to-static channel ratio. Next we vary the interconnect width down to 8 bits. From there we look at the impact of the maximum II supported by the hardware. The last section explores the addition of dedicated single-bit interconnect resources for control signals.

To generate the data, we ran each benchmark with three random placer seeds and used the result that gave the best throughput (lowest achieved II), followed by smallest area and

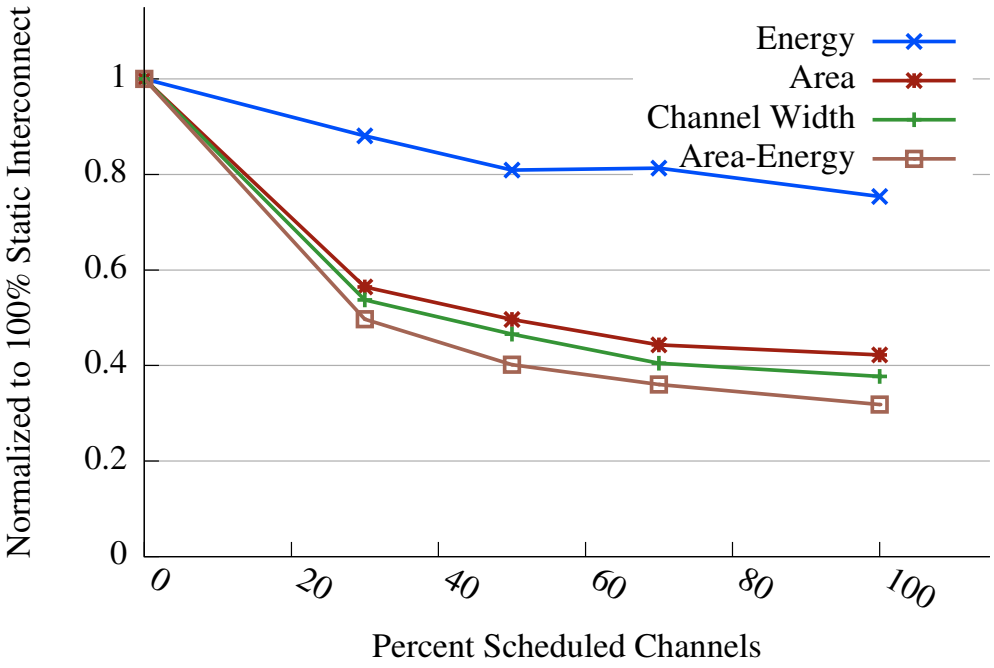


Figure 6.4: Interconnect area, channel width, energy, and area-energy metrics as a 32-bit interconnect becomes more scheduled.

then lowest energy consumption. By using placements with the same II across the entire sweep the performance is independent of the scheduled-to-static channel ratio. The data was then averaged across all benchmarks.

6.6.1 Interconnect Scheduled/Static Ratio

Intuitively, varying the ratio of scheduled-to-static channels is a resource balancing issue of fewer "complex" channels versus more "simple" channels. Depending on the behavior of the applications, we expect that there will be a sweet spot for the scheduled-to-static ratio. Figure 6.4 summarizes the results for channel width, area, energy, and area-energy product when varying the ratio of scheduled-to-static channels in the interconnect. Each point is the average across our benchmark suite for the given percentage of scheduled channels. The data is for 32-bit interconnects and hardware support for II up to 16. All results are

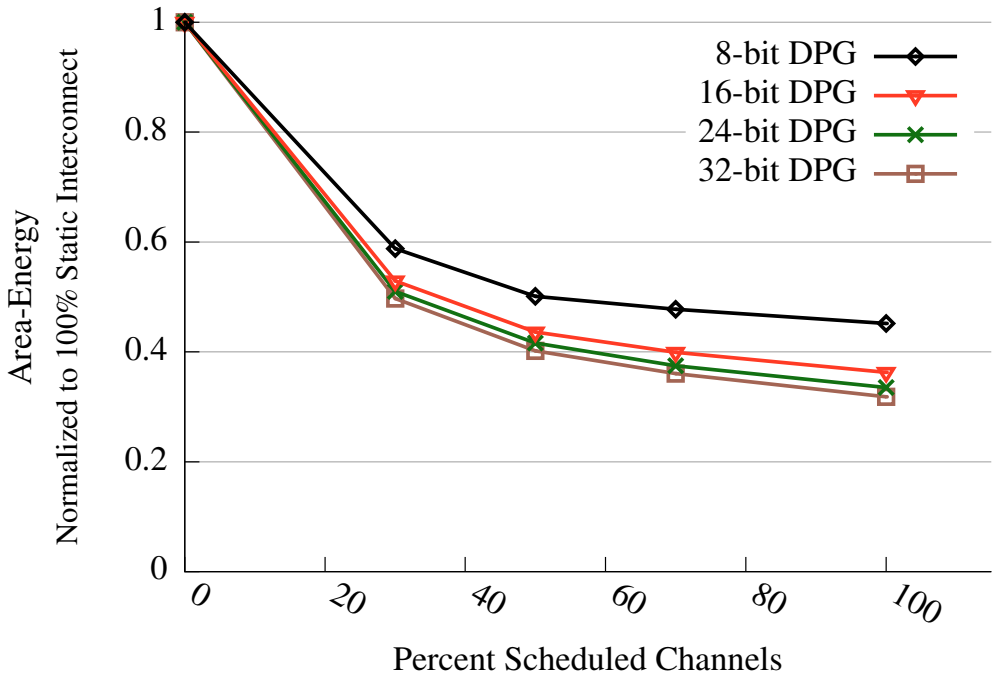


Figure 6.5: Area-energy product for different datapath word-widths, as the interconnect becomes more scheduled.

normalized to the fully static (0% scheduled) case. We observe improvements in all metrics as the interconnect moves away from static configuration all the way to 100% scheduled. By making the interconnect more flexible, the number of required channels is reduced to 0.38×, which translates into 0.42× area and 0.75× energy consumption. This is despite the area and energy overhead required to provide the configurations to these scheduled channels. Overall, the area-energy product is reduced to 0.32× that of the fully static interconnect.

6.6.2 Datapath Word-Width and Scheduled/Static Ratio

For a 32-bit interconnect, Figure 6.4 shows that having entirely scheduled channels is beneficial. However, for lower bitwidths, the overhead of scheduled channels is a greater factor. Figure 6.5 shows the area-energy trends for architectures with different word-widths. We observe that fully scheduled is best for all measured bitwidths, but as the datapath narrows

from 32 bits down to 8 bits the advantage of a fully scheduled interconnect is reduced. However, it is still a dramatic improvement over the fully static baseline. Our previous result for a 32-bit interconnect showed a 0.32× reduction in area-energy product. With the narrower interconnect, we see reductions to 0.34×, 0.36×, and 0.45× the area-energy product of the fully static baseline for 24-, 16- and 8-bit architectures respectively.

To explain this trend, we first look in more detail at the energy consumed when executing the benchmarks, followed by the area breakdown of the interconnect. Figure 6.6 shows the energy vs scheduled-to-static ratio in the categories:

- **signal** - driving data through a device or along a wire
- **cfg** - reconfiguring dynamic multiplexors and static and dynamic energy in the configuration SRAM
- **clk** - clocking configured registers
- **static** - static leakage (excluding configuration SRAM)

There are two interesting behaviors to note. First, the static channels have less energy overhead in the configuration logic / control system. So ignoring other effects, if two CGRAs have the same number of channels then the energy consumed will go down as the percentage of static channels is increased. However, the overhead of a scheduled channel is small at a datapath width of 32-bits, and so the energy reduction is small. The percentage of the energy budget consumed by configuration overhead does increase dramatically at narrower datapath widths.

The second behavior is a non-obvious side-effect of sharing static channels, as detailed in [48]. When multiple signals share a portion of a static interconnect they form a large fanout tree that is active in each phase of execution. As a result, each individual signal is driving a larger load than necessary, and so the additional dynamic energy consumed outweighs the energy saved due to reduced overhead. This behavior is one of the contributing factors that forces the signal energy to go up as the interconnect becomes more static.

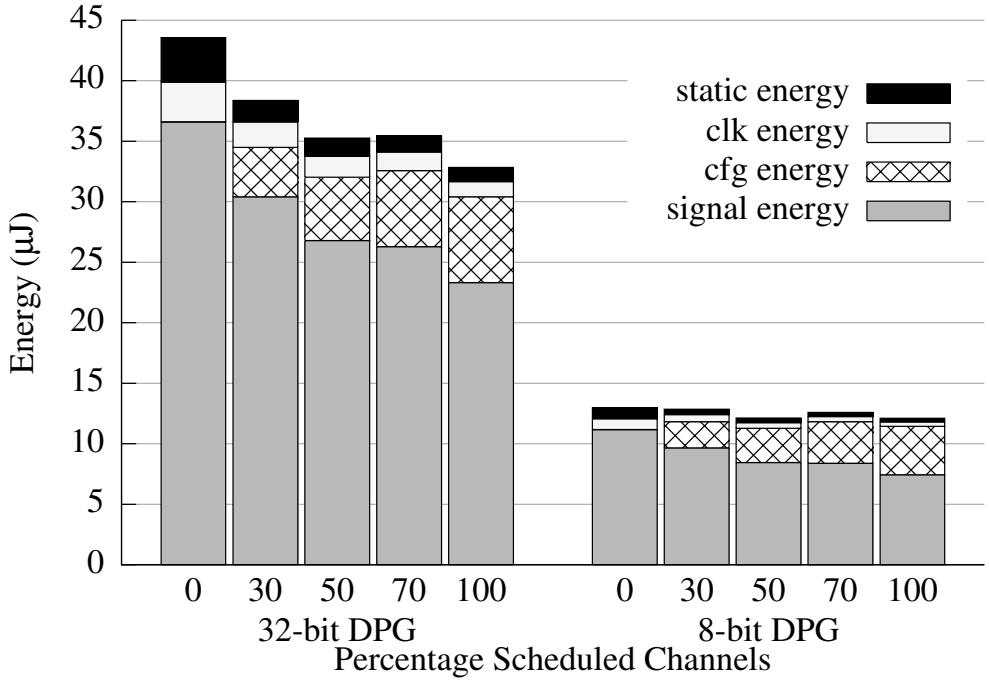


Figure 6.6: Average energy for global routing resources.

Figure 6.7 details the breakdown of the interconnect area vs. the scheduled-to-static ratio. As with energy overhead, we observe that the configuration system's area is tiny for fully-static systems, and a small portion of the fully-scheduled 32-bit interconnect. However, at narrower datapath widths, the area overhead of the configuration system accounts for a non-trivial portion of the interconnect, up to 19% for the fully scheduled case. All other areas are directly dependent on the channel width of the architecture, and dominate the configuration area. As channels become more scheduled, the added flexibility makes each channel more useful, reducing the number of channels required.

In summary, Figures 6.5, 6.6, and 6.7 show that the overhead required to implement a fully scheduled interconnect is reasonably small. However, as the bitwidth of the datapath narrows, those overheads become more significant. At bitwidths below 8-bit a static interconnect will likely be the most energy-efficient, though a scheduled interconnect is likely to be the most area-efficient until we get very close to a single-bit interconnect.

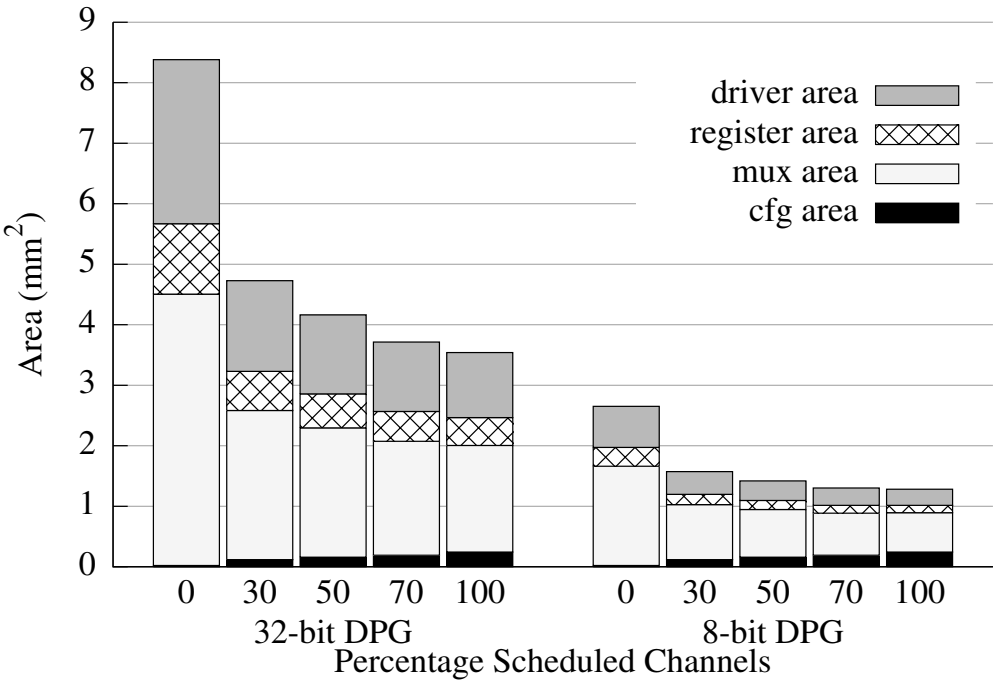


Figure 6.7: Average area for global routing resources.

6.6.3 Hardware Supported Initiation Interval

Each architecture has a maximum initiation interval, set by the number of configurations for the CGRA that can be stored in the configuration SRAM. For an application to map to the CGRA the application's II must be less than or equal to the CGRA's maximum supported II.

Our results indicate that scheduled channels are beneficial for word-wide interconnect. However, the overhead associated with scheduled channels changes with the number of configurations supported. Figure 6.8 shows the area-energy curves for 32-bit and 8-bit datapaths with hardware that supports an II of 16, 64 or 128. The curve for an II of 128 on an 8-bit datapath is most interesting. In this configuration the overhead of the scheduled channels begins to dominate any flexibility benefit. For the other cases, fully scheduled is a promising answer. Note that support for a max II of 16 to 64 in the hardware is a reasonable range in the design space, given that 9 is the largest II requirement for any of our existing

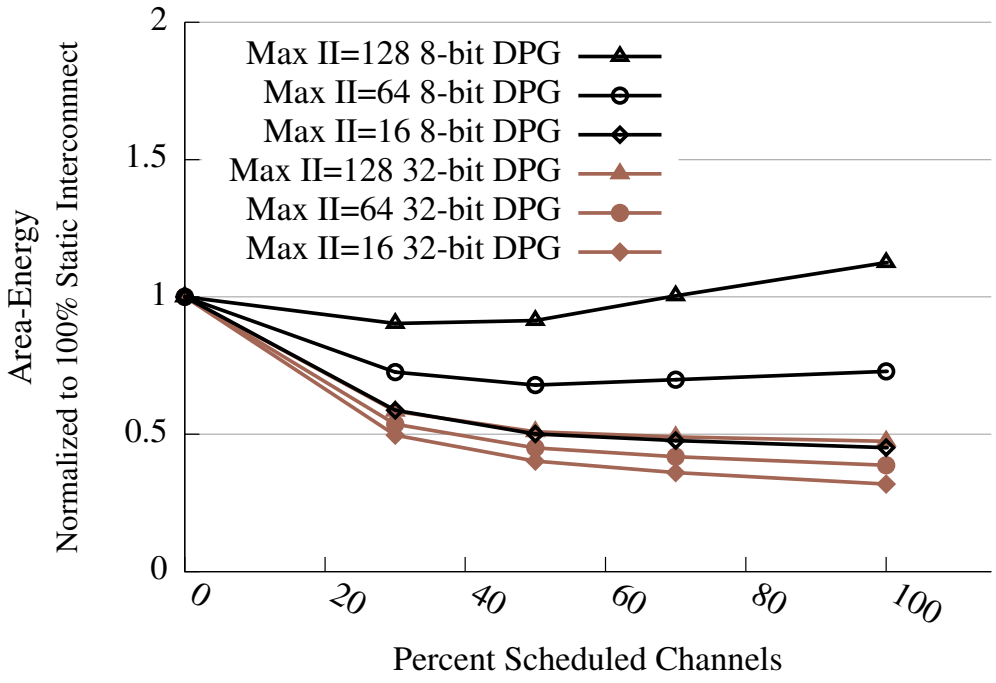


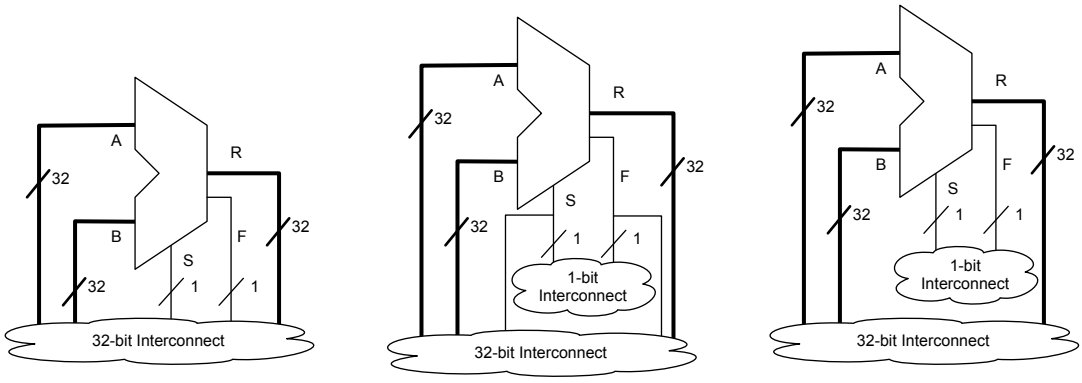
Figure 6.8: Area-energy product for 32- and 8-bit datapath word-widths and maximum supported II of 64 and 128 configurations, as the interconnect becomes more scheduled.

benchmarks. A related study [78] showed that an II of 64 would cover >90% of loops in MediaBench and SPECFP, and that the maximum II observed was 80 cycles.

6.6.4 Augmenting CGRAs with Control-Specific Resources

The datapath applications we used for benchmarks are dominated by word-wide operation. However, there is still a non-trivial amount of single-bit control logic within each kernel. One major source of this control logic is due to if-conversion and predication; another is loop bound control. Since our architectures can only execute a fixed schedule, to achieve data dependent control flow `if`-statements must be predicated such that both possible outcomes are executed and the correct result is selected using the single-bit predicate. It is common in our applications for 20-30% of the operations and values in the dataflow graph to be control logic that only require 1 bit of data. Routing 1-bit signals in a 32-bit interconnect

wastes 31 wires in each channel.



(a) Word-wide only interconnect (b) Word-wide and integrated single-bit interconnect (c) Word-wide and split single-bit interconnect

Figure 6.9: Examples of how an ALU is connected to interconnects that are: word-wide only, integrated word-wide and single-bit, and split word-wide and single-bit.

The obvious alternative is an interconnect that includes both 32-bit and 1-bit resources, optimized to the demands of each signal type. Examples of how an ALU attaches to each type of interconnect are shown in Figure 6.9. We experiment with two distinct combinations of word-wide and single-bit interconnects: split and integrated. For the split architecture, there are disjoint single- and multi-bit interconnects, where each port on a device is only connected to the appropriate interconnect. For the integrated architecture, datapath signals are restricted to the 32-bit interconnect, while control signals can use either the 32-bit or 1-bit interconnect, which should reduce inefficiencies from resource fragmentation. Note that the integrated and split architecture's use a fully scheduled 32-bit interconnect, while the 1-bit interconnect is tested with a range of scheduled-to-static ratios.

As we see in Figure 6.10, adding a single-bit interconnect reduces the channel width of the multi-bit interconnect. This translates into a reduced area-energy product, though it did not vary significantly with the single-bit interconnect's scheduled-to-static ratio. Given a similar area-energy product, a fully static single-bit interconnect is simpler to design and implement, and thus may be preferable. The variations in the single-bit channel width

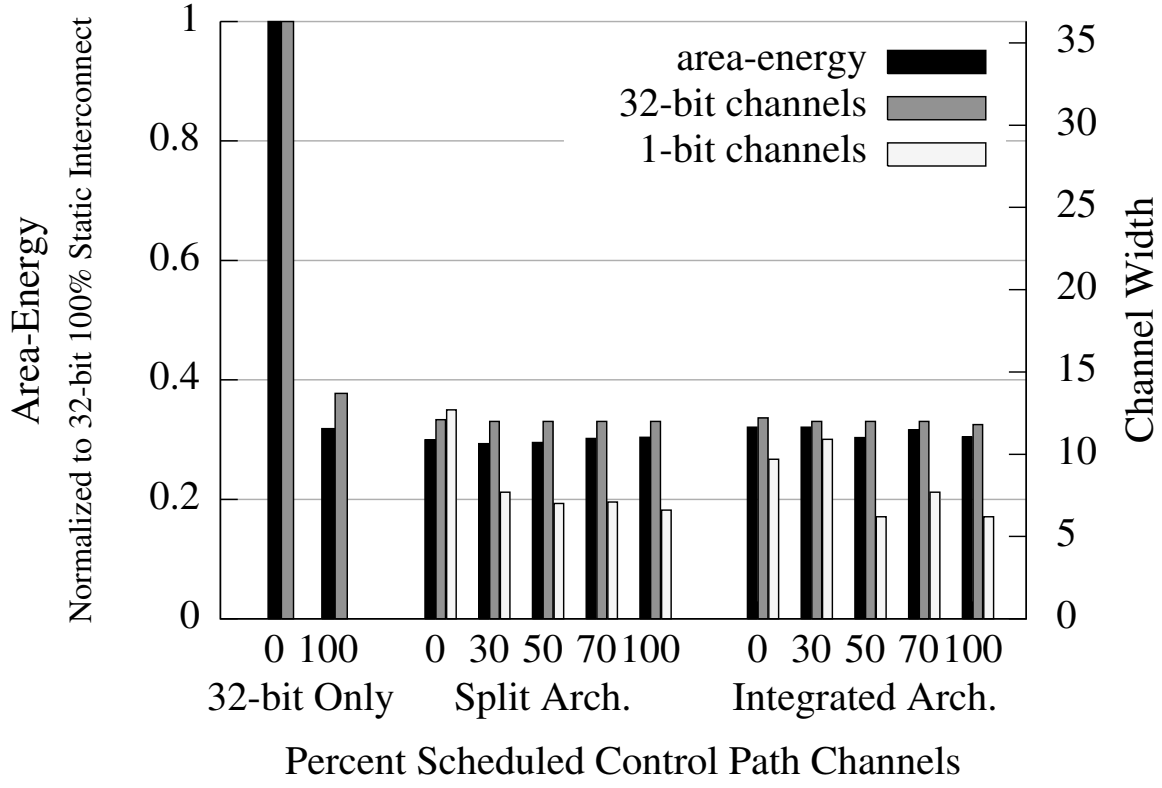


Figure 6.10: Comparison of 32-bit only, split, and integrated interconnects. Area-energy is shown on the left y-axis and 32- and 1-bit channel width on the right y-axis.

for the integrated architecture likely results from the increased complexity in the routing search space, making it more challenging for the heuristic algorithms to find good solutions. Even if we only examine the benchmarks that achieved a smaller channel width in the integrated tests compared the split tests, there is no measurable improvement in area-energy product. This leads us to conclude that the integrated architecture is not worth the additional complexity or reduction in SPR's performance.

We can compare the results for a split architecture with a scheduled 32-bit and a static 1-bit interconnect to the scheduled 32-bit-only baseline from 6.6.1 using the line-graph visualization in Figure 6.11. We see a reduction in the number of scheduled channels to 0.88× that of the scheduled 32-bit-only baseline, though there should be somewhere

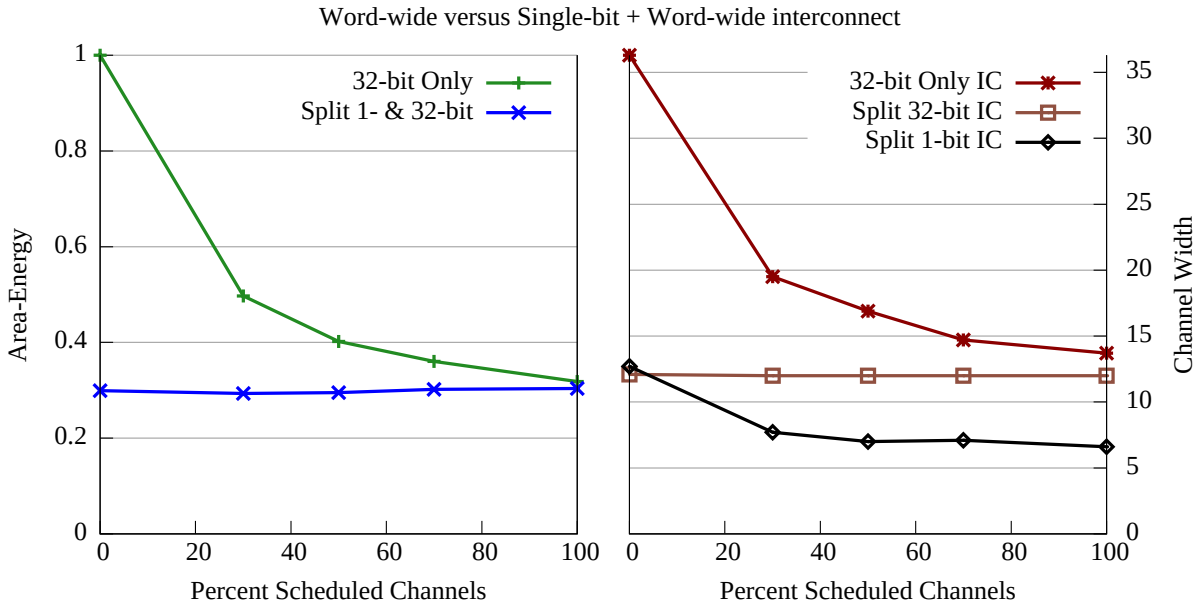


Figure 6.11: Alternate visualization of 32-bit only versus split interconnects. Area-energy is shown in the left graph and 32- and 1-bit channel width on the right graph.

between half as many and an equal number of 1-bit channels as 32-bit channels in the split architecture. Alternatively, we can view this as replacing 1.6 32-bit channels with 12.7 1-bit channels. Note that the routing inside the clusters is also more efficient in the split architecture than the 32-bit only architecture, since separate control and data crossbars will be more efficient than one large integrated crossbar. This effect was beyond the scope of the toolchain and measurement setup when this experiment was performed, but was evaluated using the baseline Mosaic architecture from Chapter 9 and is presented in the subsequent paragraph. Comparing split to scheduled 32-bit only architectures, the overall area and energy of the interconnect (ignoring any impact from the crossbar) is reduced to 0.98× and 0.96× respectively, and area-energy product is reduced to 0.94× the 32-bit only architecture. The split architecture's area-energy product is 0.30× that of the 100% static datapaths without dedicated control resources. Incorporating the impact of the split crossbar would further reduce the area-energy of the split architecture.

To evaluate the effect of splitting the intra-cluster crossbar into two disjoint crossbars, we used the baseline Mosaic architecture that was described in Chapter 9, which had physical models for the crossbar components. We compared the baseline architecture to a variant that was identical except for the split dpg-cpg intra-cluster crossbar, thus isolating the advantages of the disjoint crossbars. For the split dpg-cpg architecture (where each crossbar had the same number of I/O ports as the baseline), the average area of the crossbars was 0.36× the area of the baseline's crossbar. Furthermore, for the multiplexers and drivers in the crossbars, the average dynamic energy per bit, static energy, and configuration energy, was 0.70×, 0.43×, 1.02× the baseline's crossbar resources, respectively.

6.7 Conclusions

This chapter explored the benefits of time-multiplexing the global interconnect for CGRAs. We found that for a word-wide interconnect, going from 100% statically configured to 100% scheduled (time-multiplexed) channels reduced the channel width to 0.38× the baseline. This in turn reduced the the energy to 0.75×, the area to 0.42×, and the area-energy product to 0.32×, despite the additional configuration overhead. This is primarily due to amortizing the overhead of a scheduled channel across a multi-bit signal. It is important to note that as the datapath width is reduced, approaching the single bit granularity of an FPGA, the scheduled channel overhead becomes more costly. We find that for datapath widths of 24-, 16-, and 8-bit, converting from fully static to fully scheduled reduces area-energy product to 0.34×, 0.36×, and 0.45×, respectively.

Another factor that significantly affects the best ratio of scheduled versus static channels is the maximum degree of time-multiplexing supported by the hardware, *i.e.* its maximum II. Supporting larger II translates into more area and energy overhead for scheduled channels. We show that for a 32-bit datapath, supporting an II of 128 is only 1.49× more expensive in area-energy than an II of 16, and a fully scheduled interconnect is still a good choice. However, for an 8-bit datapath and a maximum II of 128, 70% static (30% scheduled) achieves the best area-energy performance, and fully static is better than fully scheduled.

Lastly, while CGRAs are intended for word-wide applications, the interconnect can be further optimized by providing dedicated resources for single-bit control signals. In general,

we find that augmenting a fully scheduled datapath interconnect with a separate, fully static, control-path interconnect reduces the number of datapath channels to 0.88× and requires a control-path of roughly equal size. As a result the area-energy product is reduced to 0.94×. This leads to a total area-energy of 0.30× the base case of a fully static 32-bit only interconnect, a 3.3× improvement.

In summary, across our DSP and scientific computing benchmarks, we have found that a scheduled interconnect significantly improves channel width, area, and energy of systems with moderate to high word-widths that support a reasonable range of time-multiplexing. Furthermore, the addition of a single-bit, fully static, control-path is an effective method for offloading control and predicate signals, thus further increasing the efficiency of the interconnect.

Chapter 7

MANAGING SHORT-LIVED AND LONG-LIVED VALUES

Spatial processors, such as field programmable gate arrays (FPGAs), massively-parallel processor arrays (MPPAs), and coarse-grained reconfigurable arrays (CGRAs), accelerate applications by distributing operations across many parallel compute resources. As a result, these spatial architectures are unable to efficiently take advantage of very large, centralized storage structures that are commonly found in sequential processors. To maintain a high level of performance, it is necessary to have storage structures distributed throughout the architecture that hold all live values and provide the required bandwidth.

There are several possible design choices for these small distributed storage structures, but to date there hasn't been a thorough comparison of their area and energy trade-offs. This chapter explores the role of specialized register structures for managing short-lived and long-lived values in coarse-grained reconfigurable arrays. Furthermore, we characterize the area and energy costs for several register structures, and quantify their advantages along with their disadvantages.

7.1 Challenges

As described in Chapter 1, a common use case for CGRAs is accelerating the loop-level parallelism in the inner loops of an application. One technique for exploiting loop-level parallelism (described in Section 3.3) is to software pipeline the loop, which is then executed using a modulo schedule. Modulo scheduled loops are a very efficient technique for executing loops on CGRAs that complicate the already challenging design problem of efficiently distributing storage in a spatial architecture. Architectures such as ADRES [73], MorphoSys [19], and MATRIX [25], use (or support) a statically computed modulo schedule for controlling ALU, storage, and routing resources.

Modulo schedules [28, 29] were developed for VLIW processors to provide a compact

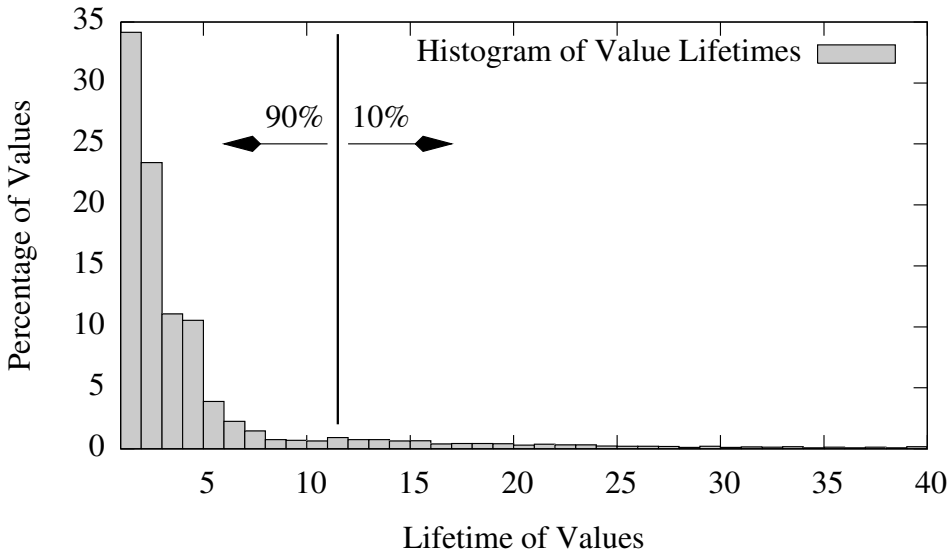
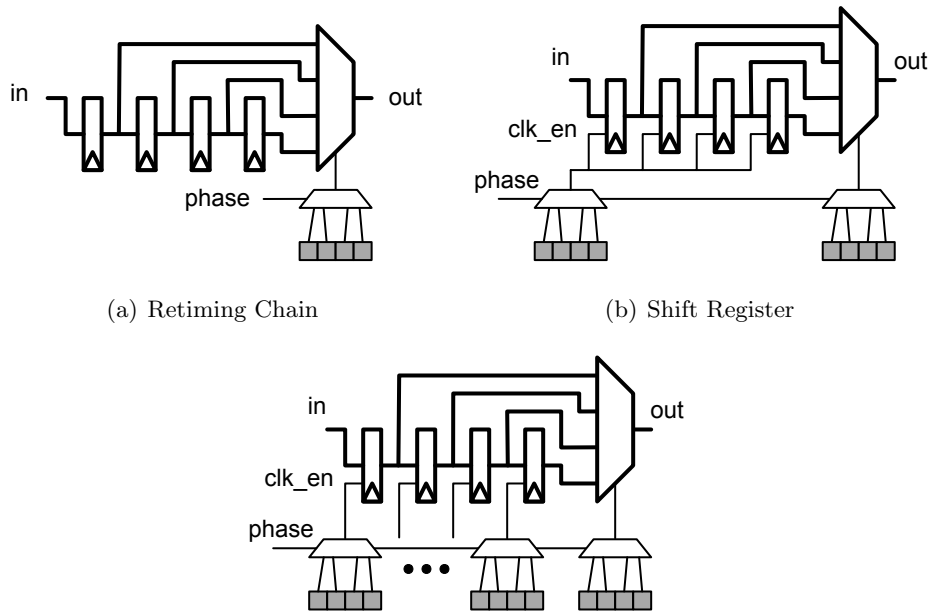


Figure 7.1: Histogram of value lifetimes for Mosaic benchmarks.

encoding of the loop while maintaining high performance. As noted in Section 3.3, a modulo schedule is executed one or more times for each iteration of the application's loop. Each pass through the modulo schedule is referred to as a wave, and the number of waves per iteration of the application's loop is determined by the ratio of the length of the application's loop and the length of the modulo schedule. The rate at which the modulo schedule restarts is the initiation interval (II), and is inversely proportional to the loop's throughput.

The use of a modulo schedule allows us to characterize the lifetimes for values as short-, medium-, and long-lived. Short-lived values have only one or a few cycles between being produced and consumed (*i.e.* less than the loop's II). An intermediate lifetime is approximately II$\pm$1 cycles, and a long lifetime is greater than II (sometimes many II) cycles. Long lifetimes typically arise when a value is produced and consumed in different iterations of the application's loop, or for constant values in the loop. Examination of applications for coarse-grained reconfigurable arrays (CGRAs) shows that most values are short-lived (Figure 7.1); they are produced and consumed quickly, but the distribution of value lifetimes has a reasonably long tail.



(a) Retiming Chain

(b) Shift Register

(c) Variable Shift Register

Figure 7.2: Retiming chain and shift register structures containing 4 registers.

Due to the cyclic nature of the modulo schedule, storing a long-lived value in a modulo scheduled architecture is challenging, particularly when using structures such as traditional register files. If the address of a storage location is statically determined, then the distance between a register write and subsequent read can be at most II cycles in a modulo schedule. Otherwise, the same write in the next wave (*i.e.* the next iteration of the schedule) will clobber the current value.

Rau et al. [79] and Dehnert et al. [80] developed the Cydra-multiconnect rotating register file for VLIW processors with modulo schedules. They showed that by rotating the address of the read and write ports it was possible to create a single code sequence for loop bodies and allow values to live longer than II cycles. This reduced the size of executable code segments and simplified the compiler code generation. Table 7.1 provides an example of how rotating register files allow a more compact code representation than traditional register files. The fourth column shows a sequence of operations for three iterations of a loop body with an II=2. The fifth column shows a code sequence that is repeated three

Cycle	Phase	Wave	Logical Operation	Logical Reg. Name	Physical Reg. Name
0	0	0	$c_0 = a_0 \star B$	$r6_{w=0} = r0_{w=0} \star B$	$r6 = r0 \star B$
1	1	0	$d_0 = c_{-1} + c_0$	$r9_{w=0} = r6_{w=-1} + r6_{w=0}$	$r9 = r5 + r6$
2	0	1	$c_1 = a_1 \star B$	$r6_{w=1} = r0_{w=1} \star B$	$r7 = r1 \star B$
3	1	1	$d_1 = c_0 + c_1$	$r9_{w=1} = r6_{w=0} + r6_{w=1}$	$r10 = r6 + r7$
4	0	2	$c_2 = a_2 \star B$	$r6_{w=2} = r0_{w=2} \star B$	$r8 = r2 \star B$
5	1	2	$d_2 = c_1 + c_2$	$r9_{w=2} = r6_{w=1} + r6_{w=2}$	$r11 = r7 + r8$

Table 7.1: Example of code generation when using a rotating register file for a three iterations of a loop with an II=2. The first columns show the execution cycle, phase, and wave. The last three columns provide the operation to be executed as well as register level code with both logical and physical register names. Note that the logical register names are annotated with a wave offset that determines the logical-to-physical renaming.

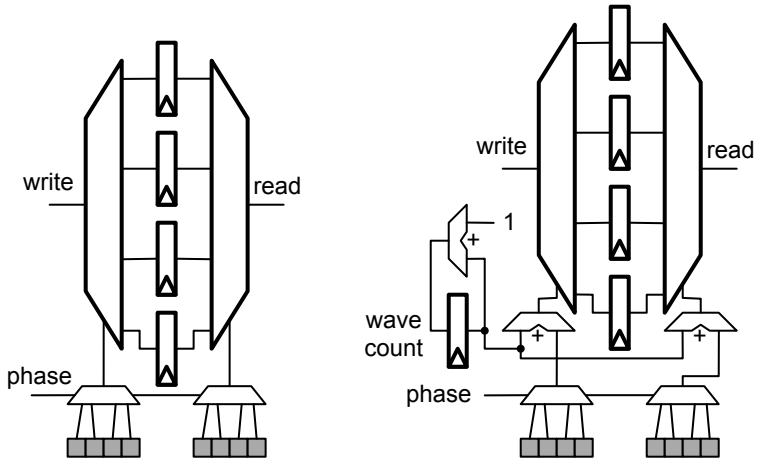
times, once for each wave. Note that the subscript wave designation is solely for the reader and not part of the actual code snippet. The last column shows the actual physical register names that would be used by the rotating register file and is a valid code snippet for an architecture with traditional register files. Thus, the rotating register file compacts a 6-line code snippet down to 2 lines of code by providing logical-to-physical register renaming.

Despite its advantages, few CGRAs with module schedules make use of the rotating register file. One notable exception is the ADRES architecture [67, 68], which describes having rotating register files but does not evaluate their advantages or overhead separately from other architectural features. The RaPiD [74] architecture also provided a rotating register file using a similar mechanism, but with an explicit shift command.

7.1.1 Value Lifetime Analysis

Figure 7.1 shows the histogram of value lifetimes for our benchmarks, which had an average II of 3.9 cycles. We see that 58% of the values have a lifetime of at most 2 cycles, and 79% have a lifetime of at most 4 cycles. Furthermore, the percentage of values with a given

lifetime rapidly approaches zero after a lifetime of 4. Overall the average of the first 90% of value lifetimes is 3 cycles, and the remaining 10% have an average lifetime of 24 cycles. Given such a distribution we expect that a heterogeneous collection of storage structures, some optimized for short-lived values and others for long-lived values, will be better than a single universal storage structure.



(a) Register File (b) Rotating Register File

Figure 7.3: Register file structures composed of 4 registers.

7.2 Storage Structures

There are several choices for storing values in a CGRA: memories, register files, shift registers (or FIFOs), pipeline registers, and individual registers distributed throughout the system. In Figures 7.2 and 7.3 several of these structures are shown, with 4 registers and one input and output port each. The grey boxes are configuration memory cells, which represent the configuration words for a CGRA's modulo schedule. These configuration words control multiplexers in all of the designs, as well as register enables in 7.2(b) and 7.2(c). These structures represent the majority of the design space for collecting and composing registers.

Table 7.2 shows the area and energy characteristics for each register structure; the methodology for generating these characteristics is presented in Section 7.3.2. Dynamic

Structure Type	Abrv.	Area (um^2)	Static energy per clock	Energy per write	Energy per read	Recurring energy per value	Max. Lifetime
Rotating Reg. File	RRF	1947	132.8	$366.7 + 29.6 * \texttt{\#bits}$	$329.7 + 5.7 * \texttt{\#bits}$	$\frac{25.5 * \texttt{\#waves}}{\texttt{\#values}}$	$4 * II$
Register File	RF	1723	130.3	$293.3 + 29.6 * \texttt{\#bits}$	$256.3 + 5.7 * \texttt{\#bits}$	0	II
Retiming chain	RT	1580	82.0	$10.4 * \texttt{\#bits}$	$256.3 + 5.7 * \texttt{\#bits}$	$(10.4 * \texttt{\#bits} * \texttt{\#cycles})$	4
Shift register	SR	1739	177.3	$10.4 * \texttt{\#bits}$	$256.3 + 5.7 * \texttt{\#bits}$	$(10.4 * \texttt{\#bits} * \texttt{\#shifts})$	$4 * II$
Dist. Reg.	DistR	292	30.0	$10.4 * \texttt{\#bits}$	0	0	II
Crossbar write port		3972	227.7	$181.9 * \texttt{\#bits}$	0	0	N/A
Crossbar read port		2839	165.6	0	$23.2 * \texttt{\#bits}$	0	N/A

Table 7.2: Area and energy metrics for each type of register block (each with 4 register entries), a single distributed register, and a read and write port on the cluster crossbar. Energy is reported in femtojoules (fJ) and is based on the number of bits that change in the structure. Static energy is based on an average clock period of 3.06ns.

energy is reported as energy per write and read plus the recurring energy to store a value for multiple cycles. Write or read energy is accrued for each bit of a value plus a fixed amount per operation. The final column is the longest that a value can be stored in a structure. Table 7.2 also includes the area and energy cost of a read and write port for an average cluster's crossbar.

7.2.1 Retiming Chains

Retiming chains were used in both HSRA [22] and RaPiD [74] for input and output retiming, buffering values with short lifetimes that travel directly between functional units. A retiming chain, illustrated in Figure 7.2(a), is a sequence of registers with one input and one output, and a multiplexer that is used to select the value at different stages of the chain. Values advance through the chain on each clock cycle, making the hardware very simple. In our implementation, SPR is able to statically clock gate inactive registers in partially used retiming chains (for the duration of the kernel), thus conserving energy. In Table 7.2 we see that the retiming chain is the smallest structure, consumes the least amount of energy per write, and has the lowest static energy. However, it also has the highest recurring energy, which results from transferring the contents of one register to the next on each cycle. Finally, the longest that a value can be stored in it depends solely on its length.

7.2.2 Shift Registers

Figure 7.2(b) shows a shift register – essentially a retiming chain with a common register enable to control the shifting of values. We use a specialized version (Figure 7.2(c)) called a variable shift register, that dynamically determines how many of the registers to shift and how many to hold. This is achieved by using individual configuration bits for each register's enable, rather than a shared enable signal. On each phase of execution the shift register is divided into two contiguous regions: the fore region that shifts data and the aft region that holds its current data. The split point for these regions varies from cycle to cycle, and either region can be empty. By varying the size of the aft region, the lifetime that a value can reside in the shift register is substantially improved.

Unlike the retiming chain, the shift register does not have to shift a value through the chain on every clock cycle. Furthermore, with the variable shift register, values in the aft portion of the structure do not have to shift when new values are pulled into the structure. This feature reduces the energy consumed and allows a value to be stored for longer in the shift register than the retiming chain. However, as the utilization increases, the values are forced forward in the chain, and its power consumption becomes similar to that of the retiming chain.

7.2.3 Register Files

Register files allow greater flexibility for reordering values than either retiming chains or shift registers. As shown in Figure 7.3(a) they use steering logic on both read and write ports. A difference between register files and both the retiming chain and shift register is that register files have a significant energy cost per write. This cost is a result of the decoder and initial fanout to all registers in the structure, versus only writing to the head of the retiming chain or shift register. As a result, to make a register file competitive against either the retiming chain or shift register a value has to live in the register file long enough to amortize the cost of the initial write. This is the primary reason that register files are a poor choice for storing short-lived values.

7.2.4 Rotating Register Files

The rotating register file enhances a normal register file for use with modulo schedules. Figure 7.3(b) shows the addition of a wave counter and register that is used as an offset for the read and write address. The wave counter is incremented on each pass through a modulo schedule. The offset acts like automatic register renaming between physical and logical locations in the register file. This simplifies the access pattern to the register file for values that live in the register file longer than the length of the modulo schedule.

For values with a medium life span (up to II cycles) both the register file and the rotating register file perform similarly, consuming dynamic energy only on reads and writes. For values with a lifetime that is longer than the II, the rotating register file has a distinct

advantage over the traditional register file. To store a long-lived value the register file has to read out the value and re-write it to a different location, to avoid being clobbered by subsequent writes to the same register location when the modulo schedule repeats. This process will typically incur a trip across the cluster's crossbar, which, as shown in Table 7.2, is expensive. In contrast, the rotating register file will simply shift its address offset counter so that the next iteration of the value does not overwrite the current instance.

Value	Value lifetime	Read Phase	Possible write phases	Write phase	Latency absorbed
a	3	0	2, 3, **4**	4	1
b	4	1	2, 3, 4, **0**	0	1
c	5	2	2, 3, 4, 0, **1**	1	1
d	1	3	**2**	2	1
e	2	4	2, **3**	3	1
f	4	0	**1**, 2, 3, 4	1	4
g	3	1	**3**, 4, 0	3	3
h	2	2	**0**, 1	0	2
i	1	3	**2**	2	1
j	5	4	**4**, 0, 1, 2, 3	4	5

Table 7.3: Example of a pathologically bad and good value pattern for a retiming chain with a modulo schedule of length 5. Columns show the lifetime of the value, the phase of a modulo schedule that it has to be read back out, possible phases for writing into the retiming chain, the actual write phase, and the number of cycles spent in the retiming chain.

7.2.5 Storage Structure Flexibility

One of the challenges for these different register structures is how they behave when either ports or registers are near full capacity. It turns out that retiming chains tend to behave poorly with certain value patterns when their write port is nearly fully subscribed. Specif-

ically, when the write port is near 100% utilization, the maximum latency that a retiming chain is able to absorb is equal to the minimum remaining lifetime of all of the values that are passing through it. An example of this is shown in Table 7.3, which highlights the differences in how a retiming chain stores two different value patterns in a architecture with an II of 5. The first two column show a sequence of values and their associated lifetimes. The read phase is when in the modulo schedule the value has to be read out of the retiming chain. The possible write phases are previous points in time when the value could arrive at the retiming chain in question. The write phase is when the value is actually scheduled to arrive, and the latency absorbed is how many cycles the value was in the retiming chain. The top part of the table, values a-e, shows a pathologically bad example where, due to scheduling, each value can only spend one cycle in the retiming chain. The bottom part of the table, values f-j, shows a better value pattern, where the retiming chain is able to absorb as much latency as each value requires.

Because of how it handles values, we describe the retiming chain as being inelastic, a result of having a single read mux. Values enter at the head of the chain and are pulled out somewhere along the line, but they cannot reorder themselves unless there is slack in the read schedule. Given that the read schedule is at least as full as the write schedule, and the short maximum lifetime of the retiming chain, there isn't much flexibility in how long a value can be retimed on a busy retiming chain. The shift register is slightly more elastic than the retiming chain because it is able to hold values for longer than the II. While this doesn't create more flexibility in the read and write scheduling, it does allow the lifetime of the values in the shift register to vary more. Both traditional and rotating register files achieve a significant improvement in elasticity by having both a read and write mux.

Overall, the result of these structural limitations mean that it is typically the read and write port occupancy, not capacity, that limits the efficacy of the register structure. However, this is in fact the main trade-off for adding the structure for any of these designs, when compared to distributed registers; therefore, by design these structures are more frequently bandwidth limited rather than capacity limited. Finally, while increasing port capacity allows the architecture to make better use of available register capacity, it does also increase the resource demands on external structures such as the cluster crossbar.

Tables 7.2 and 7.7 show that the crossbar is a large consumer of energy with respect to the storage structures and the rest of the core, respectively, and thus should be minimized. Therefore, when budgeting additional storage resources to provide architectural robustness, it is typically better to over provision register capacity rather than register bandwidth.

7.2.6 Reducing Port Pressure

One mechanism for reducing the port pressure on the register structures is to reduce the number of operands within a given dataflow graph. One large source of operands in an unoptimized dataflow graph are constant values. A parameter in the Macah compiler controls the handling of constant values within the dataflow graph. There are two options: 1) have a single, unique source for each constant value and 2) create an instance of each constant value for each use. The first option optimizes constant storage capacity at the expense of having to communicate these values, while the second option minimizes communication activity and requisite resources at the expense of storage capacity. Given the relative costs of storage versus communication resources, we have chosen to create an instance of each constant value for each use.

As described in Chapter 4.2, another advantage to fully expanding constant operands is that it increases the opportunities for optimizing some of them into specialized instructions. In our benchmarks 29% of all operands in the application's dataflow graph are constant values. Some simple techmapping techniques that fold constant values of 0 and 1 into specific operations reduces the number of live constant values to 22% of all operands. Further optimizations that folds constant values of up to 3 bits into arithmetic and logical shift operations reduces this to 17% of all operands, which is the approach used for the experiments in this chapter.

One very interesting challenge in a CGRA is how to balance optimization and folding of constant values into arithmetic operations. In a traditional RISC microprocessor the instruction word is fairly large (*e.g.* 16- to 32-bits) and the register file port identifiers typically take 5 to 6 bits each. This makes it easy to fit large constants in place of one or more of the operands, or even the majority of the instruction word. A CGRA on the other hand

can have opcodes for the arithmetic units in the range of 5 or less bits, and the configuration bits for routing operands to the arithmetic unit may not be available for re-purposing to store constant values. For these reasons efficient management of constant values is not straightforward and requires further investigation. An initial solution is explored in Section 7.11, which provides functional units with private storage structures that can hold multi-bit constant values. The local storage comes from grouping a small register block with each ALU. Then, when using either a register file or rotating register file, it is possible to partition the register file into read-write and read-only sections that can store constant values in the register block as is done in the cluster-wide register file. Details of this approach are provided in Section 7.11. Providing local storage to the functional unit that is flexible enough to be repurposed between constant and runtime value storage is superior to allocating dedicated bits in the functional unit's configuration memory. In particular, supporting two 3-bit immediate fields for each operand requires 128 bits of storage per functional unit, for an architecture with a maximum II of 16. That is the same number of storage bits required to add four 32-bit local registers to the functional unit, which can support a wider range of constant values and may also be used for runtime values. Therefore, this technique reduces the value of techmapping 2- and 3-bit constant values into arithmetic operations, establishing that the techmapper for subsequent experiments should fold only 1-bit constants into the immediate field of operators.

7.2.7 Storage Structures Synopsis

A priori we can surmise the following advantages and disadvantages for each structure. The rotating register file and shift register are more sophisticated versions of the register file and retiming chain; they can store values longer but have a small additional area and energy overhead. Register files have a disadvantage because the initial energy required to write into them make them a poor choice for short-lived values and they are unable to hold long-lived values. The retiming chain, which is the simplest possible structure, would win based on area-only analysis, but the cost of moving data on every cycle means that the dynamic energy cost will exceed its static energy and area advantages.

The remainder of this chapter describes our experiments aimed at determining the best composition of storage structures for CGRAs. We seek to answer the question of how many of these different structures should there be, and how should they be distributed throughout the architecture.

7.3 Background

A key detail about the Mosaic toolflow (described in Chapter 4) for these experiments is that all of the registers and register structures are allocated by the routing phase of SPR. However, it is important to note that the CGRA's embedded memories are exclusively managed by the programmer, with the read and write operations appearing as consumers and producers in the dataflow graph. Section 4.3 describes the flexibility and extensibility of the SPR CAD tool [48] that was instrumental to analyzing the storage structures of interest, the most crucial of which were the customization of the configuration generation that allowed SPR to provide efficient configurations for the rotating register file and shift register. Although these experiments were conducted on a CGRA architecture, we believe that they apply more broadly to other styles of spatial architectures.

Chapter 6 explored the advantages of scheduled versus statically configured interconnect for CGRAs in a grid topology, and the advantage of a dedicated 1-bit control network that augments a word-width datapath. This chapter builds upon these results, using a partitioned interconnect with a fully scheduled, 32-bit, word-wide datapath and a fully static single-bit control path. The methodology for modeling and simulating the interconnect is described in Chapter 5, and is the same as in Chapter 6.

7.3.1 Benchmarks

The applications used for benchmarking are described in Section 5.1 and are listed in Tables 7.4 and 7.5. Each application has a listed minimum initiation interval (II) in the second column of Table 7.4 that represents the largest loop-carried dependence of the original algorithm. Other columns of Table 7.4 indicate the number of operations in the application and the size of the targeted CGRA. Note that we size each application, and target architecture, so that no resource in the array is utilized more that 80% across both space and time.

Application	Min. II	DFG Operations	Memory Arrays	Stream IO	Constant Values	# Clusters	CGRA Grid Size
64-tap FIR filter	2	263	0	2	130	42	6x7
240-tap FIR filter (Banked Mem)	3	233	48	2	79	42	6x7
2D convolution	4	237	6	2	132	20	4x5
8x8 Blocked Matrix multiplication	4	362	0	80	86	49	7x7
K-means clustering (K=32)	6	538	32	34	117	30	5x6
Matched filter	4	180	30	15	69	30	5x6
Smith-Waterman	5	444	11	8	121	30	5x6
CORDIC	2	137	0	8	23	20	4x5
8×8 Motion estimation [63]	5	379	16	11	163	20	4x5
PET Event Detection	4	607	28	14	219	49	7x7

Table 7.4: Benchmark Applications and Simulated Architecture Sizes

Application	Tuning Knobs
64-tap FIR filter	Num. Coefficients = 64
240-tap FIR filter (Banked Mem)	Num. Coefficients = 240, Num. Memory Banks = 24
2D convolution	Kernel Size = 7x7, Vertical Stripe Width = 15
8x8 Blocked Matrix multiplication	Blocking Factor = 8x8
K-means clustering (K=32)	Num. Clusters = 32
Matched filter	Num. Parallel Channels Processed (Block Size) = 2, Num. Signatures = 13, Num. Channels = 16
Smith-Waterman	Stripe Width = 11
CORDIC	Num. CORDIC iterations = 3
8×8 Motion estimation [63]	Image Block Size = 8x8, Num. Parallel Searches = 8
PET Event Detection	Num. Parallel Datasets = 8

Table 7.5: Tuning Knob settings for Benchmark Applications

Chapter 5 described the evaluation and simulation methodology for all experiments, and is briefly recapped here. Each alternative architecture is evaluated by running each benchmark with four different placement seeds. Typically the area-energy-delay product is the best metric for evaluating these architectural tradeoffs, but each architecture template is optimized to provide nearly equivalent application performance. This lets us simplify the analysis to use just the area-energy product for comparison. We hold performance constant for each application by forcing SPR to use the application's minimum II across architectures, and running architectures at similar clock frequencies. The application-architecture mapping with the lowest total energy was selected and used to compute the area-energy product for that benchmark. A summary of the architectures tested and their improvements in area-energy product versus the baseline architecture will be provided in Table 7.6.

7.3.2 Energy, Area, and Performance Evaluation

The evaluation framework for each architecture template (described in Section 4.6) used the VCS Verilog simulator to track bit transitions for each signal and the activity of physical structures such as rotating register files. The physical models were created using full custom design and memory compilers in a 65nm process. Energy and delay characteristics were obtained from SPICE simulations and a commercial memory compiler. As noted in Section 4.6.2 the physical models for the experiments in this chapter were drawn from a mixture of the IBM 65nm 10SF and 10LP processes, due to the timing of their release. The memory compilers used the 10LP process, while all models that were custom designed used the 10SF process. As a result, only the large register files and embedded memories used the results from the memory compiler and the 10LP library. Given that the impact of the large register files was tested independently from the impact of the smaller register block structures the results are internally consistent, as there were no direct cross-library comparisons.

7.3.3 Related Work

In [68], Bouwens et al. performed an architecture exploration that compared interconnect topologies and register sharing for the ADRES template. Of greatest interest is the con-

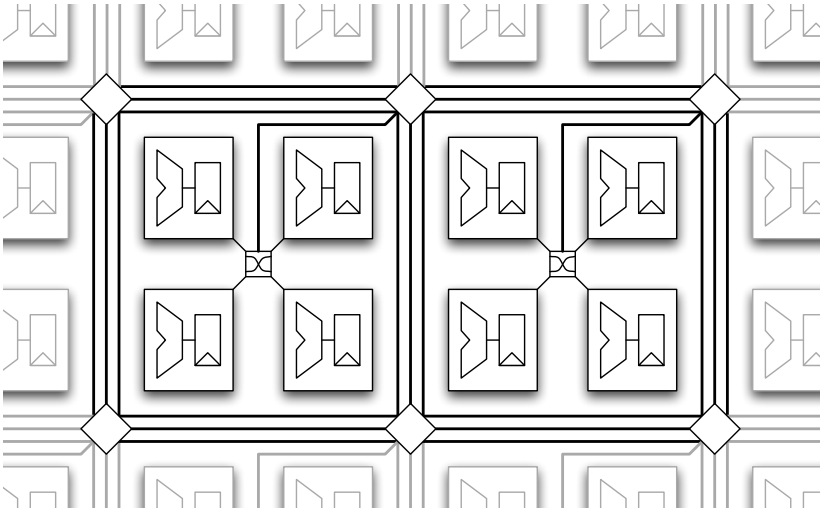
clusion that their best design uses highly shared, multi-ported (2 read / 1 write) register files where each functional unit's local register file is shared with the four diagonally adjacent functional units. Further exploration by Bouwens [67] showed an advantage when separating word-wide, datapath register files from specific functional units and just connecting four diagonally adjacent functional units. This move to decouple the local, shared storage and move to a design with less spatial locality between storage and functional units is similar to the connection pattern of the large cluster-wide register files that are evaluated in Section 7.5, but contrasts the design choices of the short-term storage in Section 7.10. Predicate register files were kept paired with each functional unit, but still shared. The push in ADRES to separate the register files from the functional units stems from the vast difference in sizes of arrays explored, and the advantages of a large unified register file that connects to all functional units in ADRES, but is absent in the Mosaic architectures.

7.4 Baseline CGRA Architecture

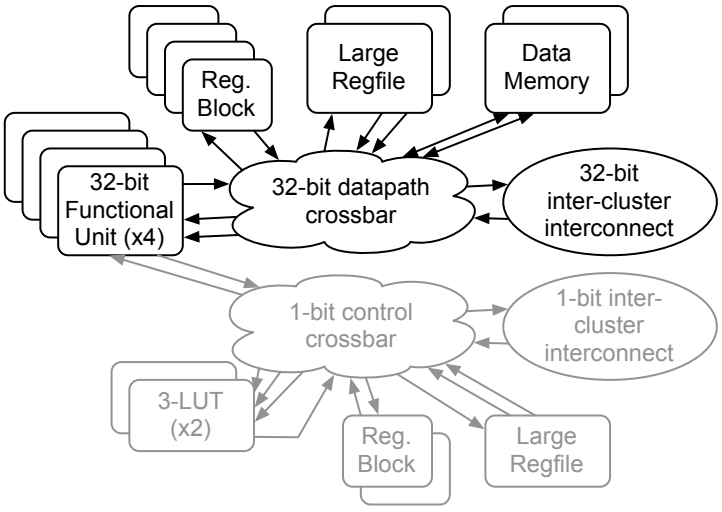
The baseline coarse-grained reconfigurable array architecture that we study is presented in Chapter 3 and shown in Figure 7.4(a). For these experiments we have extended the baseline architecture to have separate 32-bit and 1-bit components that include functional units, crossbars, and interconnect. The functional units retime values on their inputs rather than their outputs, which was shown by Sharma et al. [47] to provide a 19% improvement in area-delay product.

The 32-bit components in each cluster comprise four arithmetic and logic units, two embedded memories, short- and long-term register storage structures, and possibly send and receive stream I/O ports. The embedded memories are used for storage that is explicitly managed by the programmer, and not by the SPR compiler. Therefore they behave as simple value producers and consumers. The 1-bit components in each cluster comprise two 3-LUTs, and register storage that has the same structure as the 32-bit datapath.

Register storage is split into short-term and long-term storage structures. The short-term storage structures contain a small number of registers and are generically referred to as register blocks. Register blocks have 4 registers each because they are designed to store the short-lived values, the majority of which have a lifetime of 4 or less. The large register



(a) CGRA Fabric



(b) Cluster Diagram

Figure 7.4: Block diagrams of CGRA with clusters of 4 PEs connected via a grid of switch-boxes, and an individual cluster with 32-bit datapath in black and 1-bit control path in grey.

files, shown in Figure 7.4(b), provide long-term storage. They have 16 entries each, enough to hold long-lived values and constant values, while not too many to be limited by a single write port.

To balance resources in the CGRA and avoid gross over-provisioning, there is a modest amount of specialization in the architecture. Specifically, there are two types of clusters in the array: edge tiles and core tiles. Clusters on the edge of the CGRA fabric have stream send and receive I/O ports, while core clusters have an additional embedded memory. This heterogeneity serves to keep the average cluster crossbar size down, but does lead to some resource imbalance that lowers the target resource utilization below 80% for some applications.

7.5 Managing long-lived values

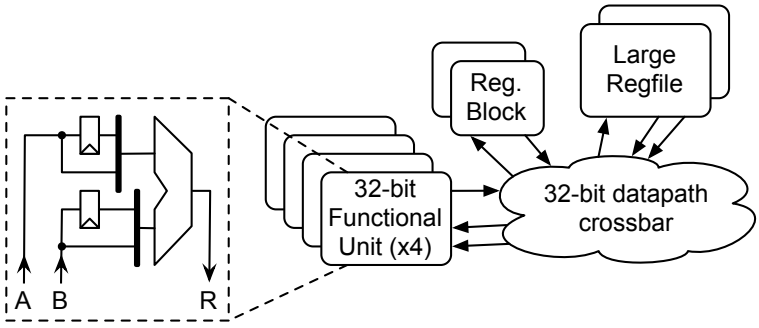


Figure 7.5: Simplified diagram of 32-bit datapath for baseline architecture and functional unit with pipeline registers for input retiming.

The first experiment examines the effect of using rotating register files for the long-term storage instead of traditional register files. Traditional register files are a common storage structure for spatial accelerators, and serve as the baseline long-term storage for these experiments. Figure 7.5 shows a simplified diagram of the baseline architecture, with two large, 16-entry register files that provide the primary long-term storage. They are directly attached to the cluster's crossbar, and shared between the functional units. The functional units use pipeline registers for input retiming. The experiments were conducted with each

Arch. Name	# Dist Regs.	# Reg Blocks	Reg Block Location	Input retiming	Feedback	Registered FU Out.	Large RF	Private Const.	Improvements in area-energy vs. Base 7.5
Base 7.5	2	0	cluster	pipeline	none	no	2x RF	no	1.0×
Opt. 7.6	2	0	cluster	pipeline	none	no	**2x Cydra**	no	0.81×
Alt. 7.6	**0**	**2x 4-RRF**	cluster	pipeline	none	no	2x Cydra	no	0.82×
Opt. 7.7	**1**	**0**	cluster	**enabled**	none	no	2x Cydra	no	0.77×
Alt. 7.7	**0**	**1x 4-RRF**	cluster	enabled	none	no	2x Cydra	no	0.77×
Opt. 7.8	0	**0**	cluster	enabled	**yes**	no	2x Cydra	no	0.75×
Opt. 7.9	0	**1x 4-RRF**	cluster	enabled	**yes**	**yes**	2x Cydra	no	0.67×
Opt. 7.10	0	1x 4-RRF	**FU**	enabled	yes	yes	2x Cydra	no	0.64×
Opt. 7.11	0	**1x 8-RRF**	FU	enabled	yes	yes	**1x Cydra**	**yes**	0.61×

Table 7.6: Summary of experiments with baseline, optimized, and alternate architectures, showing incremental and overall improvements in area-energy product. Bold entries show architectural features that change from row to row.

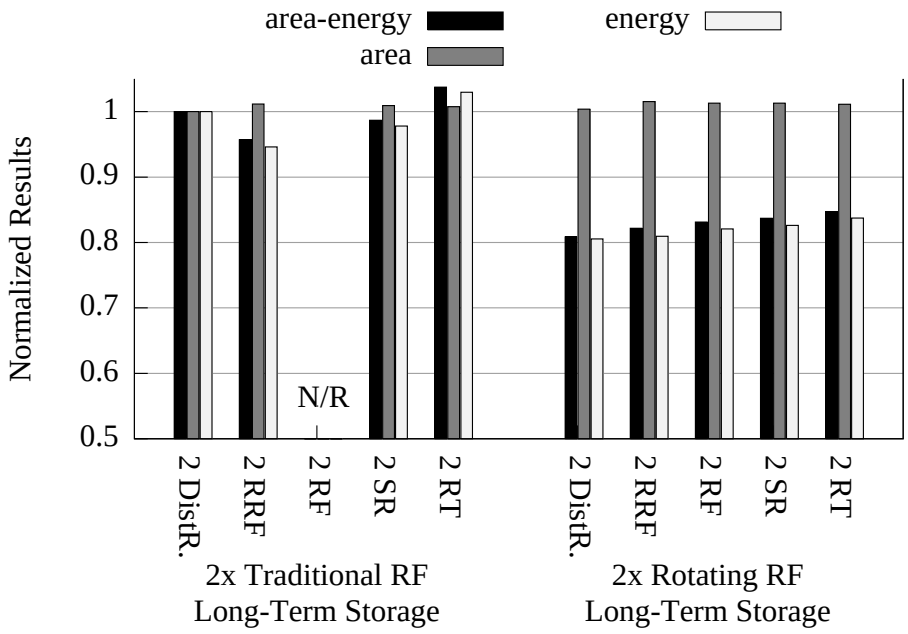


Figure 7.6: Area, energy, and area-energy results for comparing long-term storage techniques. Each metric is independently normalized to the baseline architecture that has 2 traditional register files for long-term storage and 2 distributed registers for short-term storage. Note that there was insufficient data for the architecture with 2 traditional register files for both long- and short-term storage.

type of storage structure for the register blocks, which provide short-term storage. Detailed results are shown in Figure 7.6, and a summary of the results are shown in the first group (Group V) of Figure 7.7.

Replacing traditional register files with rotating registers files reduced the area-energy product by 19% when using two distributed registers for short-term storage. As shown in Figure 7.6, evaluating the other types of short-term storage showed that the area-energy product was reduced by 14%, 15%, and 18%, for architectures with two register blocks configured as: rotating register files, shift registers, and retiming chains, respectively. Note that the architectures with traditional register files for both long- and short-term storage failed to map enough benchmarks to produce two comparable data points. Across all types

of register block structures, and numbers of register blocks in the cluster, there was an average reduction in area-energy of 13%. This occurs because a traditional register file is unable to store a value for longer than II cycles, and so long-lived values have to go through the crossbar to reenter the register file. On average a value spent 11.3 cycles in the large rotating register files and only 3.2 cycles when using traditional large register files. The average II for our benchmarks was 3.9 cycles.

7.6 Style of Short-term Storage

We described in Section 7.2 several different types of register blocks that can be used to store short-lived values. Shift registers and retiming chains offer cheap storage of short-lived values, though with less flexibility than register files or distributed registers. This experiment uses the baseline architecture, shown in Figure 7.5, with two large rotating register files for long-term storage. In this experiment we vary the type and quantity of the register blocks in Figure 7.5.

Figure 7.7-VI shows that the best area-energy product is achieved by using two distributed registers; *i.e.* two register blocks each containing a single register. However, using two rotating register file blocks is almost as efficient in terms of the area-energy product, while providing more storage capacity and routing flexibility. The results for using the simpler structures such as retimers and shift chains are at least 1-3% worse than for rotating register files. This is explained by the fact that these simpler structures consume more energy than rotating register files when storing values with intermediate lifetimes.

7.7 Using dynamically enabled registers for functional unit input retiming

The input retiming registers on the functional unit in Figure 7.5 are fairly inflexible–only holding a value for a single cycle. In this experiment, we investigate the effect of enhancing the input retiming registers with dynamic enables provided by the configuration (similar to the variable shift register), as shown in Figure 7.8(a). This allows them to store values for a longer time, reorder incoming values, and reduces the demand for short-term storage (register blocks) when compared to the previous section.

By dynamically enabling the registers in the functional unit, the demand for register

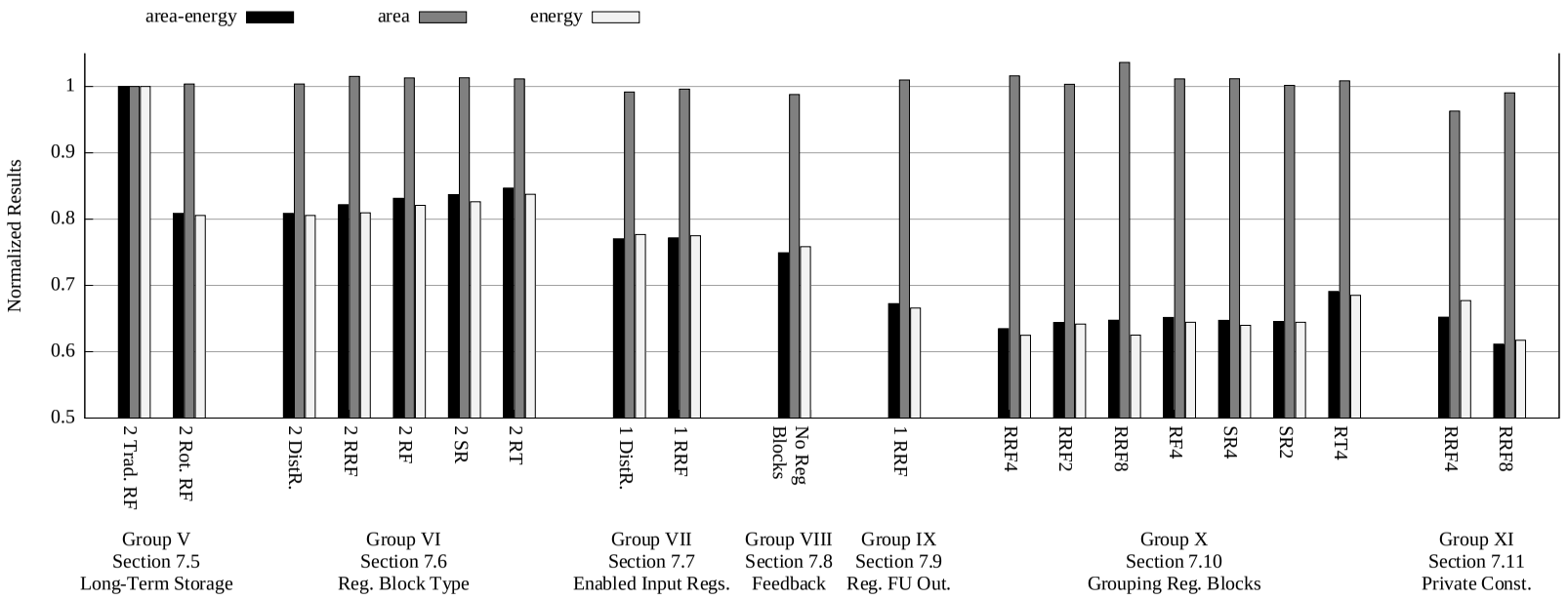
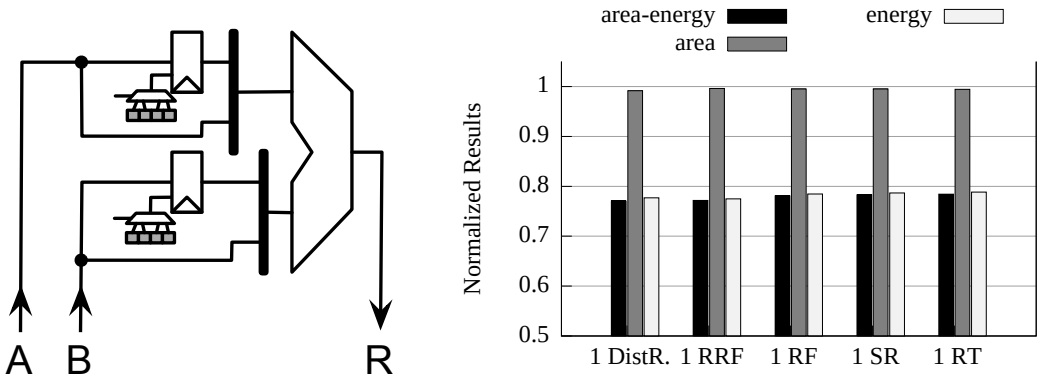


Figure 7.7: Area, energy, and area-energy results for each experimental section, separated into labeled groups. Functional units for each group are shown in Figures 7.5, 7.5, 7.8(a), 7.9, 7.10, 7.11, 7.12, respectively. Each metric is independently normalized to the baseline architecture: Section 7.5 and Figure 7.5.



(a) Functional unit

(b) Detailed results. Each metric is independently normalized to the baseline architecture: Section 7.5 and Figure 7.5.

Figure 7.8: Block diagram of functional unit with enabled registers for input retiming. Additionally, area, energy, and area-energy results for architectures with each type of register block structure.

blocks is reduced. Our testing indicated that the best number of register blocks per cluster drops from two to one. Figure 7.7-VII shows that dynamically enabling the input retiming registers reduces the area-energy product of an architecture with either one rotating register file block or a single distributed register to 0.77× the area-energy of the baseline architecture. Across all register block styles, using enabled input registers reduced the average area-energy by 4% compared to the architectures in Section 7.6, as shown in Figure 7.8(b).

7.8 Adding local feedback paths

In the functional units described thus far, if a value produced by a functional unit is subsequently used by an operation in the same functional unit (a common occurrence), this value must be routed through the cluster crossbar to a functional unit input register. In this experiment, the functional units are augmented by an internal, registered feedback path as shown in Figure 7.9. This path reduces crossbar usage and frees the input registers for storing other values when operations are chained in the same functional unit. On average, this increased static energy by 31%, but it reduced the dynamic energy by 7%.

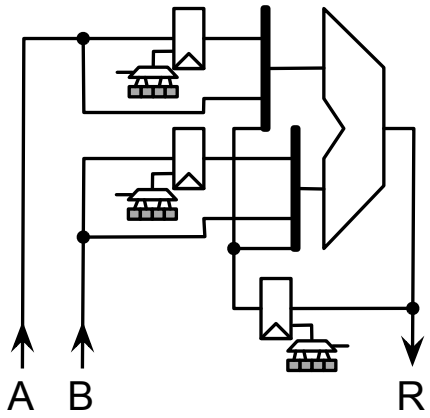


Figure 7.9: Functional unit with internal feedback.

Adding feedback to the functional unit allowed SPR to successfully route all benchmarks on an architecture with no distributed registers or register blocks. Figure 7.7-VIII shows that this architecture has an area-energy product that is 0.75× the baseline. This is 0.98× the area-energy of the best architecture from Section 7.7.

7.9 Registering the functional unit's output

Table 7.7 shows the energy for several architectures broken down into core, crossbar, and interconnect energies. The crossbar energy is the energy from all of the components in the cluster's crossbar and the core energy is the energy from all other components in the cluster. The majority of energy consumed in the architectures with shared register blocks comes from the crossbar in the cluster. Two techniques for reducing crossbar activity are registering the outputs of functional units and grouping short-term storage with the functional units. These techniques are explored in this section and Section 7.10, the results of which are shown in the fourth and fifth column of Table 7.7, respectively.

Adding a register on the output of the functional unit reduces crossbar activity, and is very successful when combined with internal feedback, reducing the area-energy product by 11%. However, unlike input retiming (which is optional) the output register cannot be bypassed and adds a cycle of latency to all values produced by the functional unit.

	Shared Register Blocks					Private Register Blocks	
Energy	Base 7.5	Opt. 7.6	Alt. 7.7	Opt. 7.8	Opt. 7.9	Opt. 7.10	Opt. 7.11
Core	13	10	10	11	11	11	11
Crossbar	33	30	29	26	23	20	18
Interconnect	31	22	21	22	18	17	18

Table 7.7: Distribution of energy consumption over the benchmark suite, for several architectures specified in Table 7.6. Energy is split into three categories: core, crossbar, and grid interconnect energies. The core energy encompasses all components in the cluster except for the cluster's crossbar. The crossbar category includes the components in the cluster's crossbar. Finally, the interconnect category contains all components outside of the clusters. Architectures are split into groups based on the distribution of register blocks. Energy is reported in μJ.

Registering the outputs of the functional units splits the architecture's critical path and could lead to a shorter clock period. However, to isolate the impact of the registered outputs on energy consumption from the changes induced by a different clock period, the critical path of the architecture was not reduced for these tests. The output register is shown in Figure 7.10; it is dynamically enabled so that garbage values from inactive cycles do not leak onto the cluster's crossbar. Figure 7.10 also shows that distinct output registers exist on both the internal feedback path and the path from functional unit to the crossbar. These are split to ensure that purely internal values do not have to propagate to the crossbar.

Figure 7.7-IX shows that registering the output of the functional unit improves the area-energy product to 0.67× the baseline, when using a single rotating register file block. The output register provides two benefits: 1) values local to the functional unit are not transmitted to the crossbar, and 2) values have one less register to traverse outside of the functional unit. Both of these benefits can reduce the number of times that a value traverses the crossbar. Furthermore, fewer values use the register blocks and large register files, both of which consume more energy than the single output register. It is worth noting that this architecture used a single rotating register file block because the tools sometimes failed to

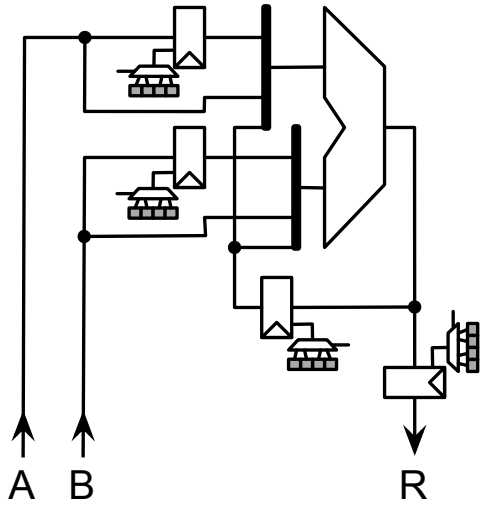


Figure 7.10: Functional unit with registered output and internal feedback.

find a solution for some applications when using the best architecture from the experiment in the previous section (Section 7.8), which had no register blocks. This highlights a pitfall of registering the functional unit's output: it reduces the slack between producers and consumers, and thus limits the number of legal routing paths. As a result the router has less flexibility and fails to find a solution more often. This was especially true when testing variants of the previous architectures (in Section 7.5, 7.6, and 7.7) that lacked the internal feedback path. When registers were included on the outputs of the functional units the routing failure rate was significantly higher than architectures without the output register.

7.10 Grouping register blocks with functional units

Instead of associating the register blocks with the cluster, the register blocks can be grouped with each ALU. This provides more local, private storage than just a feedback path, at the risk of increasing the architecture's resource fragmentation. For this experiment, we remove the register blocks that were attached to the cluster crossbar and add a register block to each functional unit, as shown in Figure 7.11. This increases the number of register blocks over the best results in the previous section, but decreases the crossbar size and traffic.

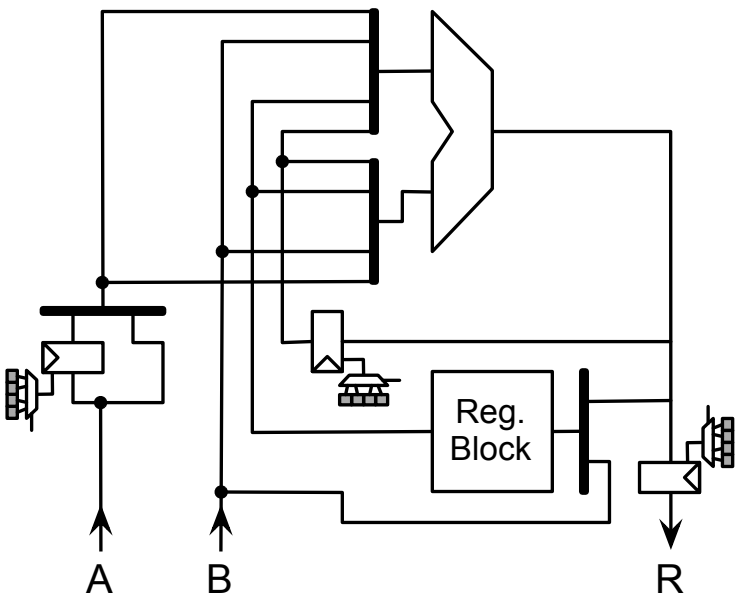


Figure 7.11: Grouping register block with ALU.

Figure 7.7-X shows the detailed area, energy, and area-energy product for each style of private register block. Using a rotating register file for the register block reduced the area-energy product to 0.64× the area-energy of the baseline architecture and 0.94× the area-energy of the optimized architecture from Section 7.9 and Figure 7.10. To minimize the increase in registers used for the register blocks we also tested designs that had only two registers per local register block for architectures with rotating register files and shift registers, shown as RRF2 and SR2 the graphs. The results show very little difference, indicating that reductions in area are offset by increases in energy, as values are forced to use more expensive resources. Furthermore, we tried a design with eight registers per local rotating register file (shown as RRF8) to see if the increased capacity would improve spatial locality. However, on average it consumed as much energy as the 4-entry rotating register file design and the increase in area led to a worse area-energy product overall. We observe that shift registers perform similarly to rotating register files, but that both the retiming chain and traditional register file perform worse. Rotating register files are a preferable design choice because they enable the optimization of storing local constant values, which

is detailed in the next section.

7.11 Supporting private constant values

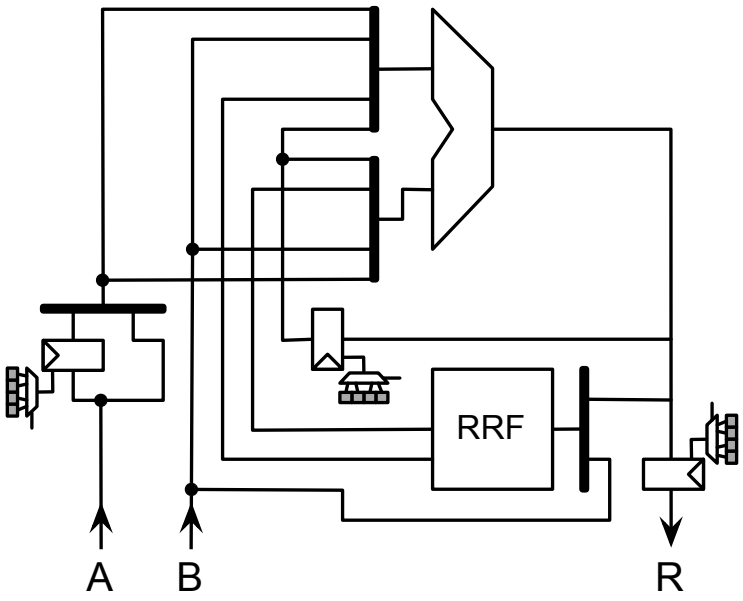


Figure 7.12: Supporting constant in local register files.

In the architectures tested thus far, constants that can not be techmapped into the immediate fields of operator opcodes are stored in the large register files, since constants are in some ways the ultimate in long-lived values. In this experiment, we study the effect of storing constants in the ALU's private register blocks when implemented using rotating register files. Constants are stored by partitioning the register file into RAM and ROM regions at configuration time. Partitioning is achieved by selectively disabling the addition of the wave counter to the upper entries of the register file and changing when the register renaming wraps high register names back to low names. This is accomplished by: 1) adding a small amount of comparison logic that prevents the wave counter from being added to the register number if the configured register number is outside of some range, and 2) changing the limit value used to perform the modulo math when the wave counter is added to each register name. SPR uses this comparison logic by calculating the number of constant values

that are stored in the rotating register file, reserving the upper portion of the register file for them, and then programming a register limit into the comparison logic of each register file. This register limit indicates the highest register that will have its address rotated during execution, and how the register names are wrapped.

We need to increase the register block ports to 2 read and 1 write (Figure 7.12), but we found that now we can also reduce the number of large register files in the cluster from two to one. Since storing many constant values in the private register file may force other values that could be local into the more expensive large register file, we tested architectures with 4 and 8 registers per register block. Furthermore, by removing one of the large, long-term storage shared register files, values with intermediate and long lifetimes will make more use of the private register files, in which case the additional capacity may be beneficial.

The last group of Figure 7.7-XI shows that supporting constants in a rotating register file results in an area-energy product that is 0.96× the area-energy of an architecture without private constants. The area-energy for an 8-entry register block is 0.61× the area-energy of the baseline architecture. The architecture with 8-entry register blocks consumed less energy than the one with 4-entry register blocks, showing that the increased capacity was able to keep values local, and highlights the benefit of using a medium sized rotating register file for storing values with medium length lifespans. The design with a 4-entry register block performed noticeably worse than the 8-entry design and worse than the best result from Section 7.10, which had a 4-entry rotating register file register block without support for private constants.

Our previous experiments detailed in [81] showed that both 4- and 8-entry private register blocks had nearly the same performance as each other when tested without a register on the output of the functional units. The difference in experimental results between [81] and this test stems from the reduction in average routing slack between producers and consumers due to registering the functional unit's output. This reduction in average slack forces short-lived values to take the most direct routing path between producers and consumers, which means that medium- and long-lived values require more flexibility to route around the more constrained short-lived values. Doubling the number of private registers from 4 to 8 increases the local routing flexibility, particularly as constants populate the register blocks.

This increased flexibility provides a larger benefit when registering the functional unit's output than previous experiments in [81], which lacked registered outputs. Furthermore, the additional flexibility is necessary to take advantage of the reduction in area-energy product gained by registering the output of the functional units. Overall, the combination of registered outputs on the functional units and local, private rotating register files reduced the area-energy product to 0.89× the results from [81].

7.12 Observations

In addition to a register structure's impact on an architecture's area, delay, and energy, the ability of the CAD tools to make effective use of different types and locations of storage structures is another metric for quantifying their advantages and disadvantages. Two questions are 1) does the location / distribution of storage impact placement quality, and 2) are some structures easier to route than others.

7.12.1 Impact of grouping register blocks

One method for judging the quality of placement is to measure the degree of spatial locality between sources and sinks, where producers and consumers that are topologically closer to each other are an indication of a better placement. Locality in a spatial architecture is of particular importance when optimizing for energy efficiency because local structures typically consume less energy than global structures, and moving values across various interconnects typically consumes energy that is proportional to the distance traveled. The degree of spatial locality of a placement can be quantified by the number of values that have their source and sinks mapped to the same functional unit, cluster, or different clusters. For each application, placements are deemed better if they achieve the same II and have more local traffic, since this reduces inter-cluster and inter-functional unit communication, which consumes more energy than intra-functional unit traffic. For the tests in Section 7.10 the architectural-specific extensions to the placer indicated that there was more local storage grouped with each functional unit; therefore values that lived for multiple cycles could stay local to a functional unit when it produced and consumed the value. As a result, the

placement cost for producer-consumer pairs on the same functional unit was lower than for producer-consumer pairs on different functional units.

Table 7.8 shows both the number of values and their average latency for the following producer-consumer pairs: extra-cluster, intra-cluster, and intra-functional unit. These results are given for the architectures from Sections 7.9, 7.10, and 7.11. Adding local, private storage allows values with an intermediate lifetime to stay local to a single functional unit, which reduces their placement costs. As a result, grouping the register blocks in Section 7.10 increases intra-FU traffic by 4% and reduces inter-FU traffic by 4%. This also reduces the mean latency of all values by 7%. By supporting private constant values in the local rotating register file, intra-FU traffic increases by 38% and inter-FU (intra-cluster) traffic decreases by 33%. It is interesting to note that this increase in intra-FU traffic is accompanied by a decrease in average intra-FU latency. This is counter-intuitive because the added local storage is intended to make it cheaper for values with intermediate lifetimes to have their producer-consumer pair in the same functional unit. This reduction stems from the migration of short-lived constant values from the large cluster-wide register files to the local register files. Additionally, the average inter-FU latency increases as there are fewer short-lived values.

7.12.2 Improving routability

The routability of an architecture can be measured by looking at how often SPR is able to find a legal routing solution given a valid placement and fixed set of parameters that controlled how hard it would try. The architectures with pipeline registers for input retiming in Section 7.5 and Section 7.6 failed to route 27% of the tests due to congestion. Switching to enabled registers for input retiming in Section 7.7 reduced the failure rate due to congestion to 3%, and adding a feedback path in Section 7.8 reduced the rate to 1%. As discussed in Section 7.9, registering the output of the functional units made the architectures somewhat less routable, and the failure rate went up to 6%. Chaining the register block with the functional unit reduced the energy consumed in Section 7.10, and reduced the failure rate to 4%. Finally, while allowing constant values to occupy local register files provided a substantial

Arch. of Short-term Storage	Mean Latency of all values	Extra-cluster		Intra-cluster / Inter-FU		Intra-FU	
		# Values	Latency	# Values	Latency	# Values	Latency
§7.9 - Shared	7.0	292	18.5	290	7.0	232	4.6
§7.10 - Grouped	6.5 (93%)	295 (101%)	17.7 (95%)	278 (96%)	6.2 (89%)	242 (104%)	4.2 (92%)
§7.11 - Grouped (Private Imm.)	7.0 (100%)	300 (103%)	18.0 (97%)	194 (67%)	7.8 (112%)	321 (138%)	3.5 (76%)

Table 7.8: Distribution of values and average latency for grouped architectures from Sections 7.10 and 7.11 versus shared architectures from Section 7.9. Percentages are with respect to the shared architectures and latencies are reported in clock cycles.

reduction in energy in Section 7.11, the architecture suffered from more congestion as the failure rates increased to 8%. The higher failure rate stems from the increased competition between constant values and runtime values in the local storage. Overall, enabled registers for input retiming and local feedback paths provided the most substantial improvements in routability; meanwhile registered outputs for functional units and support for local constant values reduced energy consumption but harmed the architectures routability.

7.13 Conclusions

This chapter considered numerous different mechanisms for storing values in a CGRA. Our experiments show that:

- Rotating register files consistently outperform other structures for storing medium- to long-lived values. Although rotating register files are more expensive in terms of area than other options, the improved energy more than makes up for their expense.

- The energy efficiency of enabled registers is well worth their configuration memory costs for short-lived values.

- Registering the functional unit's outputs provides a notable improvement in area-energy but requires increased routing flexibility to allow medium- and long-lived values to route around short-lived values.

- While rotating register files and distributed registers often provide the same area-energy product, distributed registers are more area-efficient, while rotating register files are more energy-efficient.

- Normal register files are largely ineffective in modulo-scheduled architectures, since they cannot hold values longer than II cycles, yet have much greater overheads than other structures for short-lived values.

- It is better to keep values close to the functional units using local feedback and private register blocks, than to use register blocks that can be shared among functional units

but require values to transit the cluster crossbar.

Overall, these observations provide an architecture that is 1% better in area, 38% better in energy, and 39% better in area-energy product than a reasonable baseline architecture.

Chapter 8

SPECIALIZING FUNCTIONAL UNITS

Functional units are the core of spatial accelerators. They perform the computation of interest with support from local storage and communication structures. Ideally, the functional units will provide rich functionality, supporting operations ranging from simple addition, to fused multiply-adds, to advanced transcendental functions and domain-specific operations like add-compare-select. However, the total opportunity cost to support the more complex operations is a function of the cost of the hardware, the rate of occurrence of the operation in the application domain, and the inefficiency of emulating the operation with simpler operators. Examples of operations that are typically emulated in spatial accelerators are division and trigonometric functions, which can be solved using table-lookup based algorithms and the CORDIC algorithm.

One reason to avoid having direct hardware support for complex operations in a tiled architecture like a CGRA is that the expensive hardware will typically need to be replicated in some or all of the architecture's tiles. Tiled architecture are designed such that their tiles are either homogenous or heterogenous. Homogenous architectures replicate a single tile, which simplifies the challenges faced by the placement and routing CAD tools, but would replicate a complex operator in all tiles. Furthermore, homogenous architectures are easier to design and optimize their physical layout. Heterogenous architecture can minimize the number of times that specialized hardware will be replicated by building the array from a small set of repeated tiles. However the greater the diversity of tiles used, the harder it is to balance and optimize the physical resources in the array. Additionally, this can lead to an increase in the complexity of the CAD tool's search space, which may degrade the quality of mapping an application to the architecture or it may make the tools harder to implement efficiently. Therefore supporting large, complex, and infrequent operations is in tension with a simple homogenous tile design that has the same set of devices in all tiles.

As a result, CGRAs try to support a rich set of operations with the smallest possible set of hardware devices.

The experiments in this chapter use two types of tiles in the architecture: core tiles and edge tiles, which were presented in Section 7.4. To minimize the design complexity, these tiles are homogenous in the set of functional units that are supported, and they only differ in their peripheral logic. Specifically, the edge tiles have stream I/O ports and the core tiles have an additional embedded data memory. This chapter focuses on paring down the CGRA's clusters from having four word-wide universal functional units to a design that is specialized for the intended application domains.

Prior to optimizing the set of functional units in the CGRA cluster, it is important to know how frequently each type of operator appears in the application domain, and the relative costs of hardware structures. The distribution of operations for the Mosaic benchmark suite is discussed in Section 8.1, and relative hardware costs of different functional units is presented in Section 8.2.5. In general, we know that the most complex operations typically found within the application domain (listed in order of decreasing frequency) are multiplication, followed by division, and then modulus. Multiplication is both fairly frequent and not too expensive to implement in dedicated hardware. Division (or modulus) with a arbitrary divisor is not very frequent and it is expensive to implement directly; the more common case has a divisor that is a power of two or other constant value. These special cases of division and modulus are important because they can be strength-reduced to simpler operations as described in Section 4.2. General-purpose division or modulus can be reasonably emulated using either a table-lookup based algorithm or a less efficient sequence of compare and subtract operations. Therefore it is uncommon to directly support division or modulus in a CGRA, and the Mosaic architecture does not include a dedicated hardware divider or modulus unit.

Two other design considerations for functional units that support arithmetic operations are arithmetic precision and fixed versus floating point arithmetic. Functional units and their accompanying datapaths are typically designed to support a single (maximum) width arithmetic precision. Support for fracturable arithmetic designs, that can operate on multiple (logical independent) sub-words in parallel, is common in both FPGAs and the vector

processing units in sequential processors. For example, many modern processors have SIMD or vector units that support up to a single 64-bit (or 128-bit) arithmetic operation, or as many as eight 8-bit (or sixteen 8-bit) parallel sub-word operations. If there is a mismatch between an application's desired amount of arithmetic precision and the hardware's arithmetic precision, it is typically the responsibility of the programmer to make the application accomodate the architecture. Support for fixed versus floating point arithmetic is a similar issue, where the application programmer is responsible for resolving any mismatches between the demands of the application and the capabilities of the architecture. The hardware required to implement each type of fixed and floating point operation is distinct, and floating point operations require more resources. The Mosaic architecture supports 32-bits of arithmetic precision (*i.e.* 32-bit words) and fixed point arithmetic. Support for either single or double width precision floating point arithmetic would not have too much impact on the previous results for the interconnect and storage distribution, but would dramatically change the relative costs of specialized functional units. In particular, the relative cost for floating point addition and multiplication are the opposite of fixed point addition and multiplication; floating point addition is much more expensive to implement that floating point multiplication.

8.1 Evaluating operator frequency in the benchmark suite

This experiment uses the benchmark suite as detailed in Section 5.1. The characteristics of each application are listed in Tables 8.1 and 8.2. Similar to the previous experiments, Table 8.1 lists the minimum initiation interval (II), the number of operations in the application, and the size of the targeted CGRA. As before, each application and target architecture were sized so that no resource in the array is utilized more that 80%. Table 8.2 enumerates the tuning knobs and their settings for each application. Table 8.3 shows the breakdown of operations within the benchmark suite, as well as an average frequency for each class of operation. Operations are divided into groups, based on how those operations map to different hardware devices. The groupings of operations are self-explanatory, except for the difference between simple and complex arithmetic operations. Simple arithmetic operations are addition, subtraction, and negation. While complex operations are multiplication and

Application	Min. II	DFG Operations	Memory Arrays	Stream IO	Constant Values	# Clusters	CGRA Grid Size
64-tap FIR filter	2	136	0	2	130	20	4x5
240-tap FIR filter (Banked Mem)	3	112	48	2	77	42	6x7
2D convolution	3	352	12	4	200	36	6x6
8x8 Blocked Matrix multiplication	3	438	0	24	27	49	7x7
K-means clustering (K=32)	3	400	32	35	43	42	6x7
Matched filter	3	126	30	15	44	30	5x6
Smith-Waterman	5	462	11	8	101	30	5x6
CORDIC	2	140	0	8	23	20	4x5
8×8 Motion estimation [63]	4	487	16	20	245	36	6x6
PET Event Detection	4	551	28	16	219	42	6x7

Table 8.1: Benchmark Applications and Simulated Architecture Sizes

Application	Tuning Knobs
64-tap FIR filter	Num. Coefficients = 64
240-tap FIR filter (Banked Mem)	Num. Coefficients = 240, Num. Memory Banks = 24
2D convolution	Kernel Size = 7x7, Vertical Stripe Width = 15, Num. Parallel Pipelines = 2
8x8 Blocked Matrix multiplication	Blocking Factor = 8x8
K-means clustering (K=32)	Num. Clusters = 32
Matched filter	Num. Parallel Channels Processed (Block Size) = 2, Num. Signatures = 13, Num. Channels = 16
Smith-Waterman	Stripe Width = 11
CORDIC	Num. CORDIC iterations = 3
8×8 Motion estimation [63]	Image Block Size = 8x8, Num. Parallel Searches = 8
PET Event Detection	Num. Parallel Datasets = 8

Table 8.2: Tuning Knob settings for Benchmark Applications

fused multiply-add or multiply-subtract operations. Table 8.4 shows which operations map to each available hardware device.

In addition to grouping operations by type, it is valuable to categorize applications by the distribution of operation groups that are performed. These categories are then used to examine how similar applications perform on architectures with specific functional unit distributions. Table 8.3 shows how applications can be broadly categorized, by the dominant type of operations, into the following groups.

- Balanced - relatively even mix of all operation types
- Simple-dominant - $\geq$ 60% operations are simple arithmetic or logic / comparison operations
- Complex-dominant - $\geq$ 40% operations are complex arithmetic operations
- Select-dominant - $\geq$ 60% operations are select operations

The criteria for these categories was determined both by the importance of complex arithmetic and select operations, and by identifying natural groupings within the benchmark suite. Complex arithmetic operators are important because they are the most complicated operations that are directly supported, they require multiple cycles to execute as detailed in Section 8.2.2, and have the most expensive hardware. Select operations are very simple and very frequent; they also are primarily used for data movement rather than computation. Identifying categories that had at least 60% of their operations from a related set of operation groups (or 40% for complex arithmetic) provides a reasonable partitioning of the benchmark suite.

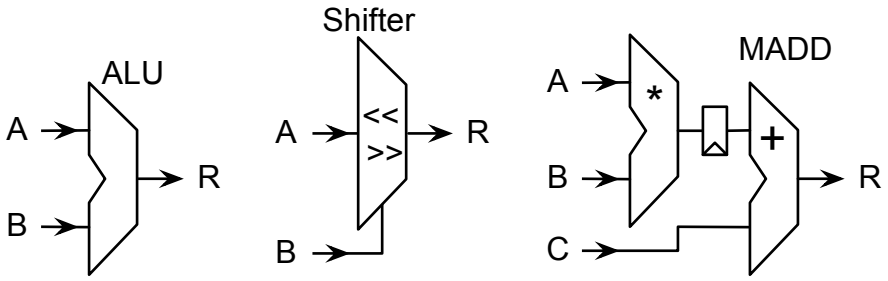
These applications were compiled with the enhanced loop flattening variant of the Macah compiler and the Mosaic techmapper set to fuse back-to-back multiply-add operations into a single MADD operator. The profile of the benchmark applications shows two key things: select operations are extremely common, while complex multiply or multiply-add operations rarely exceed 25% (and never 50%) of operations in the suite. Select operations provide

Benchmark	Category	32-bit Logic & Comp.	Simple Arith.	Complex Arith.	Select	Arith. & Logical Shifts	1-bit Logic & Comp.
FIR	Complex	0.7	0.7	47.1	47.1	0.7	3.7
FIR (Banked)	Complex	3.6	23.2	42.9	24.1	0.9	5.4
Convolution	Select	3.4	1.1	27.8	63.4	0.6	3.7
Dense matrix multiply	Select	5.5	0.9	14.6	61.2	14.6	3.2
Motion Estimation	Simple	40.9	18.5	7.2	18.1	3.9	11.5
Smith-Waterman	Simple	39.8	21.0	0.2	36.8	0.0	2.2
K-Means Clustering	Simple	34.5	25.3	0.0	33.5	0.0	6.8
CORDIC	Simple	15.0	26.4	0.0	39.3	7.1	12.1
PET Event Detection	Simple	33.8	8.9	2.9	46.6	5.8	2.0
Matched filter	Balanced	9.5	7.1	21.4	22.2	25.4	14.3
Average		24.4	13.1	11.0	41.0	5.0	5.5

Table 8.3: Frequency of operations, reported as percentage, in the benchmark suite. Benchmarks are categorized by the dominant type of operation.

dynamic data steering within the statically scheduled CGRA, typically staging data for subsequent computation. This makes it attractive to co-locate select hardware with other functional units, as explored in Section 8.2.

One key contribution to the relatively high frequency of select operations is that the combination of data reuse and the more complex control flow afforded by Macah's enhanced loop flattening increased the number of select operations relative to the number of arithmetic operations. The two key programming patterns that generate select operations in Macah are `if-then-else` statements and multiple sequenced and nested loops. These patterns cause select operations to be generated to either merge values generated on multiple control paths, or to re-initialize variables when loops are re-entered. Macah's enhanced loop flattening algorithm allows the compiler to generate efficient dataflow graphs from applications with sequenced and nested loops. This allows programmers to create "richer" and more complex variants of an application's algorithm, which in turn lead to a larger percentage of select



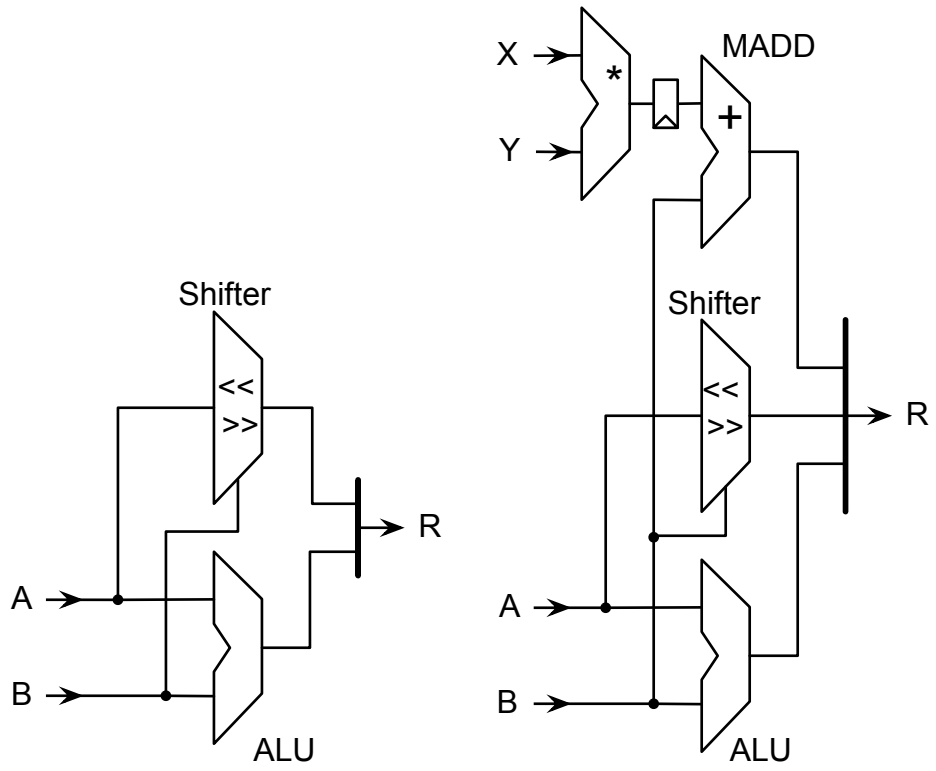
(a) ALU FU (b) Shifter FU (c) MADD FU

Figure 8.1: Block diagram of primitive functional units. Note that the MADD FU is a pipelined two-cycle operation, and that the register is actually in the multiplier's partial product tree.

operations in the application's operation mix. As a result, many of the benchmarks that were hand flattened for experiments in Chapters 6 and 7 were rewritten to be functionally equivalent, but used multiple sequenced and nested loops. Note that the combination of more complex data movement and fusing back-to-back operations is the main source of the differences between the number of DFG operations present in this and previous versions of the benchmark suite.

8.2 Designing the Functional Units

Functional units (FUs) are the devices that perform actual computation for an application. In sequential processors, digital signal processors, and dedicated ASICs they range from simple adders to dividers and even specialized hardware like an add-compare-select function or butterfly unit that are tailored for the Viterbi and Fast Fourier Transform algorithms, respectively. Tiled spatial accelerators typically eschew embedding the more complex and esoteric functional units, in favor of a simpler repeated tile, focusing on a range that spans simple adders to complex fused multiply-add units. In previous experiments, we have modeled the functional units for the word-wide datapath as idealized devices that can execute all operations within a single cycle. However, the diversity of supported operations make that idealized construct untenable. In particular, the hardware required for performing



(a) S-ALU (Shifter-ALU) FU (b) Universal FU

Figure 8.2: Block diagram of compound functional units.

multiplication is significantly larger and more complex than what is necessary for any other supported operations. In this experiment we explore the following primitive and compound functional units to replace the idealized function unit for the word-wide datapath: ALU, shifter, MADD, S-ALU FU, and universal FU. A short description of each functional unit explored in the chapter is presented below. Block diagrams for each word-wide functional unit are shown in Figures 8.1 and 8.2. Note that the single-bit datapath uses a 3-input lookup table for its functional units. The 3-LUT is very efficient in terms of flexibility versus area and energy and is included here for completeness, but is not considered for specialization.

- ALU - arithmetic and logic unit, with support for select

- Shifter - logarithmic funnel shifter
- MADD - 2-cycle fused multiply-add
- S-ALU - compound unit with shifter, ALU, and select
- Universal - compound unit with MADD, shifter, ALU, and select
- 3-LUT - 3-input lookup table

8.2.1 Primitive functional units

This work uses the ALU, shifter, and MADD as primitive functional units. The primitive functional units provide independent hardware devices with (mostly) disjoint functionality, the exception being the overlap between the ALU and MADD with respect to addition. The operations performed by each functional unit are enumerated in Table 8.4. The three main groups of operations that can share hardware resources are: 1) basic arithmetic, logic, and comparisons; 2) shifting; 3) multiplication. These groups are supported by the ALU, shifter, and MADD FUs respectively.

With these groupings, it is not immediately obvious if a select operation should use independent hardware or share with a device like the ALU. The primary reason to co-locate the select and ALU hardware is that both operations are very common in the benchmark suite's instruction mix, making it desirable to keep data local to a single logical functional unit. If the select hardware were independent, then sequences of operators that included select operations would be forced to spread out between multiple functional units, which would typically involve a trip across the cluster's crossbar. Since traversing the crossbar consumes a non-trivial amount of energy, it is likely that we would want a compound device that performs both ALU and select operations if the select hardware was independent. The tradeoffs in energy optimization techniques for compound versus a multi-function primitive device is discussed in Section 8.2.4; in summary, a compound functional unit with independent ALU and select hardware will probably be unable to statically disable either component given the frequency of both operations. Without this power savings technique, a

Operation Class	Operation	ALU	Shifter	MADD	S-ALU	Universal
Arithmetic	ADD	√		√	√	√
	SUB	√		√	√	√
	NEGA	√			√	√
	MUL			√		√
	MADD			√		√
	SHL		√		√	√
	SHR		√		√	√
	ASHR		√		√	√
Comparison	LT	√			√	√
	LTE	√			√	√
	GT	√			√	√
	GTE	√			√	√
	EQ	√			√	√
	NEQ	√			√	√
Bitwise Logic	NOTA	√			√	√
	AND	√			√	√
	OR	√			√	√
	XOR	√			√	√
Select	SELECT	√			√	√
	$\overline{\text{SELECT}}$	√			√	√

Table 8.4: List of word-wide operations, split into operation classes, and showing which device supports which operation.

compound functional unit is likely to consume more energy than a multi-function primitive device because the multi-function primitive device allows the hardware synthesis tools to share physical resources. Therefore we decided to create a primitive functional unit that could perform either an ALU or select operation.

One of the design considerations for the functional units was that most of them should complete within a single clock cycle. The notable exception for this is the multiplier hardware, which is significantly slower than the other devices and should be pipelined to balance its critical path with other devices. The pipelining is done by splitting the partial product addition between two cycles. As a result, all complex operations require two cycles to execute – scheduling of these operations is detailed in Section 8.2.2. When pipelining a multiplier, it adds very little overhead to include an additional addend in the last stage of the multiplier. This leads to the common optimization of transforming a multipier into a MADD unit that supports fused multiply-add operations.

8.2.2 Scheduling Pipelined Operations

Most operations in the Mosaic architecture are single-cycle operations that permit time for operands and results to be routed from and to nearby registers within a single clock cycle. The exceptions are the complex arithmetic operations, where the hardware has a long critical path and cannot execute in a single-cycle at reasonable clock frequencies. Therefore the MADD hardware is pipelined and split into two stages. It is crucial that the hardware for these operations is fully pipelined to allow high-performance execution and back-to-back scheduling of these complex operators. If the MADD hardware were not fully pipelined, indicating that there is a structural hazard that prevents back-to-back scheduling of operations, not only would performance suffer but it would present a significant challenge to the tools.

Fully pipelined multi-cycle operations are supported using a trick that allows them to be scheduled and placed as if they were single-cycle operations. The trick used for the current version of SPR is to schedule, place, and route the last stage of the multi-cycle operation and to force the router to route the operands to earlier stages of the hardware, so that they

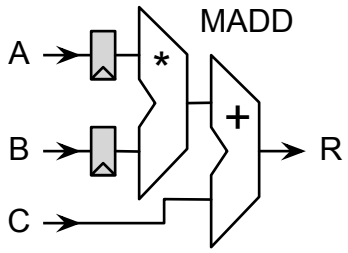


Figure 8.3: Logical diagram of MADD FU with virtual registers shown in grey.

arrive sufficiently early. Looking at the MADD hardware in Figure 8.1(c), operands A and B have to arrive one cycle before operand C. The router can be coerced into making the A and B operands arrive early by having forced virtual registers on those inputs. These registers, are shown in grey in the logical representation of the MADD hardware in Figure 8.3.

Table 8.5 shows how these virtual registers change when operands arrive with respect to their scheduled operations. The left side of this example schedule shows how SPR schedules the MADD operation to execute in a single cycle, with the operands arriving at the MADD unit and the result being produced in the same cycle. The right side shows when the operands actually arrive, due to the forced virtual registers and the activity of the hardware pipeline in each cycle. Note that the A and B operands actually arrive one cycle before they were scheduled to arrive, and the multiplier pipeline starts computing partial products immediately. The multiplication and addition are completed in the second cycle of operation. Note that the current methodology works well for devices that have operands arriving in multiple cycles. For pipelined hardware where all operands arrive in the same cycle an alternative trick also exists, where the output is pipelined with virtual registers rather than the input. SPR is informed about the need for additional latency on particular inputs or the output of a device (to accomodate the mandatory registering) by the architecture-dependent portion of the distance function, which is discussed in Section 4.3.1.

The trick of scheduling pipelined multi-cycle operations as if they only required a single cycle makes it possible to support both single- and multi-cycle operations on the same device

Time	Scheduled Operation	Post-Register Input Ports A	B	C	Output Port R	MADD Activity	Pre-Register Input Ports A	B
0						$*_0$	a_0	b_0
1	$(*+)_0$	a_0	b_0	c_0	$(a_0 * b_0) + c_0$	$*_1$ & $+_0$	a_1	b_1
2	$(*+)_1$	a_1	b_1	c_1	$(a_1 * b_1) + c_1$	$*_2$ & $+_1$	a_2	b_2
3	$*_2$	a_2	b_2	0	$(a_2 * b_2) + 0$	$+_2$		

Table 8.5: Example schedule for a MADD unit. Subscripts on operators and operands indicate iteration. Note that the scheduled operation column shows when SPR has scheduled the logical operation, and the post-register input ports columns indicate when SPR schedules operands to arrive. After the double line, the MADD activity column shows the activity of the hardware's pipeline and the pre-register input ports show when the A & B operands actually arrive at the hardware.

without mandatory stalls of the hardware when switching between operation types. The requirement for supporting back-to-back scheduling of single and multi-cycle operations is that any registered input operands for the device cannot share ports with non-registered input operands. By providing independent input ports for the registered operands, those operands can arrive sufficiently early with respect to when the operation is scheduled. Table 8.7 and Section 8.2.3 show an example of this in the construction of the universal functional unit, which provides separate inputs for the multiply stage of the MADD device.

It is possible to efficiently support both single- and multi-cycle operations within the same device and share both the input and output ports; however, this would require additional sophistication in the CAD tools. One example of the complexity for the CAD tools, is that back-to-back operations can be scheduled if they are of the same type, single- or multi-cycle. However, when switching from executing single-cycle operations to multi-cycle operations, the scheduler has to allow a gap before the multi-cycle operation to fill the device's pipeline. This constraint avoids over-using the shared inputs. An example of adding this gap to solve the problem is shown at time 2 in Table 8.6 for a fictional device that can perform either a single-cycle addition, or a two cycle multiplication.

Time	Operation	Input Ports A	Input Ports B	Output Port R
0	+	a_0	b_0	$a_0 + b_0$
1	+	a_1	b_1	$a_1 + b_1$
2		a_2	b_2	
3	*	a_3	b_3	$a_2 * b_2$
4	*	a_4	b_4	$a_3 * b_3$
5	*			$a_4 * b_4$
6	+	a_5	b_5	$a_5 + b_5$

Table 8.6: Example of scheduling operation execution and operand arrival on a fictional device that performs a single-cycle addition and a two cycle multiplication.

8.2.3 Compound functional units

The S-ALU and universal compound functional units combine multiple primitive functional units into a single logical group. It is notable that while a compound functional unit could support multiple concurrent operations internally, it lacks the input and output ports necessary to supply and produce multiple sets of operands and results. Sharing the input and output ports mitigates the need to increase the size of the cluster's crossbar as more functionality is added to each functional unit. The cost for adding a port to the crossbar is detailed in Section 8.2.5, but is approximately the same as the hardware for an ALU. In addition to the cost of adding a port to the crossbar, each input and output of the functional unit requires some peripheral hardware in the processing element (PE) to stage operands and results. The costs of these peripherals are discussed in Section 8.2.5.

By treating compound FUs as one logical device, the placement tool will only map a single operation onto the functional unit per cycle. This ensures that an architecture maintains the same number of concurrent operations per cluster when mixing primitive and compound functional units. Two other advantages of maintaining the same number of concurrent operations per cluster are 1) the external support resources in the cluster do not

		Input Ports				Output Port
Time	Operation	A	B	X	Y	R
0	+	a_0	b_0			$a_0 + b_0$
1	−	a_1	b_1	x_1	y_1	$a_1 - b_1$
2	*			x_2	y_2	$x_1 * y_1$
3	*+		b_3			$(x_2 * y_2) + b_3$

Table 8.7: Example of universal functional unit execution and operand arrival.

have to be scaled as capabilities are added to each functional unit, and 2) it is easier to test and make comparisons between two architectural variants.

The universal FU, Figure 8.2(b), is a realistic implementation of the idealized functional unit that was used in prior experiments. It is composed of all three primitive devices, implementing all word-wide operations, only one of which is ever active in a given cycle. This allows it to share a single output port, which reduces the cluster crossbar size by ∼1.4% per output port saved, as shown in Table 8.10. Ideally, the universal FU would only require three input ports, but it uses four input ports to allow proper staging of operands to the multiply stage of the MADD unit. The example schedule in Table 8.7 shows the interplay of operands for single-cycle operations and the complex, pipelined, arithmetic operations. Each row of the table indicates what operation is being executed for each time step, which operands are required, and what result is produced. As alluded to earlier, the universal FU is only able to produce a single output value at a time, but the first part of the pipelined MADD operation can be overlapped with other operations, as shown in cycle 1 of Table 8.7. A multiply operation is scheduled for cycle 2, but its operands arrive on the X and Y ports in cycle 1, and the calculation starts in cycle 1.

8.2.4 Minimizing signal activity

One challenge in designing the functional units is determining how to minimize the signaling activity of inactive hardware devices. This is particularly true for the compound

functional units, but also for the ALU when executing a select operation or the MADD when performing just a multiplication. The challenge is that both compound and multi-function hardware devices share a set of inputs and instruction operation codes (opcodes). In a naive design, this sharing leads to signal activity in all parts of the hardware, including inactive subsystems.

One method for reducing energy consumption is to power gate all parts of the device that are inactive, which can be done on a per application or on a cycle-by-cycle basis. For our compound functional units, if a device or part of a device is never used by an application, it is statically power gated. Currently our functional units are not designed to use cycle-to-cycle power gating because a recent small-scale experiment detailed in [82] shows that a subcomponent would have to be inactive for at least 2-3 cycles to make power gating effective. While this is an interesting area of future research, given the low II of our applications we believe that it is of marginal benefit.

Due to the overhead of power gating it is typically not feasible to power gate specific parts of a multi-function device such as the ALU with select support or the MADD. Instead, we reduce energy consumption in these devices by feeding tailored constant values into the inactive parts of the design to minimize signal transitions. Another power saving option that we apply to the functional units is to hold the last valid instruction on cycles when the FU is inactive. This reduces the energy consumed in switching the logic's functionality and, when the input operands are held constant by resources outside of the functional unit, then there is minimal dynamic energy consumed.

Given that the external logic cannot be guaranteed to keep a functional unit's inputs constant when the device is idle, another technique is to place latches on each input. It turns out that adding input latches to the primitive devices incurred up to a 20% overhead in power, and thus was not worth it in most circumstances. However, it did provide a net benefit to add latches inside of the ALU design to isolate the inputs to the adder hardware from the rest of the select and bitwise logic. This optimization provides a reduction in dynamic energy that is proportional to the distribution of arithmetic and comparison operations versus bitwise and select operations. In small scale tests this optimization showed a $\sim 30\%$ reduction in dynamic energy. An interesting exploration for future work would be

	Device's Metrics						Peripherals' Metrics		
Device	Area	Static Energy	Config. Energy	Critical Path	# Cfg. Bits	Datapath Ports (I/O)	Area	Static Energy	Config. Energy
ALU	1409	0.1	75.1	1.00	6	2/1	7215	7.0	223.6
Shifter	1358	0.1	75.1	1.00	6	2/1	7215	7.0	223.6
MADD	9968	1.4	50.1	1.00	4	3/1	9184	8.1	299.4
S-ALU	2832	0.2	87.7	1.05	7	2/1	7215	7.0	223.6
Universal	13017	1.8	98.4	1.06	8	4/1	11266	9.9	372.7

Table 8.8: Characteristics of functional units, including the number of word-wide ports to and from the crossbar and word-wide peripheral storage/routing resources. Area is reported in μm^2 and critical path in ns. Static and configuration energy was computed for a clock period of 1.93ns, and are reported in $fJ/cycle$.

to apply latches to the individual primitive devices within a compound functional unit.

In summary we used the following techniques to minimize energy consumption within the functional units:

- Static power gating of unused devices within compound functional units.
- Selectively driving constant values into MADD units to minimize signal transitions.
- Generate application configurations such that functional units repeat the last valid instruction on idle cycles.
- Latch internal operands for adder hardware in the ALU functional unit.

8.2.5 Comparison of Functional Units

To evaluate the tradeoff between flexibility and overhead for the functional units we examine several of their characteristics. Table 8.8 presents several metrics for both functional units and their word-wide peripheral resources, including: area, static energy, number of configuration bits, configuration energy, and the requisite number of word-wide crossbar ports. Note that the configuration energy includes both the static and dynamic energy of the configuration SRAM and associated logic, since the dynamic energy consumed per clock

cycle can be precomputed. The peripheral resources include the local register files, input retiming registers, and multiplexers. Table 8.9 shows the average and peak dynamic energy consumed when executing the different operators supported by each type of primitive functional unit. Additionally, Table 8.9 presents the dynamic energy cost per bit that changes on the cluster's crossbar.

One design implication of making compound functional units a single logical device is that they have only one output port and one set of peripheral resources. The average cost of each input and output port in the FU is shown in Table 8.10. The cost of an output from the FU to the crossbar is the combination of a crossbar driver and an input tap on each mux that reads a value out of the crossbar. There are approximately 31 of these taps that are the inputs to muxes in the crossbar, each of which can then transmit (*i.e.* read) a value out of the crossbar. On average, the cluster's word-wide datapath has 20 source devices and thus there are 20 channels in the crossbar. The cost of an input to the FU, from the crossbar, is a single, 20-input read mux in the crossbar (*i.e* one tap per crossbar channel). Since the functional units consume more values than they produce, there are fewer inputs to the crossbar than outputs from it, and thus the crossbar is not square. Therefore, the cost to add an output port to the functional unit (or attaching another device to the crossbar) is significantly more expensive than adding an input port, primarily due to the high number of loads within the crossbar. One advantage of using compound functional units instead of a larger number of primitive functional units is that it minimizes the number of output ports and peripheral resources required. The area and energy metrics for the each type of processing element (*i.e* functional unit plus peripheral logic) and their associated crossbar I/O ports are presented in Table 8.11.

8.3 Experiments

To explore the impact of specializing the functional units in a cluster we test several clusters built with different functional units. The baseline architecture was the optimized design from Chapter 7, with a private rotating register file in each functional unit and one cluster-wide large rotating register file. The four idealized functional units per cluster were replaced with universal functional units. Each test in this experiment replaced some of those four

Device	Dynamic Energy		
	Average per operation	Peak per operation	per bit
ALU	524	1643	N/A
Shifter	506	1157	N/A
MADD	13521	17924	N/A
Crossbar signal	N/A	N/A	171

Table 8.9: Dynamic energy cost for each primitve functional unit and bit transition on the crossbar, reported in fJ per operation and fJ per bit toggled respectively.

Device	Area	Percentage of Cluster Area	Static Energy	Config. Energy	Avg. # Loads
Output Port (to crossbar)	4186	1.4	2.5	97.5	31
Input Port (from crossbar)	2472	0.9	1.3	61.5	20

Table 8.10: Average area and energy cost for each input and output port attached to the cluster crossbar, in μm^2 and $fJ/cycle$ at an average clock speed of 1.93ns, respectively. The average number of loads for the output port is the number of read muxes and for the input port is the number of crossbar channels.

Processing Element (with FU)	Area	Static Energy	Config. Energy	Datapath Ports (I/O)
ALU	17754	12.2	519.2	2/1
Shifter	17703	12.3	519.2	2/1
MADD	30754	16.0	631.4	3/1
S-ALU	19177	12.4	531.7	2/1
Universal	38357	19.5	814.4	4/1

Table 8.11: Aggregating the cost of the functional unit, peripheral logic, and crossbar I/O ports from Tables 8.8 and 8.10. Area is reported in μm^2. Static and configuration energy was computed for a clock period of 1.93ns and are reported for $fJ/cycle$.

Test Number	Universal	S-ALU	ALU	Shifter	MADD
A	4	0	0	0	0
B	**3**	**1**	0	0	0
C	3	**0**	**1**	0	0
D	**2**	**2**	**0**	0	0
E	2	**1**	**1**	0	0
F	2	**0**	**2**	0	0
G	**1**	**3**	0	0	0
H	1	**2**	**1**	0	0
I	1	**1**	**2**	0	0
J	1	**0**	**3**	0	0
K	**0**	**3**	**0**	0	**1**
L	0	**2**	**1**	0	1
M	0	**1**	**2**	0	1

Table 8.12: Set of experiments with different functional units per cluster. Changes in experiments are marked in bold. Experiments B-J had at least one universal functional unit, and experiments K-M had dedicated MADD devices.

functional units with a more specialized device. Throughout the tests, each cluster in each architecture had four placable devices, and four output ports from functional units to the crossbar. The number of input ports per functional unit varied with its degree of specialization, but aside from the input ports' impact on the cluster crossbar, everything else about the cluster remained the same for each test.

Two design considerations that were followed during this experiment were 1) each cluster could perform all supported operations, and 2) the number of concurrent operations per cluster remained constant. The first of these design constraints is important to simplify the task of logically arranging the clusters within the computing fabric's grid topology. If some clusters did not support the full operation set of the CGRA, then it is very easy to construct scenarios where particular sequences of operations cannot map to the architecture with minimum latency. When that occurs on the application's critical loop-carried dependence, then that application is structurally unable map at its recurrence II. For example, if part of a critical loop is scheduled onto a cluster that does not support all operations in the loop, then the loop will be split across multiple clusters. This can lengthen the critical loop due to the added latency of inter-cluster communication. Therefore, with computational heterogeneity between the clusters, finding a physical floorplan that has the minimum set of poorly performing operation sequences would require a great deal of sophistication to do well.

The second design constraint is that each cluster has the same number of placeable devices in it, namely four for these experiments. This means that an architecture with four universal functional units will be more flexible and can execute more concurrent operations for some operator mixes than an architecture with one MADD and three ALUs. Maintaining the number of placeable devices ensures that each cluster supports the same number of total concurrent operations and has the same number of outputs from the functional units, and thus the same number of input ports on the cluster crossbar.

To specialize the functional units we looked at the frequency of operations in the benchmark suite. The first optimization is to reduce the total number of multipliers within the architecture because they are the most expensive units, require the most input ports, and multiplication and MADD operations only occur $\sim$11% of the time. Based on the frequency

of operations, it is reasonable to provision at most one multiplier per cluster. However, due to their importance within the application domain, architectures with two or three multipliers per cluster may provide a good solution. To pare down the number of concurrent multiply operations supported per cluster it is possible to either strip away the MADD device from universal FUs, or to replace universal FUs with dedicated MADD FUs. Stripping the MADD device from a universal FU results in either a S-ALU, or a plain ALU if shift support is also removed. Therefore, as the number of multipliers is reduced, architectures will have a small number of either universal FUs or MADDs, and the remaining functional units will be either be S-ALUs or ALUs. Note that the low frequency and relatively high cost of shift operations does not justify including a standalone shifter functional unit.

It is possible to consider architectures with either multiple MADD FUs or a mix of MADD and universal FUs. However, given the relative frequency of the MADD operations neither of these optimization paths offer a compelling advantage over the afore-mentioned architectures. Given the relatively infrequent occurrence of shift operations, a second optimization is to minimize the amount of shifter hardware in the cluster. The effectiveness of this optimization is explored by stripping the shifter away from the S-ALU, which results in S-ALUs being replaced with ALUs. The set of permutations that we explored is detailed in Table 8.12. The first ten designs, A-J, have one or more universal functional units; while the last three designs, K-M, use a dedicated MADD FU to support all multiplications.

Similar to previous experiments, each application to architecture mapping was performed multiple times with different random placement seeds. Experiments in previous chapters used 4 placer seeds, and shared an application's placement across multiple architecture variants. This sharing was possible because the experiments were evaluating features that were largely ignored by the placer, only being seen by the router. In the tests for this chapter we were unable to share the placements between different experiments, which meant that we needed more placement seeds to find good mappings. As a result, we used 12 placement seeds per application, although FIR and convolution were tested with 20 seeds because they showed a higher variability in quality of placement.

Previous experiments selected the best results from all trials by choosing the mapping with the lowest II, then energy, and finally area. These experiments also forced the tools

to find a mapping at a consistent II for each application, or to fail placement and routing. This was done to ensure that each application was tested at a similar level of performance, regardless of placement seed. By holding performance constant, the earlier experiments were able to report results using just the area-energy product. Unlike previous experiments, it was very difficult for the tests in this chapter to enforce that all placement seeds for an application map at the same II, and so it was necessary to use the area-delay-energy (ADE) product for these tests. The result with the lowest ADE was selected for each test.

8.4 Results

Specialization of the functional units involves three key principles: 1) stripping away expensive and underutilized resources, 2) avoiding overly specialized hardware, and 3) creating functional units with rich functionality when the extra logic is inexpensive (*i.e.* maximizing the utility of fixed resources). To apply these principles we started from a design that was homogenous, flexible, and where resources were abundant: each cluster had four universal functional units. This design was simple and provided a high-performance baseline fabric. The effects of each of these principles is explored in the following three sections as the architecture moves from a general to a specialized fabric design.

8.4.1 Paring down underutilized resources in the Functional Units

The first step of specializing the functional units is to pare down the most expensive and underutilized resource, which is the MADD device. Looking at Table 8.11, we can compare the size of each processing element when the functional unit is combined with its support logic. One big decrease in size is the S-ALU FU compared to the universal FU, which cuts the area in half by eliminating the MADD unit.

Figure 8.4 shows the area-delay-energy product, averaged across all benchmarks, for several architecture designs. As before, the total energy reported includes dynamic, static, configuration, and clock energy. This section focuses on the first four columns, which shows the trends as the MADD units are removed from individual functional units in architectures A, B, D, and G, from Table 8.12. Specializing the functional units by removing superfluous

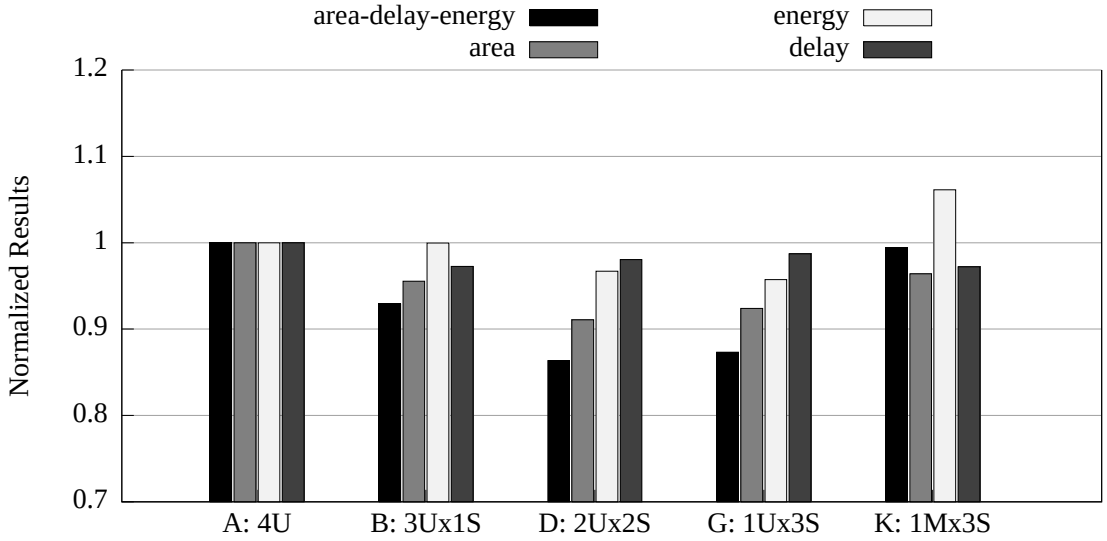


Figure 8.4: Average area-delay-energy product for architectures that are specialized with a mix of universal, S-ALU, and MADD FUs functional units.

MADD devices reduces the area-delay-energy product by as much as 0.86× the area-delay-energy of the baseline architecture. The best specialized design, D, has 2 universal FUs and 2 S-ALU FUs, versus a baseline with 4 universal FUs.

The trade-offs that occur when reducing the overall capabilities of a single cluster are highlighted in Figures 8.5 and 8.6. Figure 8.5 shows the per-cluster area and average static energy as the number of MADD units is changed. As expected the size and energy drop linearly for designs A, B, D, and G, as the clusters go from having 4 universal FUs to 1. The counterbalance of this trend is shown in Figure 8.6, which highlights the average number of clusters required for each architecture. We see that as the designs go from having 2 to 1 universal FU per cluster in architectures D and G, the number of clusters required goes up, which gives architecture D its performance advantage. This happens because, as we reduce the number of MADDs, the complex-dominant (45.0% multiplication) and select-dominant (21.2% multiplication) applications become resource-starved, and must use more than the minimum number of clusters to find enough multipliers.

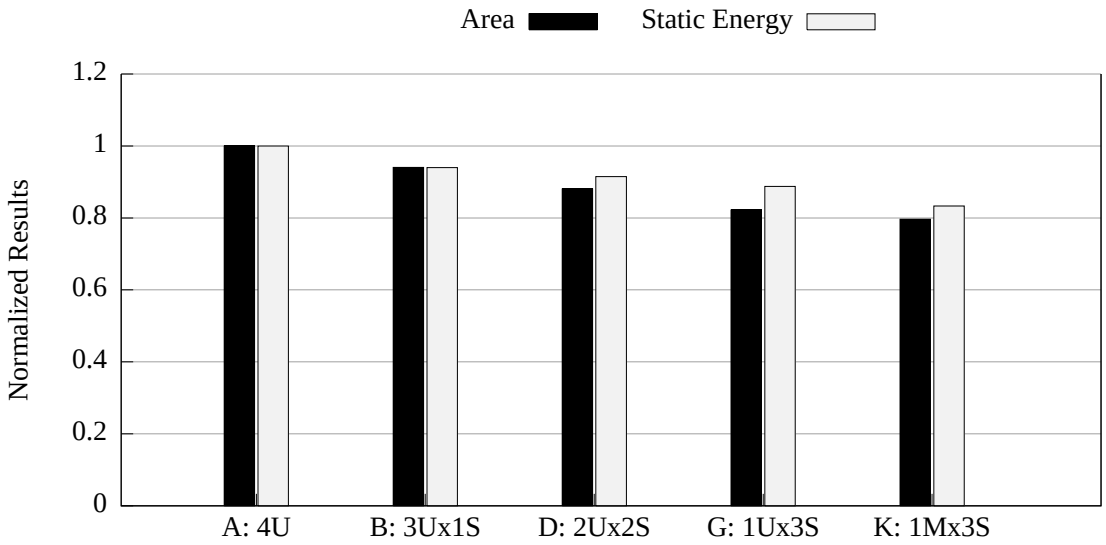


Figure 8.5: Average area and static energy for the clusters in architectures that are specialized with a mix of universal, S-ALU, and MADD FUs functional units. Static energy is dependent on the average application runtime.

Finally, the distribution of energy throughout the architectures is presented in Figure 8.7, where it is broken down into groups for the interconnect, cluster crossbar, and core; all of the results are normalized to the core energy for the A architecture. The core group is all of the functional units, register files, memories, and stream ports in a cluster. Between the core and crossbar energy we see that ∼73% of the energy is consumed within the clusters. Looking at the crossbar and interconnect we see that communication accounts for ∼47% of the energy. The distribution of energy is fairly flat for both the crossbar and interconnect across all designs, while the core energy is reduced as more MADD hardware is removed in architectures B, D, and G. As shown, the combination of the advantages in cluster size, average number of clusters required, and core energy consumed illustrate the superior performance of architecture D. D represents a "sweet spot", where we get steady gains from removing unneeded hardware, yet does not suffer the bloating on multiplier-heavy designs seen in architecture G. It turns out that this design performs better than others that strip away even more hardware, for reasons that are discussed in the subsequent sections.

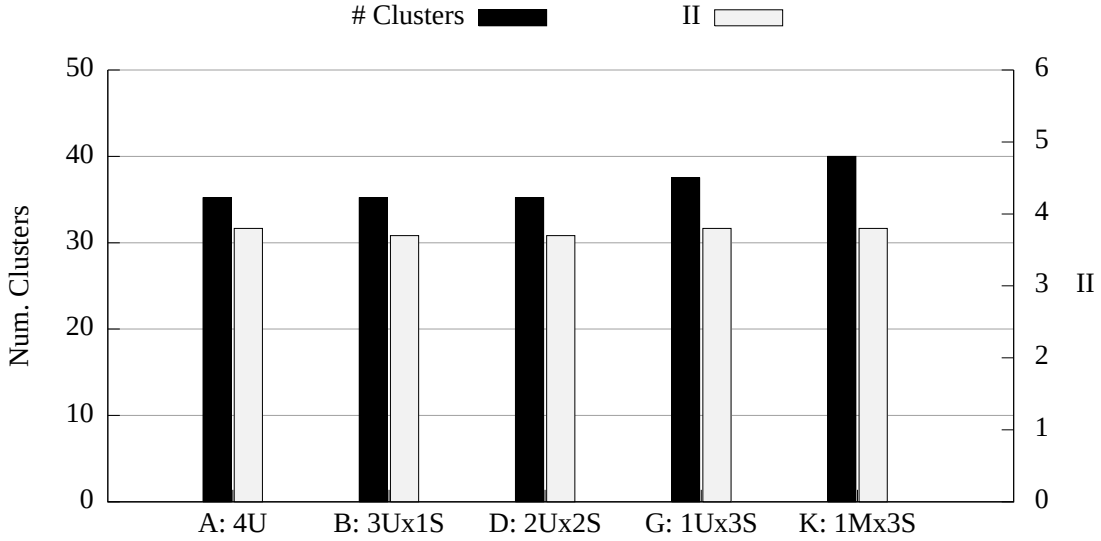


Figure 8.6: Average number of clusters required and application II for architectures that are specialized with a mix of universal, S-ALU, and MADD FUs functional units.

8.4.2 Avoiding excessive specialization

Given the benefits of paring down underutilized resources it may seem obvious to replace at least one universal functional unit with a dedicated MADD FU. Not only is the MADD functional unit smaller than the universal FU, it also requires one less crossbar port. Architecture K in Table 8.12 illustrates this design choice, and the results are shown in Figures 8.4, 8.5, 8.6, and 8.7 from the previous section. It turns out that architecture K is overly specialized with the MADD FU and performs worse, in terms of overall area-delay-energy product and total energy consumed, than all other specialized architectures, and only marginally better than the general design with four universal FUs as shown in Figure 8.4. Furthermore, not only does it require more clusters on average as shown in Figure 8.6, but it also consumes disproportionally more core energy (Figure 8.7).

The fact that architecture K consumes more core energy than can be attributed to the average increase in array size leads us to another observation about the advantages of co-locating resources to improve spatial locality; this observation can be made by considering

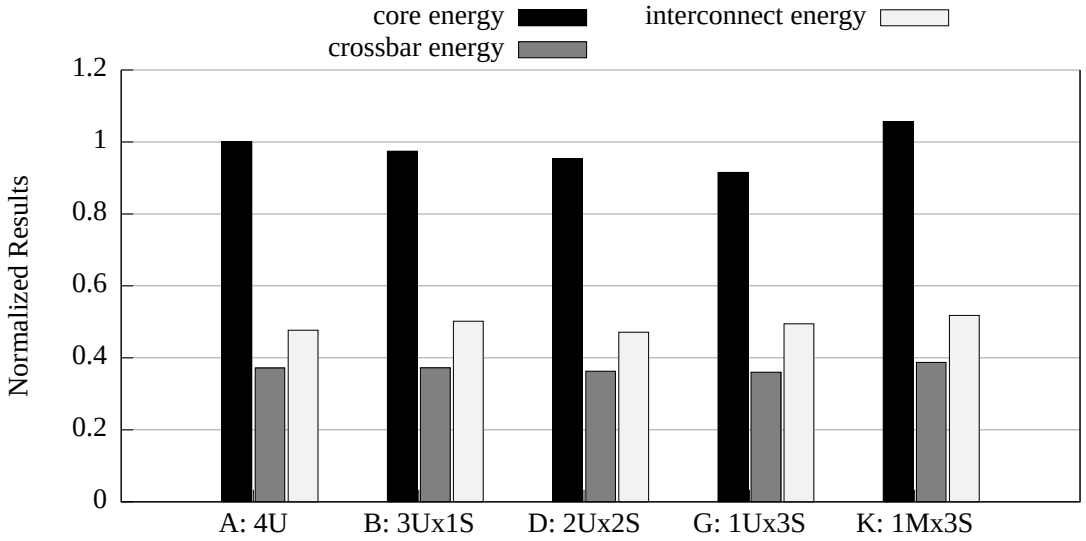


Figure 8.7: Average energy distribution for architectures that are specialized with a mix of universal, S-ALU, and MADD FUs functional units. Energy distribution is for the core components, crossbar logic and interconnect, and the global grid interconnect. All energy results are normalized to the core energy for the A architecture.

the per-cluster energy (Figure 8.5), and core energy (Figure 8.7), for experiments G and K. Architecture G has a greater per-cluster energy (by 7%) than K, since the universal ALU consumes more energy than a MADD. However, overall architecture G consumes 13% less core energy than architecture K, which is a greater margin that would be expected due to the fact that K required an average array that was 7% larger. We believe this is due to G being able to co-locate instructions on a single functional unit better than K, allowing sequences of operations to stay in the same processing element instead of having to traverse the cluster's crossbar (discussed in Section 7.12.1). Specifically, when using universal FUs, entire sequences of operations can execute on a single functional unit, taking advantage of the internal feedback paths in the functional units. This kind of execution pattern is unavailable to dedicated MADD units, especially for operation sequences that contain select operations; sequences that have select operations before or after an arithmetic operation

are fairly common due to predicated execution and loop entry and exit.

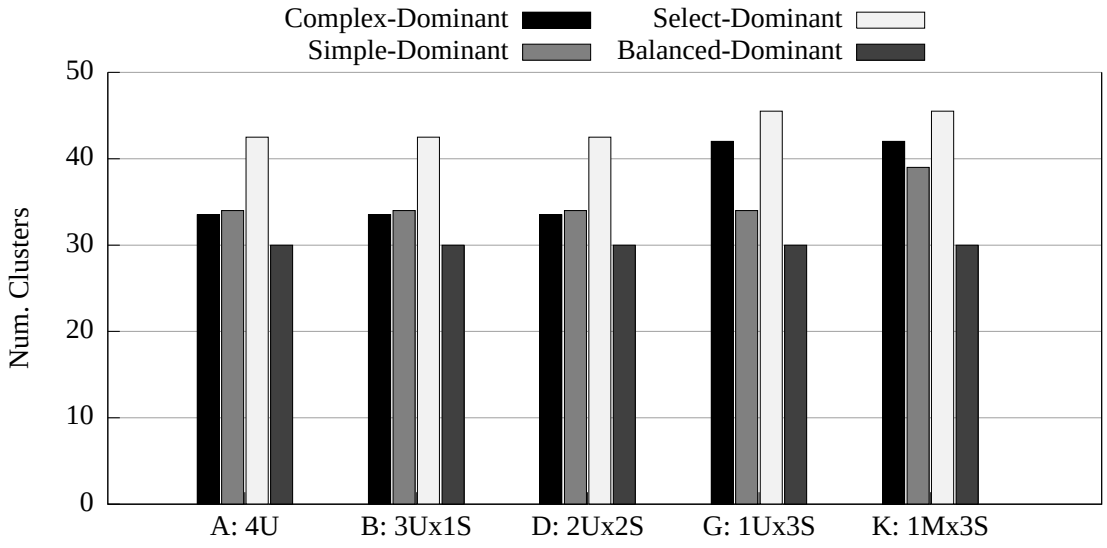


Figure 8.8: Average number of clusters required for each application category and for architectures that are specialized with a mix of universal, S-ALU, and MADD FUs functional units.

As alluded to earlier, overly specialized units cause problems with benchmarks that do not contain enough operations that are supported by that unit. In such a case, the number of useful functional units per cluster is effectively reduced by the number of overly specialized functional units, which in turn can increase the average array size required to execute applications. Figure 8.8 shows the number of clusters required for each application category. For complex- and select-dominant applications architecture K and G both require more clusters than architectures A, B, and D, which have two or more universal FUs. Furthermore, for simple-dominant applications architecture K requires more clusters than all of the other designs, which underscores the limitation of a dedicated MADD unit to adapt to applications that are not rich with complex arithmetic.

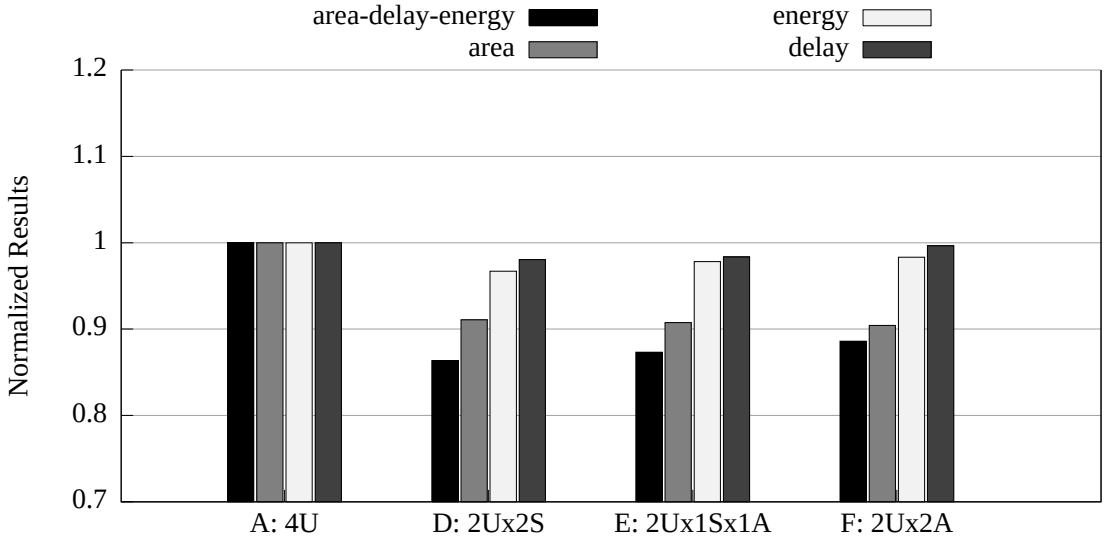


Figure 8.9: Average area-delay-energy product for architectures with 2 universal FUs and a mix of S-ALU and ALU functional units, with architecture A as reference.

8.4.3 Exploiting fixed resource costs

The third design principle we mentioned earlier is to make functional units as rich as possible when the additional functionality is cheap. In the previous section we considered replacing universal FUs with S-ALUs, which remove the multiplier but retain the ALU, select, and shift functionality in one unit. Since shift operations are relatively rare in our benchmarks, it might make sense to reduce some or all of the S-ALUs to pure ALUs. The costs of the S-ALU, ALU, and dedicated shifter are presented in Table 8.8, which show that the ALU and shifter hardware consume approximately the same amount of area and energy. However, when we look at Table 8.11, we see that the shifter is relatively cheap, and that an ALU is only 0.93× smaller than a S-ALU when the peripheral logic and crossbar connections are also factored in.

In Figure 8.9 we take the best answer from the previous section, the D architecture with 2 universal FUs, and vary the other 2 functional units between S-ALUs and ALUs. As can be seen, the differences between the architectures D, E, and F are not significant, though

as we convert from S-ALUs to ALUs we slightly decrease the area, and slightly increase the power and delay. Due to the small cost of including a shifter with each ALU, it makes sense to use the S-ALU to maximize the utility of both the peripheral logic and crossbar ports. The other advantage of building richer functional units is an increased opportunity for spatial locality when executing a sequence of operations that requires a diverse set of operators.

8.4.4 Further Analysis: some small details

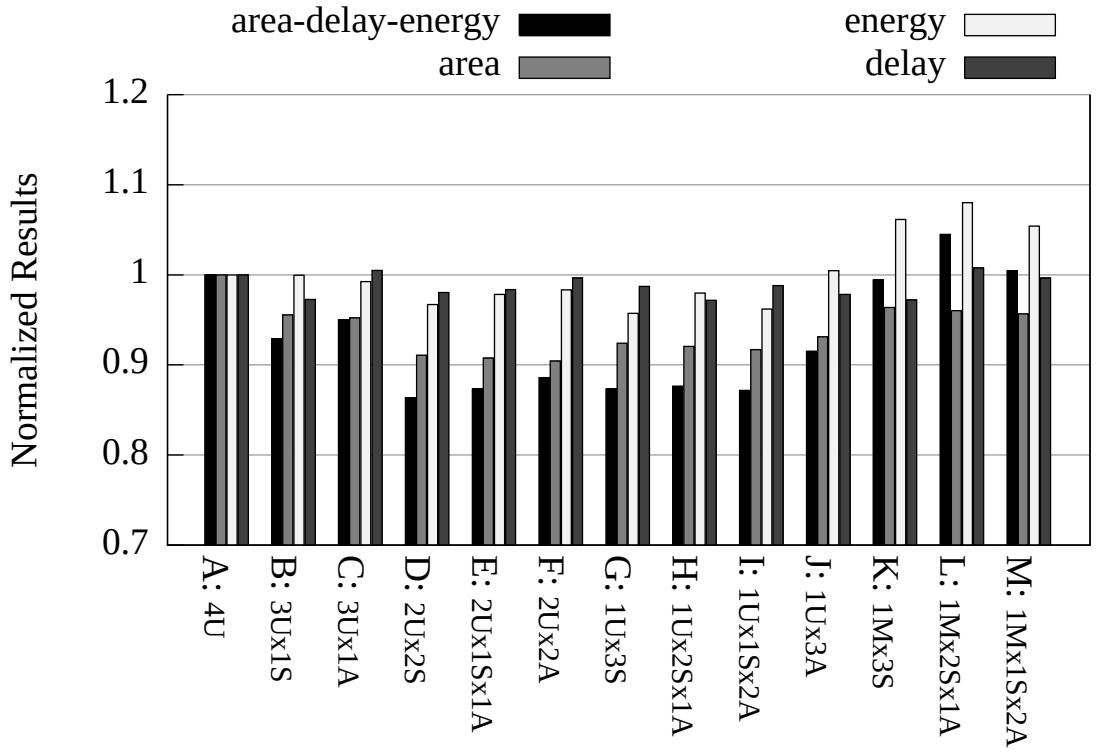


Figure 8.10: Average area-delay-energy product for architectures with specialized functional units.

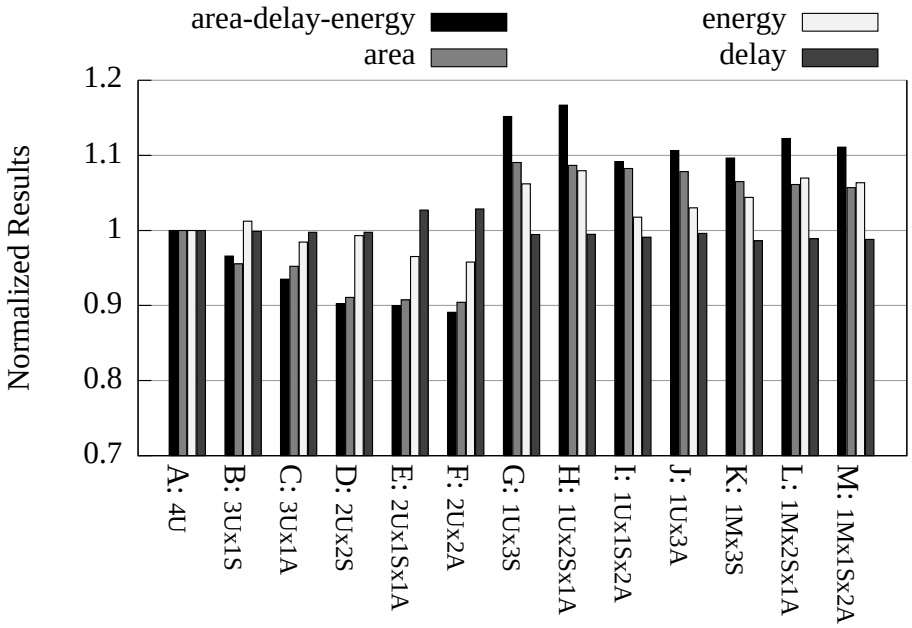
Figure 8.10 presents the area-delay-energy product for all of the tested architectures. Beyond the initial observations, there are a few interesting phenomena to explore. Design J has a worse area-delay-energy product than the rest of the designs with a single universal

FU. This likely is a result of oversubscribing the universal FU by co-locating all of the MADD and shift operations, which is another byproduct of the over specialization of the ALU functional units in architecture J.

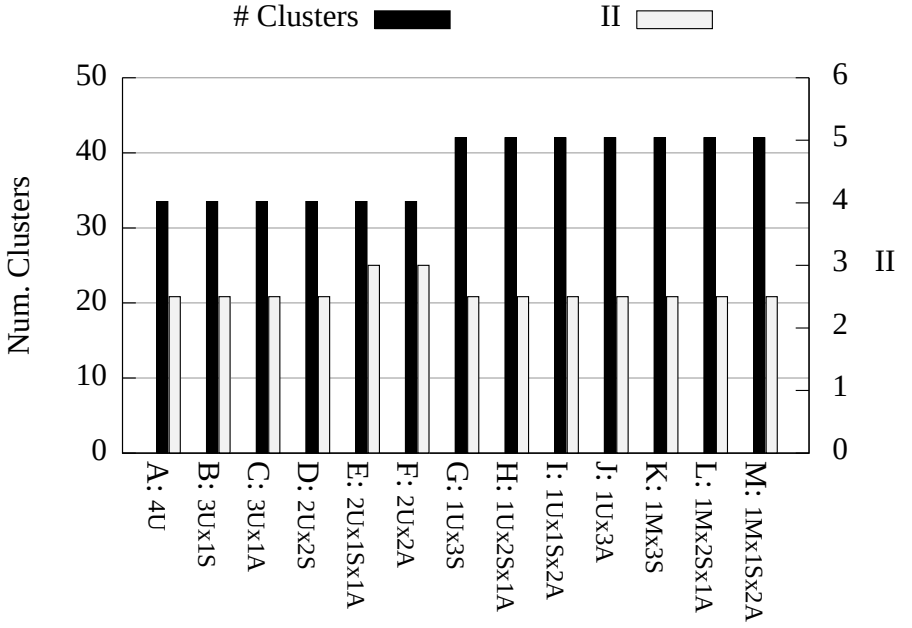
In Section 8.1 applications were grouped by the dominant type of operation that was executed. Area-energy-delay and number of clusters are shown for all architectures and each application group in sub-figures of Figures 8.11, 8.12, 8.13, and 8.14. Based on the design principles presented, the expected performance patterns are still visible when examining the performance of each application group on the different architectures. Furthermore, justification for the optimized design choices can be isolated by examining both the distribution of benchmarks within the benchmarks suite (Table 8.3), and the results for each application group. The prevalence of simple-dominant applications in the domain guides the optimized design to an architecture with no dedicated MADD FUs, and at least a single universal FU plus a mix of ALUs and S-ALU FUs. The select-dominant applications perform better on architectures with at least two universal FUs as shown in Figure 8.13, because they also perform a fairly large number of complex arithmetic and shift operations that take advantage of the extra universal FUs. Finally, Figure 8.14 shows that the diversity of the operations within the balanced-dominant class perform reasonably well on all designs B thru I and K thru M, but not on the overly specialized design of J which puts all operational diversity on a single universal FU.

8.5 Predicting the mix of Functional Units

Having explored the design space of functional units in the context of several application domains, it is valuable to be able to predict a good functional unit mix for different application domains without extensive testing. We build such a model by combining the three principles for specializing the functional units presented in Section 8.4, and other constraints presented in Section 8.2. Then we incorporated several Mosaic-specific considerations for use in this work. Aside from their applicability to the Mosaic CGRA, these guideline can be applied to other CGRAs and tile-based spatial accelerators (beyond CGRAs) that focus on maximizing their computation-to-communication ratio and leverage simple computational elements.

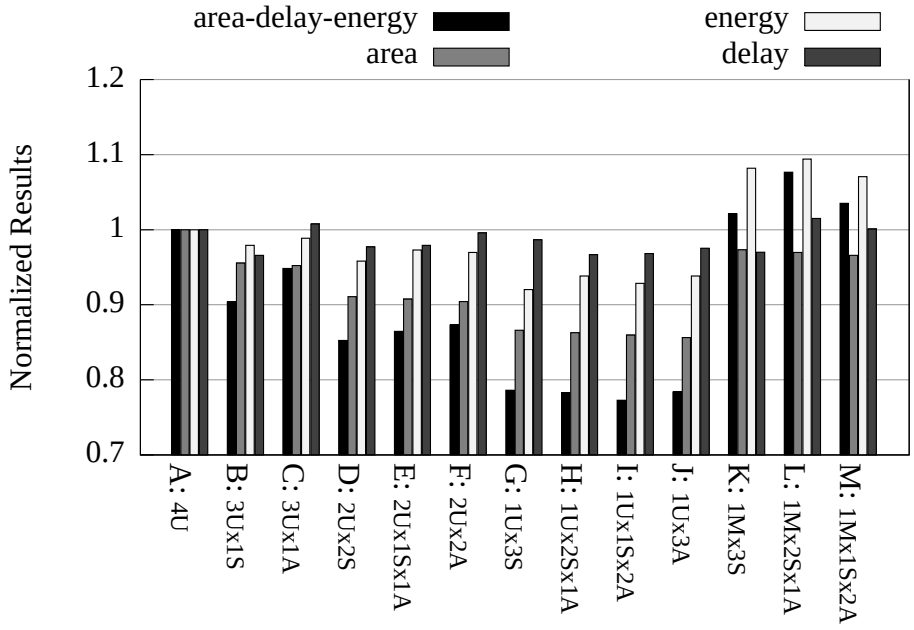


(a) Complex-Dominant: Area-delay-energy product and individual metrics.

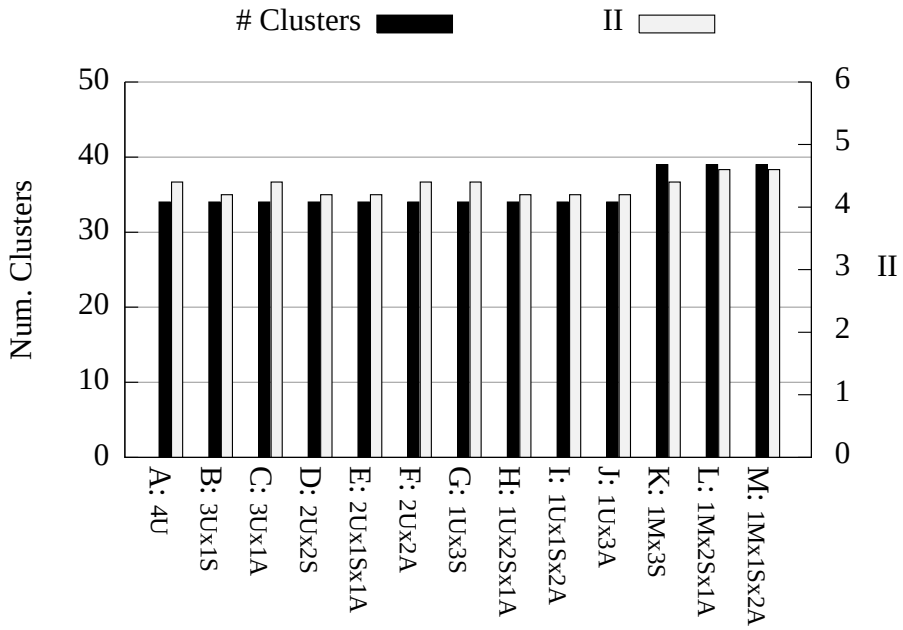


(b) Complex-Dominant: CGRA Size (# Clusters) and initiation interval (II)

Figure 8.11: Results for complex-dominant application group with specialized functional units.

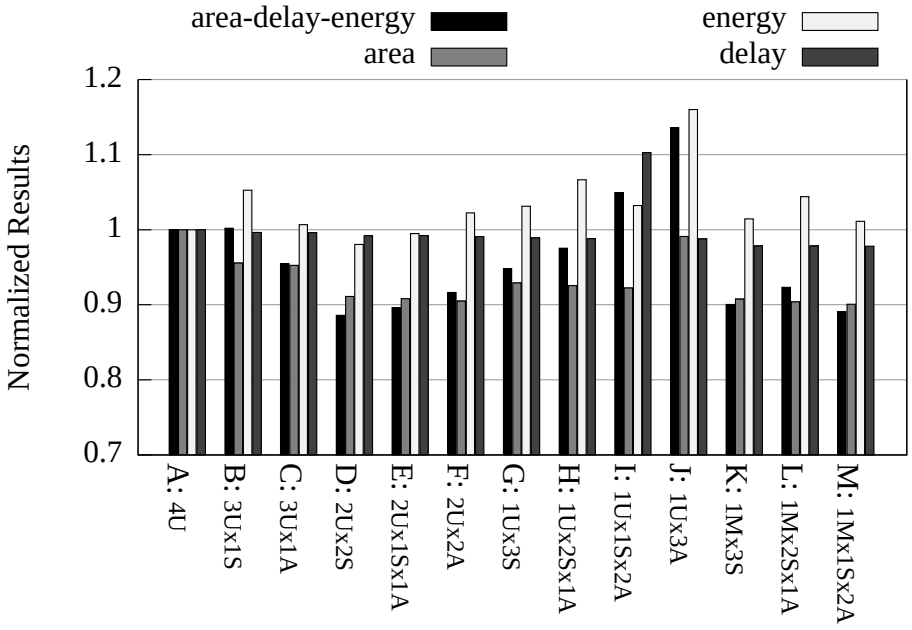


(a) Simple-Dominant: Area-delay-energy product and individual metrics.

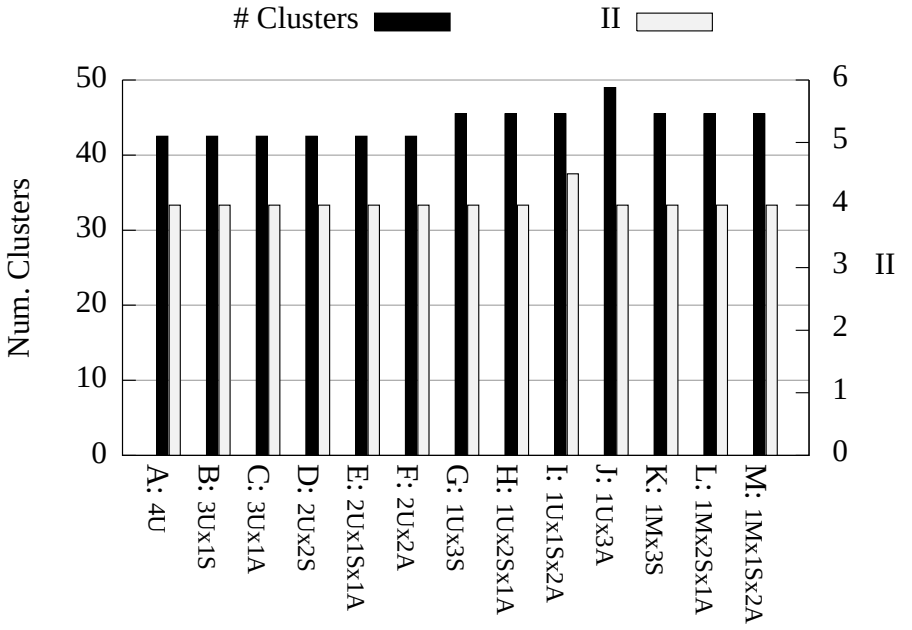


(b) Simple-Dominant: CGRA Size (# Clusters) and initiation interval (II)

Figure 8.12: Results for simple-dominant application group with specialized functional units.

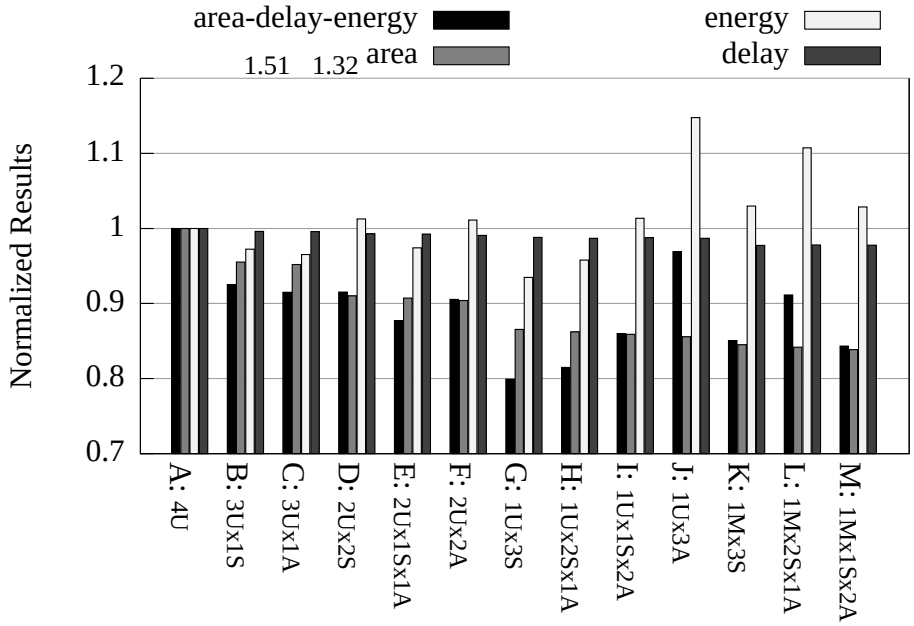


(a) Select-Dominant: Area-delay-energy product and individual metrics.

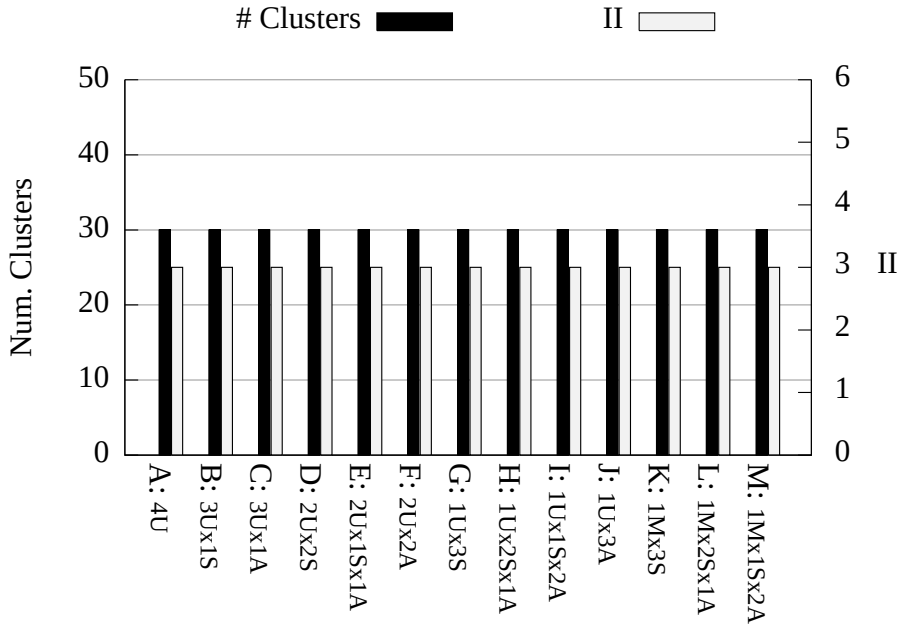


(b) Select-Dominant: CGRA Size (# Clusters) and initiation interval (II)

Figure 8.13: Results for select-dominant application group with specialized functional units.



(a) Balanced-Dominant: Area-delay-energy product and individual metrics.



(b) Balanced-Dominant: CGRA Size (# Clusters) and initiation interval (II)

Figure 8.14: Results for balanced-dominant application group with specialized functional units.

Using this model, we show that it is possible to directly predict a good resource mix for the Mosaic CGRA's repeated tile(s) based on the characteristics of the application domains or benchmark suite. Combining the previous principles and various contraints, we present the following guidelines for selecting a composition of functional units for a CGRA's repeated tile(s):

1. Every tile must support all operations.

2. Remove expensive and underutilized hardware: strip away MADD devices where possible.

3. Avoid over specialization:

   - No standalone MADD devices since they are bad for applications with few complex operations, and suffer from poor spatial locality due to a lack of co-location.
   - No standalone shifter devices since they do not support a significant percentage of executed operations.

4. Make FUs support rich functionality when it is inexpensive to do so: avoid standalone ALUs since a S-ALU is only marginally more expensive and can make better use of peripheral logic and crossbar resources.

5. Provide functional diversity within a tile to allow collocation and simultaneous execution of operations on expensive resources.

Looking at our benchmark suite and the best specialized design (D), let's see what resource mix we would have predicted given our guidelines. On average, across all benchmarks, complex arithmetic operations were 11% of the operation mix, but the peak demand was 47.1% of the operation mix. Naively this would argue that each cluster only requires one to two MADD devices per cluster. While two MADD devices per cluster is on the high side of the demand, our results have shown a definite advantage for modest over-provisioning of critical resources. The primary reason for this is that a small surplus allows critical

loops and collections of critical loops to co-schedule expensive operations, like the MADD, simultaneously in the same cluster. This added flexibility helps avoid potential scheduling bottlenecks. The capabilities of the functional unit can be further enriched by using universal FUs to provide the MADD devices. In addition to two universal FUs, the guidelines would suggest that the two remaining FUs should be S-ALUs since they are only marginally more expensive than ALUs and maximize the utility of their peripheral resources. This leads us to an architecture prediction that matches the best empirical results.

To test the utility of these guidelines, let us repeat this procedure for two of the application categories presented in Table 8.3 for the benchmark suite: simple-dominant and select-dominant. Starting with the simple-dominant, there is a much smaller percentage of complex operations, only 2.1% on average and a peak of 7.2%. The low frequencies predicts that there will be limited utility for two universal FUs, thus pushing for a single universal FU. One challenge is that the number of shift operations is significantly smaller (only 3.4%) than for the balanced-dominant and select-dominant categories. Therefore, it may be advantageous to remove the shifter from at least one of the S-ALUs, despite their small area and static energy overhead; moreover, if a shifter is occasionally used in an S-ALU it cannot be statically power-gated, and thus will consume unnecessary dynamic energy. However, the design guidelines suggest that architecture G (with one universal FU and three S-ALU FUs) would perform the best on simple-dominant applications. Looking at Figure 8.12(a) we see that architecture G reducing the area-delay-energy product to 0.79× the baseline, but that architecture I (with one universal FU, one S-ALU, and two ALUs) performed slightly better and reduced the area-delay-energy to 0.77× the baseline. Therefore the predicted architecture is only 2% worse than the best architecture.

Repeating the procedure for the select-dominant applications, we see that both complex and shift operations are a reasonable percentage of the resource mix. With more than a quarter of the operations being complex arithmetic, the architecture can take advantage of two universal FUs to provide ample opportunity for executing these operations. The number of shift operations is high enough that the advantages of having two S-ALUs is going to outweigh any marginal benefit from paring away one of the additional shifters. Therefore we would predict that architecture D with two universal FUs and two S-ALUs

would perform the best. Looking at Figure 8.13(a) show that architecture D performed the best and reduced the area-delay-energy product to 0.89× the baseline for the select-dominant category.

8.6 Caveats and Challenges

A challenge that we have observed with the current methodology is that the placement stage of SPR has a significant amount of variability in it, especially when an operation is supported by multiple devices. Experiments in Chapters 6 and 7 have focused primarily on resources that are ignored by the placer and so we have been able to reuse placements across a family of architectures, which has minimized placer noise in the experimental results. When specializing the functional units, placements cannot be reused because changing the types of functional units changes the legal placements of operations. For example, an architecture with 4 universal FUs per cluster can place 4 MADD operations in one cluster, while no other architecture tested can support this. Technically it is possible to take the placement of a specialized architecture and use it on a more general architecture. Such an effort will ensure that the more general architecture has a slightly worse ADE product because it is larger than the specialized architecture. This technique can be used to establish a reasonable expected area-delay-energy product on the more general architecture if the placer fails to independently find a good solution. However, when using a shared placement it is impossible to determine any advantages of a more general architecture over a specialized one. For example if we look at the ADE results for the complex-dominant benchmarks (Figure 8.11(a)), there are several architectures that have results that are worse than expected. These irregularities are due to placement noise in the FIR application.

Specifically, even though the FIR filter has very few shift operations, the area-delay-energy on architectures G and H are worse than on architecture I, and these architectures only differ in the number of shifters per cluster. So, aside from a small reduction in area, these architectures should all perform similarly, but SPR is unable to find a placement for G or H that is as good as one for I. We verified that these types of irregularities were placement noise by taking the best placement for a more restricted architecture and rerunning routing and simulation on the architecture with a questionable placement. The result was an area-

Table 8.13: Opportunities for operator fusion and profile of produce-consumer patterns.

(a) Profile of produce-consumer relationships.

Patterns	Percentage of Total Ops
Op-Op	11.1
Op-Mux	5.2
Mux-Op	7.1
Mux-Mux	9.0
Op with fanout	23.1
Mux with fanout	15.1
Op to iter. delays	8.1
Mux to iter. delays	8.9
Misc	12.4

(b) Frequency of word-wide back to back operations that could be fused into compound operations.

Operation Sequence	Num. Operations	Percentage of Total Ops
ALU-ALU	60	2.0
ALU-Shift	9	0.3
Shift-ALU	11	0.4
MADD-ALU	8	0.3
MADD-Shift	9	0.3
ALU-Mux	159	5.3
Shift-Mux	8	0.3
Mux-ALU	45	1.5
Mux-Shift	64	2.1
Mux-MADD	9	0.3
Mux-Mux	289	9.0

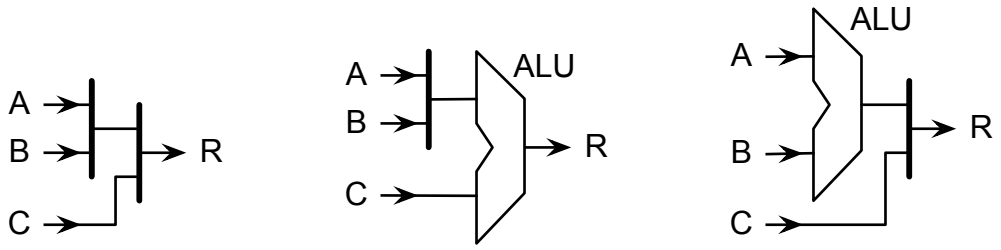
delay-energy product that was very similar to the area-delay-energy product of the more restricted architecture. This shows that it is a result of the stochastic part of the toolchain failing to find as good of a mapping on a more flexible architecture as on a more restricted architecture. Despite the variability found in the placement phase of SPR, we are still able to find strong trends and consistent results when averaging across the best results for all architectures.

8.7 Opportunities for chaining functional units

Throughout this chapter we have explored ways to improve the area-delay-energy product of an architecture by specializing its functional units. Exploring additional avenues of specialization requires us to look beyond the current set of primitive and compound functional units that have been presented. One technique for developing richer, more complex

functional units is to compose and cascade hardware operators to perform operations in sequence, similar to the MADD hardware. Fusing back-to-back multiply and add operations that had no intermediate fanout was a valuable optimization, but it was primarily driven by an inexpensive optimization available in hardware multipliers. A possible future optimization is to find other back-to-back operations that have no intermediate fanout and fuse them into a single operation. Doing this reduces both inter-functional unit communication and can potentially shorten an application's critical path; however, it is unclear if this optimization would be effective. Table 8.13(a) identifies opportunities for fusion by enumerating the various producer-consumer relationships found in the benchmark suite. The first four categories comprise 32.4% of the operations and have no fanout on the connecting signal, thus making them possible candidates for fusion. Some of the biggest categories are operations (and multiplexers) with some immediate fanout, followed by operations (and multiplexers) that feed iteration delays, neither of which are amenable to the simple fusion being considered here. Despite the apparent simplicity of fusing a pair of operators that have an iteration delay as an intermediary, iteration delays represent the delay from one application loop iteration to the next and thus turn into a variable amount of real delay based on the length of the modulo schedule. Therefore, determining if moving an iteration delay in the dataflow graph is profitable (*i.e.* does the benefit of a fused operator exceed the additional storage requirements), is beyond the current capabilities of the techmapper.

Of the 32.4% of operations that have no fanout, ones that are word-width and form a pair that can be fused, are broken down further by the type of operation pair in Table 8.13(b). This excludes those pairs that have single bit operations and fused operators such as MADDs. Examining the back-to-back pairs in Table 8.13(b) we see that outside of MUL-ADD fusion, there are few opportunities for combining other operations. The most common patterns are Multiplexer to Multiplexer, ALU to ALU, ALU to Multiplexer, and Multiplexer to ALU. Figure 8.15 shows three possible designs. The two costs for supporting compound operations are additional input ports from the crossbar with supporting peripheral storage, and augmenting the functional unit's hardware. With such a diverse set of fusion patterns and low frequencies of individual combinations, it is unlikely that supporting many of the compound operations across all functional units or clusters would be profitable. Using a



(a) Fused Multiplexer-Multiplxer FU

(b) Fused Multiplexer-ALU FU

(c) Fused ALU-Multiplexer FU

Figure 8.15: Three block diagrams of possible fused devices that compute a pair of select operations, or both an ALU and select operation in the same cycle.

homogenous cluster design would have lots of underutilized hardware, and a heterogenous approach would have such a diverse set of functional units in the computing fabric that it would be difficult to balance resources. While the mux-to-mux is the most frequent pattern, this highly specialized functional unit is unlikely to prove beneficial for many of the same reasons that the dedicated MADD FU failed to outperform other specialized architectures as noted in Section 8.4.2. Therefore, it would probably be better to augment the existing ALU to provide fused mux-to-mux operations. The majority of the remaining identified opportunities would be covered by fusion between multiplexer and ALU hardware. However, these opportunities are split between fused select-ALU and ALU-select operations.

Supporting a fused select-ALU operation with the current ALU hardware is fairly straightforward, requiring only a bit of logic and one additional input to the existing ALU functional unit, as shown in Figure 8.15(b). In this design, the A and B inputs would be operands to the select stage, and the result is the first operand to the ALU stage. A third input, C, would provide the second operand to the ALU stage. The cost of this augmentation to the ALU functional unit is some peripheral logic and the crossbar read mux. To form as many of the fused select-ALU operation patterns as possible, the techmapper is used to commute the ALU operators inputs, as described in Section 4.2. Supporting a fused ALU-select operation as illustrated in Figure 8.15(c), is similar, except for the inter-

nal arrangement of hardware and ports. The exploration of the right balance of augmented ALUs is left as future work. A brief analysis suggests that some compiler transformations and/or techmapper optimizations may be able to consolidate some of those ALU-select and select-ALU combinations into a common pattern, thus improving the chances for a single fused select+ALU device to succeed.

8.8 Conclusions

Specializing the functional units within a CGRAs tile can improve the architecture's area-delay-energy product by 0.86× just by paring down infrequently used hardware devices. Specifically, multiply and multiply-add operations are expensive and relatively infrequent within the benchmark suite, thus requiring only one to two out of the four functional units to support them. While shift operations are also infrequent, they are relatively inexpensive to implement. More importantly, they do not require additional ports from the crossbar beyond what is required for an ALU. Therefore, they can be added for minimal cost and improve the opportunities for spatial locality; increased spatial locality reduces crossbar activity and increases the utility of each input port to the functional unit and the peripheral logic that supports the functional unit. Furthermore, maintaining a couple universal functional units within each cluster, rather than a set of FU types without functional overlap, provides better performance for applications within the domains that do not require specialized hardware.

Chapter 9

ENERGY-OPTIMIZED MOSAIC CGRA ARCHITECTURE

This dissertation examined the subspace of CGRA architectures defined in Chapter 3 and explored several architectural optimizations for improving those architecture's energy efficiency without sacrificing performance. The optimizations that were covered in this dissertation explored improvements in communication, storage, computation, and composition (topology). This chapter presents an optimized Mosaic CGRA architecture that incorporates the best set of architectural features, as well as a comparison with a baseline architecture to show the aggregate improvement in area-delay-energy product. In addition, this chapter closes with some broader conclusions and observations about the future of energy efficiency in spatial accelerators, as well as possible avenues for future research.

9.1 Baseline Mosaic CGRA

The Mosaic CGRA architectures are a class of statically-scheduled coarse-grained reconfigurable arrays. They are designed to exploit loop-level parallelism in an application's computationally-intensive inner loops (*i.e.* kernels) in an energy-efficient manner. The architectures are dynamically reconfigured, so that they time-multiplex their functional units and interconnect on a cycle-by-cycle basis. Like many CGRAs, the Mosaic architecture fits in the design space between FPGAs and VLIW processors, with similarities to a word-wide FPGA and a 2-D VLIW.

Chapter 3 defined the Mosaic CGRA architecture, and in broad strokes identified the class of CGRA architectures that it represented. From this initial definition, each set of experiments has started from a baseline architecture that is used as a point of comparison for subsequent optimizations. We provide the following baseline Mosaic CGRA architecture as a point of comparison for the final evaluation of the optimizations that were presented in this dissertation. As before, this architecture is designed to be representative of existing CGRA

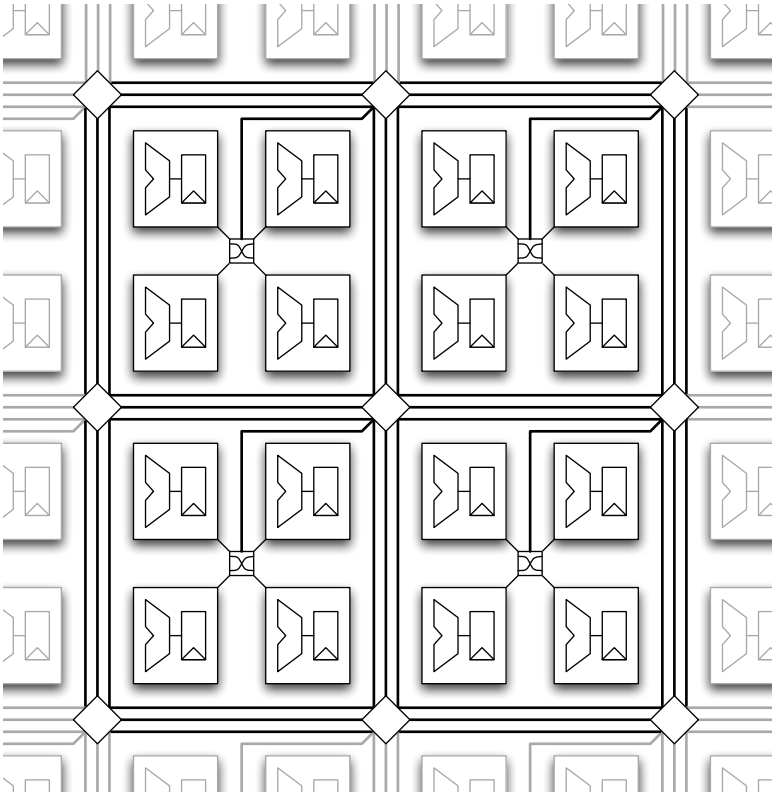
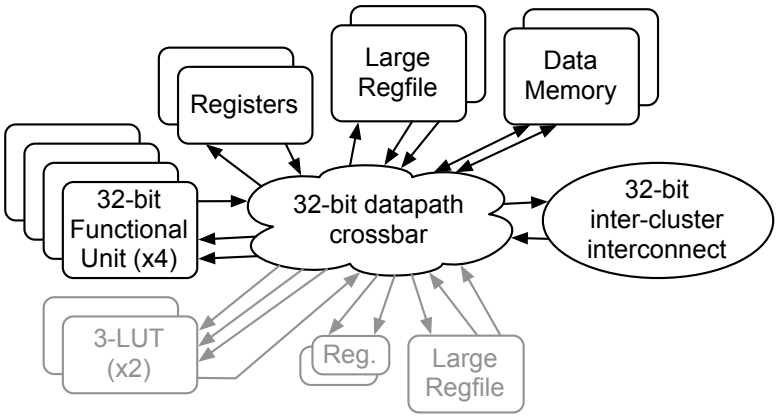
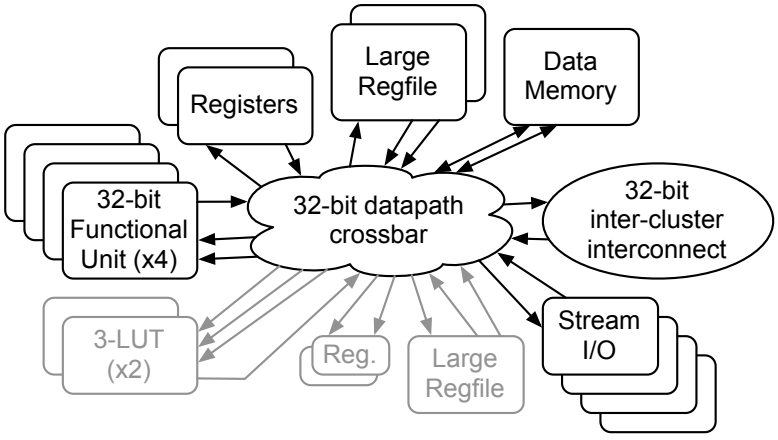


Figure 9.1: CGRA Block Diagram - Clusters of 4 PEs connected via a grid of switchboxes.

research architectures and incorporates state-of-the-art design practices. The Mosaic CGRA architecture presented in Chapter 3 is a cluster-based architecture that is arranged as a 2-D grid. We use a design that isolates the stream I/O to the periphery of the CGRA's compute fabric, and provides a higher density of data memories in the fabric's core. Therefore, there are two types of clusters that form the computing fabric, core tiles and edge tiles, which are shown in Figure 9.2. All of the clusters have four 32-bit processing elements. The processing element is shown in Figure 9.3, which provides 1 stage of retiming on each input. Figure 9.3(b) shows the design of the universal functional unit from Chapter 8, which is a compound functional unit that contains the ALU, shifter, and fused Multiply-ADD devices. The 32-bit datapath in each cluster also has two large traditional register files for long-term storage, two distributed registers, one or two data memories, and connections to the grid

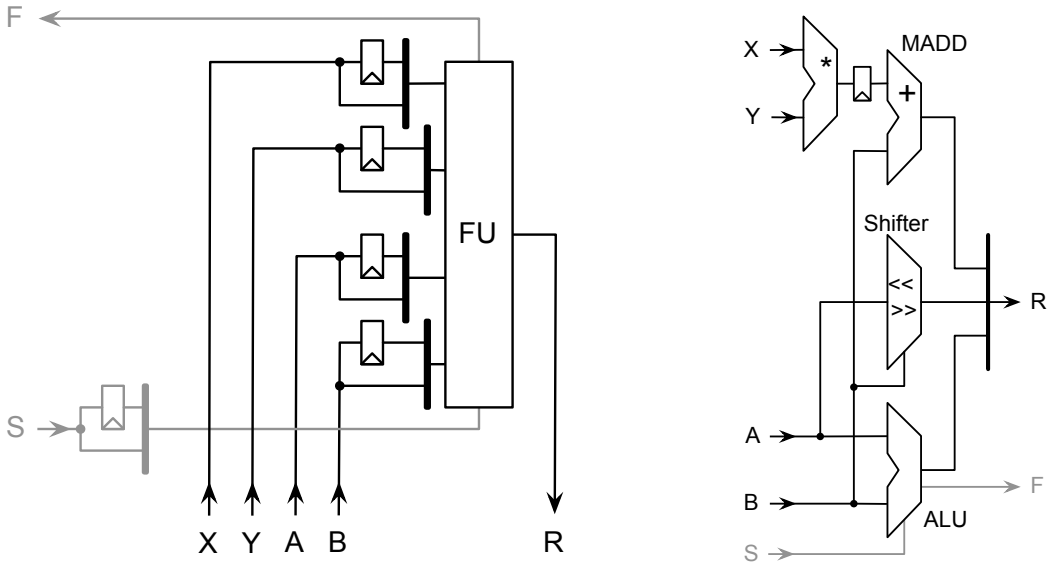


(a) Baseline Cluster Core Tile



(b) Baseline Cluster Edge Tile

Figure 9.2: Baseline clusters for core and edge tiles. Note that core tiles have an additional data memory, while edge tiles have stream I/O ports. Word-wide components are shown in black, and 1-bit components in grey.



(a) Baseline PE w/Universal Functional Unit

(b) Universal Functional Unit

Figure 9.3: Block diagram of processing element with a universal functional unit and 1 stage of registers for input retiming, and the universal functional unit internals. Note that word-wide components are shown in black, and 1-bit components in grey.

interconnect.

Additionally, there are several control resources in each cluster that share the word-wide interconnect within the cluster and in the grid between clusters. There are two 1-bit processing elements that contain 3-LUTs (shown in Figure 9.4), and a large traditional register file for single-bit long-term storage. The clusters in the center of the CGRA are the core tiles (Figure 9.2(a)), which have an additional data memory, while the edge tiles (Figure 9.2(b)) are on the peripheral of the compute fabric and have the stream I/O ports. Note that the edge tiles also contain a special control resource that is used to terminate the execution of the kernel, and one register per tile that transfers its value back to the host processor when the kernel exits.

A breakdown of the contents of the baseline core and edge cluster tiles follows:

- 4× 32-bit processing elements with universal functional units and input retiming reg-

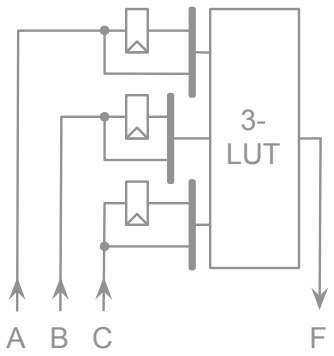


Figure 9.4: Block diagram of single-bit processing element with a 3-input lookup table (3-LUT) functional unit and 1 stage of registers for input retiming.

isters (Figure 9.3)

- 2× 32-bit distributed registers
- 2× 32-bit 16-entry traditional register files
- 1024K-word 32-bit data memories (**2 per core tile and 1 per edge tile**)
- 4× stream I/O ports (**edge tiles only**)
- 1× 32-bit live-out register that transfers its value to host processor
- 2× 1-bit processing element with 3-LUT and input retiming registers
- 2× 1-bit distributed registers
- 1× 1-bit 16-entry traditional register files
- `kernel_done` control resource

9.2 Optimizations

Throughout this dissertation we have explored several optimizations related to communication, storage, and specialization of the functional units. The following sections summarize the optimizations that are incorporated into the final optimized Mosaic architecture.

9.2.1 Communication

Chapter 6 explored the advantages of statically scheduled word-wide channels over configured channels in a grid interconnect. It showed that the flexibility of a statically scheduled interconnect far outweighed the additional configuration hardware required to time-multiplex the multiplexers in each channel. Its advantages in area and energy were primarily derived from a reduction in channel width. Furthermore, the dynamic, switching energy of the configured interconnect exceeded that of the scheduled interconnect, due to an increase in switched capacitance. As described in Section 6.6.2, this highlights a fundamental trade-off for configured channels, either they are not shared and the architecture requires a lot more of them, or they are shared but each signals ends up switching more capacitance. Specifically, the additional capacitance is a product of SPR's algorithm that shares configured channels with multiple signals over time, which lead to configurations with larger signaling trees that were active for all cycles of execution. In particular, each individual signal that shared part of its interconnect with another signal ended up switching more capacitance than the corresponding signal in a statically scheduled version of the interconnect. Forcing SPR to disallow values from sharing configured channels would lower the total switched capacitance of the interconnect (because of smaller routing trees), but the requisite increase in channel width would far outweighs any advantages.

Another optimization that was explored was the advantage of providing a separate single-bit interconnect for control signals. This optimization provided a modest reduction in datapath channel width, but also split the intra-cluster crossbar into two disjoint crossbars. Splitting the crossbar dramatically reduced the overall area and energy consumed by intra-cluster communication, since reducing the number of inputs and outputs to each individual crossbar led to a non-linear reduction in area (due to the quadratic relationship between crossbar I/Os and area). Therefore, we found that the best interconnect design for the Mosaic architects used a statically-scheduled word-wide interconnect and a separate static (*i.e.* configured) single-bit interconnect for control signals.

9.2.2 Storage

Chapter 7 explored several design options for managing the range of values that are produced and consumed during the execution of an application. The challenge for distributing storage structures throughout the CGRA is that most of the values are short-lived, existing for only a few clock cycles between production and consumption, and yet a non-trivial number of values are long-lived with a lifetime of many tens of clock cycles. We showed that customizing storage structures for short-, medium-, and long-term storage provided a significant reduction in the architecture's area-delay-energy product.

The distribution of registers and register structures is a key part to improving energy efficiency. For long-term storage we found that a single rotating register file that was shared within each cluster was valuable. Despite the fact that long-lived values only comprise about 10% of the total values that are produced and consumed, providing dedicated long-term storage in each cluster significantly reduced the number of times each value had to traverse the cluster's crossbar, which consumed a lot of energy. To handle values of an intermediate lifetime, and those that were produced and consumed by the same functional unit, we added a local rotating register file that was private to each functional unit that included a feedback path. Finally, retiming registers on the inputs to functional units, data memories, and stream I/O, provided the short-term storage, holding values for only one or a few cycles. Note that the feedback path internal to the functional units also had a fast path that only had a single retiming register on it, instead of going through the local register file.

For medium-term and long-term storage, we used an 8- and 16-entry rotating register file, respectively, because they were able to store a value with minimal recurring energy cost and hold it for enough cycles to amortize the cost of initially writing the value into the structure. Furthermore, we show that rotating register files provide a significant advantage for holding long-lived values versus traditional register files in a time-multplexed (modulo-scheduled) architecture. Short-term storage is primarily provided by retiming registers (with clock enables) on the input paths to the functional units and other devices, and the feedback paths attached to the functional units. The use of clock enables allow these input retiming registers to hold a value for multiple cycles, with no recurring energy, when they are lightly

utilized.

9.3 Functional Units

Chapter 8 explored designs for specializing functional units within the cluster. It shows that mixing universal functional units with specialized functional units, which only performed a subset of all operations, is desirable since some of the less frequent operations are also fairly large. In particular, we found that a design with two universal and two S-ALU functional units provided the best area-delay-energy product. The motivation for this resource mix is to provide the greatest flexibility and maximize the use of the processing element's input ports from the cluster crossbar. The universal functional unit combines a fused Multiply-ADD (MADD) unit with a shifter and an ALU, giving it the ability to execute all operations. Essentially, this combination extends the expensive MADD unit, which already has three input ports, with one more input port and a small amount of additional logic. This design decision guarantees that the universal FU can make good use of its input ports for all types of operations and in all applications. Similarly, the S-ALU FU combines the shifter and ALU, excluding the MADD hardware. This combination groups the two devices with only two input ports, again maximizing the utility of each input port. The inclusion of a universal FU also improved the opportunities for spatial locality across diverse sequences of instructions.

9.4 Impact of each optimization

Looking over the optimizations that have been evaluated in this dissertation, it is interesting to compare the impact of each optimization. Table 9.1 highlights all of the explored optimizations and their relative improvement versus their respective baseline or previously (incrementally) optimized architectures. The optimizations are listed in the order of the significance of their impact, on both the architecture's design quality and area-delay-energy product. The first two optimizations listed are the fully scheduled interconnect and the auxiliary single-bit interconnect. These optimizations offer the most significant improvements in energy efficiency due to their ability to reduce interconnect channel width and the size of each cluster's crossbar. The next most important optimization is the use of the

rotating register file for long-term storage, because of its superior ability to hold long lived values. Finally, the specialization of the functional units is effective at reducing cluster area by removing underutilized resources. The remaining optimizations, while valuable, provide smaller improvements in the overall energy efficiency.

9.5 Optimized Mosaic CGRA Architecture

Combining the results from each exploration, we present the optimized Mosaic CGRA architecture. Similar to the baseline architecture, it is a cluster-based architecture that is arranged as a 2-D grid. As noted in the optimizations, a key difference is that cluster and interconnect resources are split into a 32-bit datapath and a 1-bit control-path, as shown in Figure 9.5. Within the core and edge tiles, the processing elements have been optimized and replaced with more sophisticated designs that are shown in Figures 9.6, 9.7, and 9.8. Furthermore, the traditional register file in the large, long-term storage is replaced by a rotating register files, and the number of long-term storage structures is reduced from two to one.

A breakdown of the contents of the optimized core and edge cluster tiles follows:

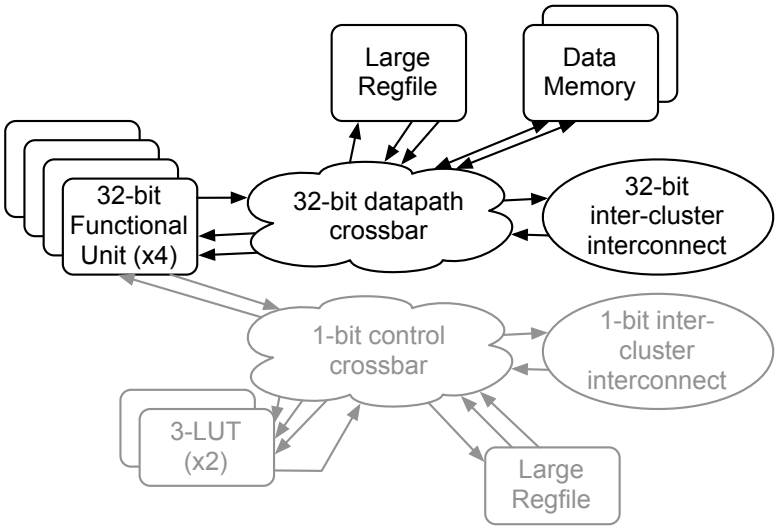
- 2× 32-bit processing elements with universal functional units and peripheral logic (Figure 9.6)
- 2× 32-bit processing elements with S-ALU functional units and peripheral logic (Figure 9.7)
- 1× 32-bit 16-entry rotating register files
- 1024K-word 32-bit data memories (**2 per core tile and 1 per edge tile**)
- 4× stream I/O ports (**edge tiles only**)
- 1× 32-bit live-out register that transfers its value to host processor
- 2× 1-bit processing element with 3-LUT and peripheral logic
- 1× 1-bit 16-entry rotating register files
- `kernel_done` control resource

It is important to note that the use of a 4-ported local, rotating register file in the

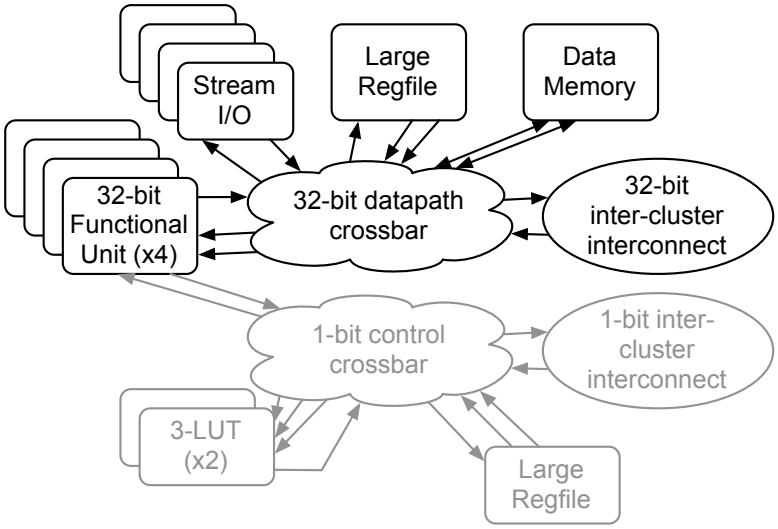
Optimization	Area-Delay-Energy	Area	Delay	Energy
Fully scheduled interconnect[1]	0.29×	0.42×	0.90×	0.75×
Auxiliary single-bit interconnect[1]	0.93×	0.99×	0.99×	0.95×
—with split word-wide and single-bit crossbar[2]	N/R	0.36×[3]	N/R	0.99×[4]
Rotating register file for long-term storage	0.82×	1.00×	1.01×	0.81×
Specializing Functional Units	0.86×	0.91×	0.98×	0.96×
Registering outputs of functional units[5]	0.93×	1.02×	1.04×	0.88×
Intra-PE feedback paths[6]	0.97×	1.00×	0.99×	0.98×
Private, local rotating register file	0.92×	0.98×	1.01×	0.93×
Dynamically-enabled input retiming registers	0.95×	0.99×	1.00×	0.96×

Table 9.1: Relative improvements of each optimization versus either a baseline architecture or previously optimized architectures, as described in Chapters 6, 7, 8. Optimizations are listed in order of overall impact on both the architecture's design quality and the area-delay-energy product. N/R indicates that the metric was not reported.

1. The impact of a scheduled interconnect and complementary single-bit interconnect was measured in reference to the interconnect's area, energy, and delay.
2. Effectiveness of splitting the crossbar was evaluated by examining the area and static energy plus configuration energy consumed by the crossbar resources. Therefore the overall impact on the architectures area-delay-energy product, area-energy product, and delay was not reported for this isolated test.
3. Area for direct comparison of a single word-wide crossbar versus split word-wide and single-bit crossbars only considered crossbar's area.
4. Energy for direct comparison of a single word-wide crossbar versus split word-wide and single-bit crossbars only considered the crossbar's static energy and configuration energy.
5. To help isolate the impact of adding registers on the outputs of the functional units, the critical path of the architecture was not shortened for this test as noted in Chapter 7.
6. The internal feedback paths within a processing element provide a small improvement in isolation, but are key to the success of registering the outputs of the functional units.

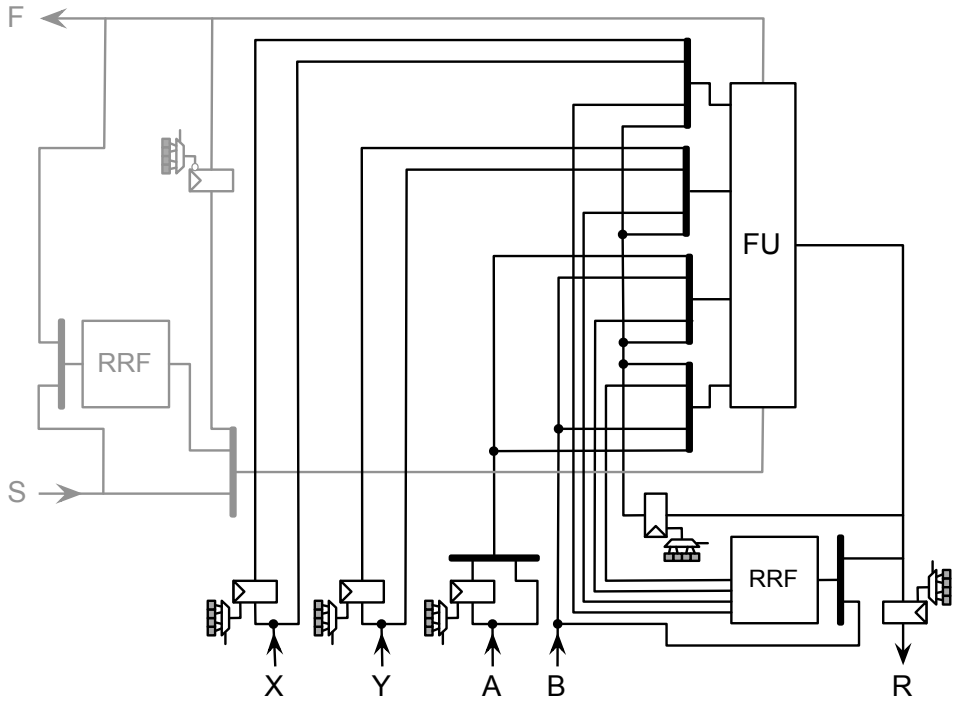


(a) Optimized Cluster Core Tile

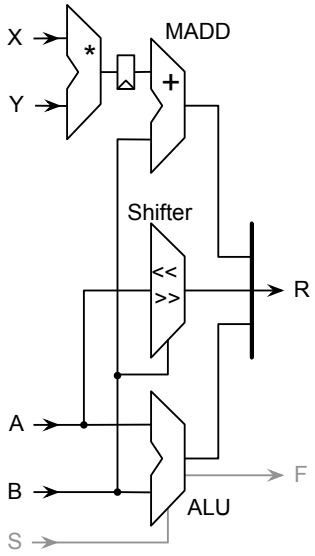


(b) Optimized Cluster Edge Tile

Figure 9.5: Optimized clusters for core and edge tiles. Word-wide components are shown in black, and 1-bit components in grey.

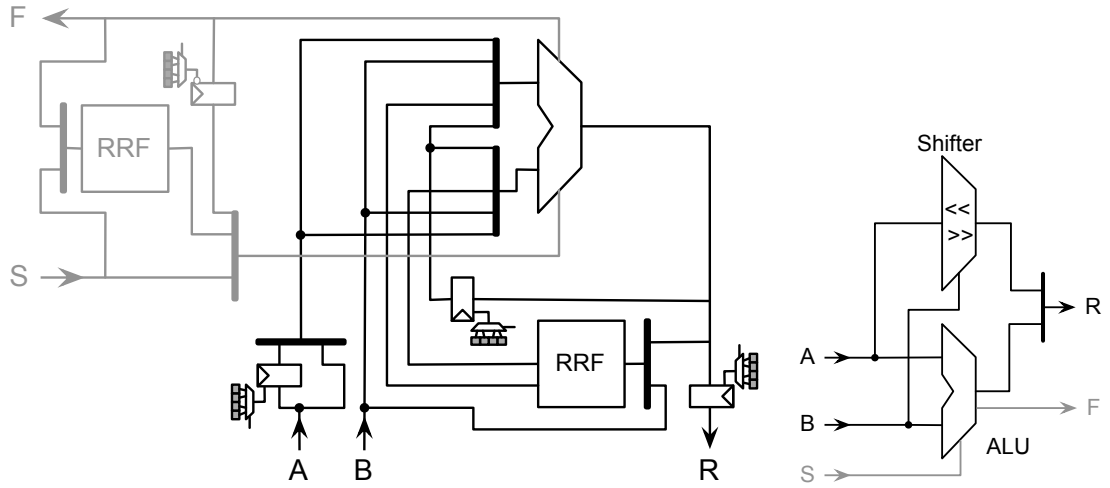


(a) Optimized PE w/Universal Functional Unit



(b) Universal FU

Figure 9.6: Block diagram of optimized PE with a Universal FU, used in final Mosaic CGRA.



(a) Optimized PE w/S-ALU Functional Unit (b) S-ALU (Shifter-ALU) FU

Figure 9.7: Block diagram of optimized PEs with a S-ALU FU, used in final Mosaic CGRA.

optimized universal FU processing element (Figure 9.6(a)), is a concession to SPR's method of placing constant values. Currently, SPR assigns a location and phase in the schedule for each constant, just like operations on functional units. Given that the universal FU has four inputs, it is possible for SPR to assign four constant values to the local register file that have to be read out simultaneously. In a structure without sufficient read bandwidth, such a placement would be unroutable. While this scenario is unlikely, we provisioned (and pay for) one port in the local register file per input port on the functional unit. This design concession was done to avoid the possibility of SPR creating a placement that was unroutable, based on the placement of local constants. In the future we expect that SPR can be modified to minimize the chance of co-locating more than a maximum number of constants on a given storage structure. This would allow us to redesign the universal FU processing element to use only a 2-ported rotating register file (Figure 9.9).

9.5.1 Evaluation of Optimized Mosaic Architecture

To evaluate the net gain from all of the optimization explored in this dissertation we mapped the benchmark suite to both the baseline and optimized architectures. We used the same set

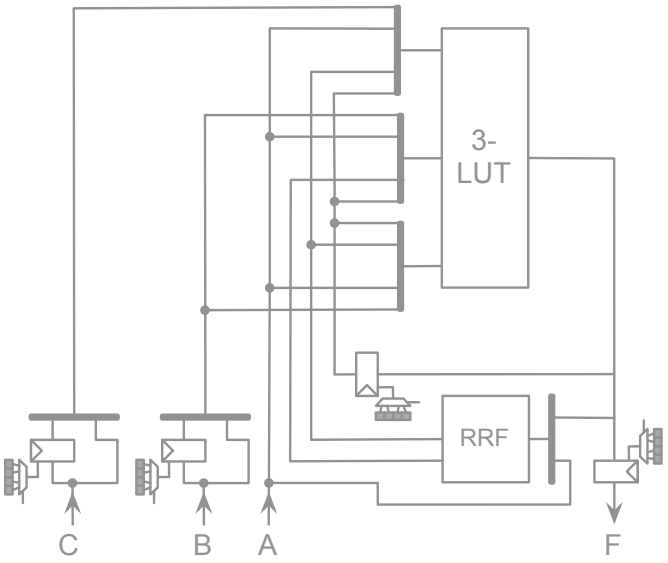


Figure 9.8: Block diagram of optimized single-bit processing element with a 3-LUT, used in final Mosaic CGRA.

of benchmarks, tuning knob settings, versions of the toolchain, and techmapper optimizations as Chapter 8. The tests were conducted using the same experimental methodology as Chapter 8, except that each benchmark was mapped to both architectures using eight random placement seeds, and the minimal required channel width for the grid interconnect was found using the binary search technique from Chapter 6.

Figure 9.10 shows the area-delay-energy product, as well as other metrics, for both the baseline and optimized architectures. Overall, for the optimized architecture we see a combined reduction in area-delay-energy product to 0.10× the baseline Mosaic architecture. As noted in Chapter 6, a large percentage of the reduction in area results from the smaller channel width and partitioned cluster crossbar. In Figure 9.11 we see that the optimized architecture requires 0.37× the number of word-wide channels as the baseline architecture.

A coarse distribution of area between the clusters and global interconnect is shown in Figure 9.12, for both the baseline and optimized architectures. Interestingly, the distribution of cluster versus interconnect area for both baseline and optimized architectures is approximately 88% cluster resources and 12% global interconnect. Despite the relatively

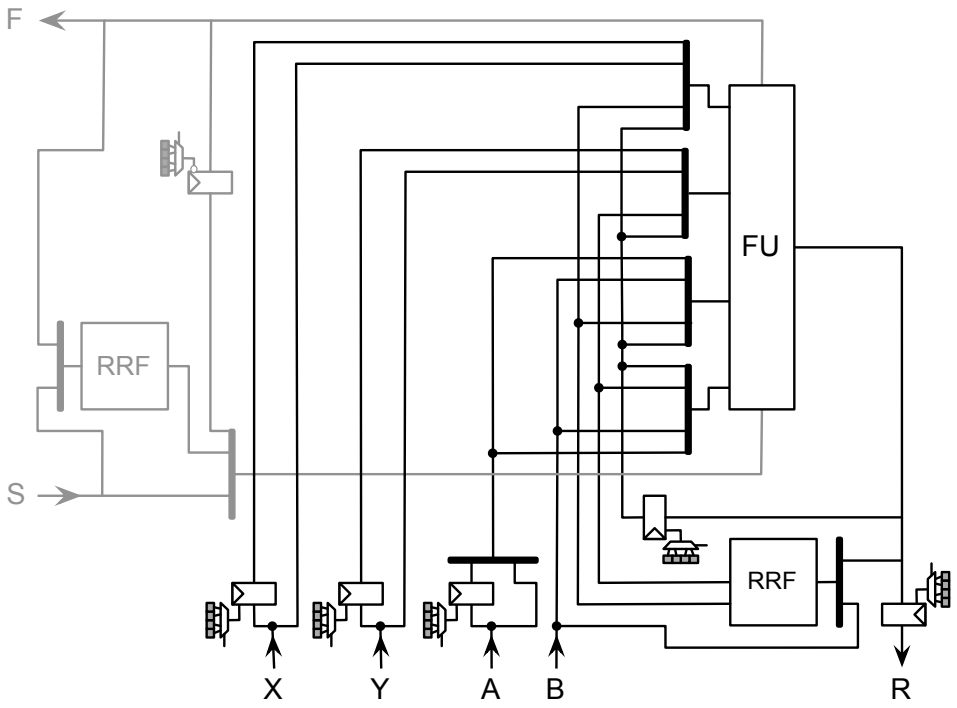


Figure 9.9: Revised block diagram of optimized PE with Universal FU and dual-ported local rotating register file.

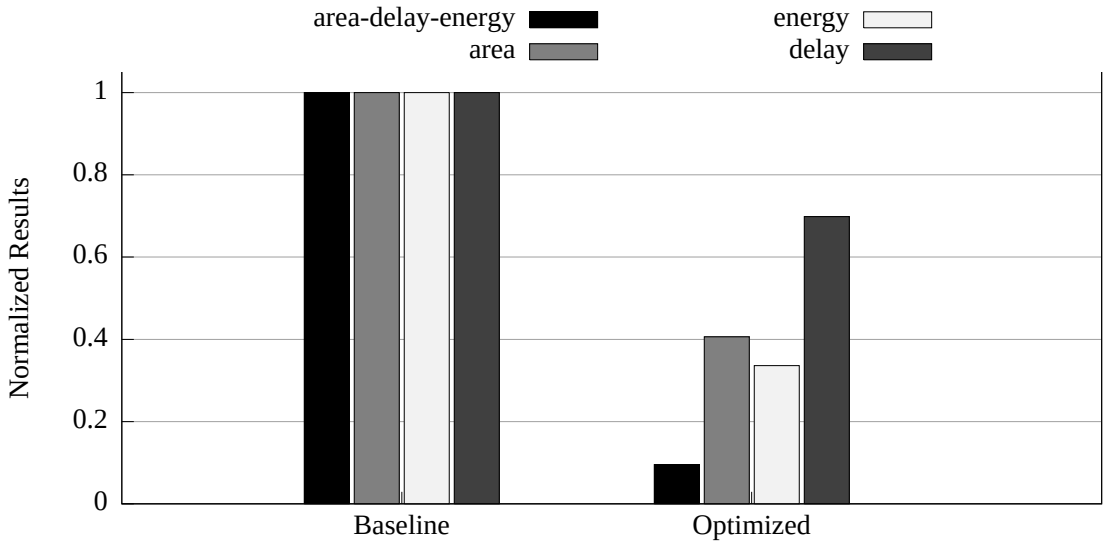


Figure 9.10: Average area-delay-energy product for final baseline and optimized architectures. Each metric is normalized to the baseline architecture

small area consumed by the global interconnect, it accounts for a significant percentage of the overall energy, as seen in Figure 9.13.

Finally, Figure 9.13 presents the distribution of energy between the core logic (excluding the crossbar), the intra-cluster crossbar, and the top-level grid interconnect. Each bar is normalized to the core energy of the baseline architecture. As we would expect, the optimized architecture uses substantially less energy in both the interconnect and crossbar. It is interesting to note that in the baseline architecture the average energy distribution is 31% computation and 69% communication, which is very similar to the energy distribution of FPGAs [21]. Due to the success of optimizing the interconnect, the optimized architecture has essentially reversed the energy distribution and is now 62% computation and 38% communication.

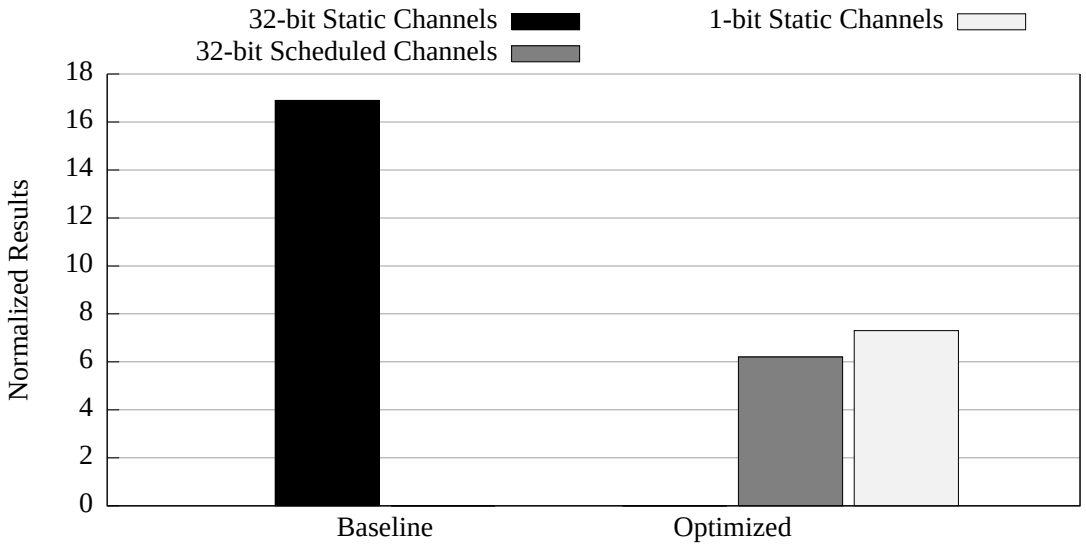


Figure 9.11: Average channel width of the grid interconnect for final baseline and optimized architectures.

9.6 Looking Beyond the Rim

This dissertation has shown that it is possible to dramatically improve the energy efficiency of coarse-grained reconfigurable arrays via systematic evaluation that isolates and identifies the impact that individual architectural features have on a CGRA's area, delay, and energy. Furthermore, we have shown that it is possible to compose these features, which compounds their improvements, and that a methodical approach can lead to significant improvements in designing an energy-efficient CGRA. The rest of this chapter will discuss the impact of these observations, as well as possible areas of future research.

9.6.1 CGRAs: Impact beyond research?

Coarse-grained reconfigurable arrays have been actively researched off and on over the past 20 years, but have yet to find broad commercial adoption. Despite a number of failed startups and engineering efforts from large companies, the most significant commercial efforts have been the adoption of hard-macro multipliers and embedded block RAMs in modern

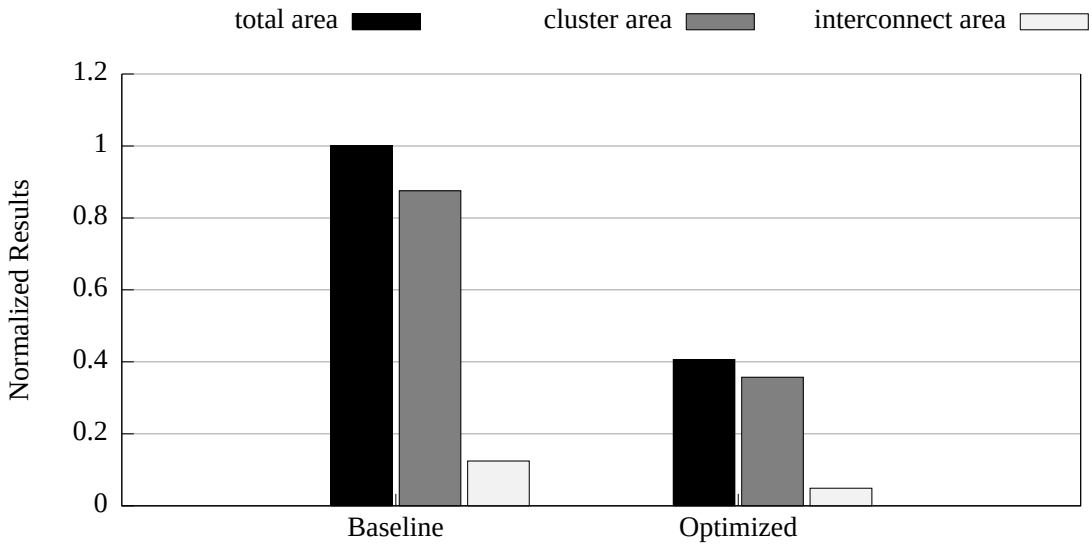


Figure 9.12: Average area distribution for final baseline and optimized architectures. All are results are normalized to the total area for the baseline architecture.

FPGAs. Aside from a number of business reasons, two technical hurdles to widespread adoption have been 1) the challenges of adopting a new programming environment and associated toolchain, which is typically immature or nonexistent; and 2) a lack of sufficiently impressive and demonstrable advantages in either performance, price, or energy efficiency over incumbent technologies. The challenges of creating a favorable development environment has not been addressed in this work, although the Macah language, which is part of the Mosaic project, has explored the advantages of a C-level language and compiler optimizations for CGRAs.

This work helps demonstrate a compelling advantage for CGRAs. In the past, it has been repeatedly shown that CGRAs provide significant speedup and area advantages over existing architectures such as FPGAs and VLIW processors. However, these advantages have not yet been sufficient to spur a successful commercial adoption of the CGRA class of spatial accelerators. A large part of the challenge is that, for raw performance, FPGAs are typically competitive solutions that can be obtained from well established vendors. Fur-

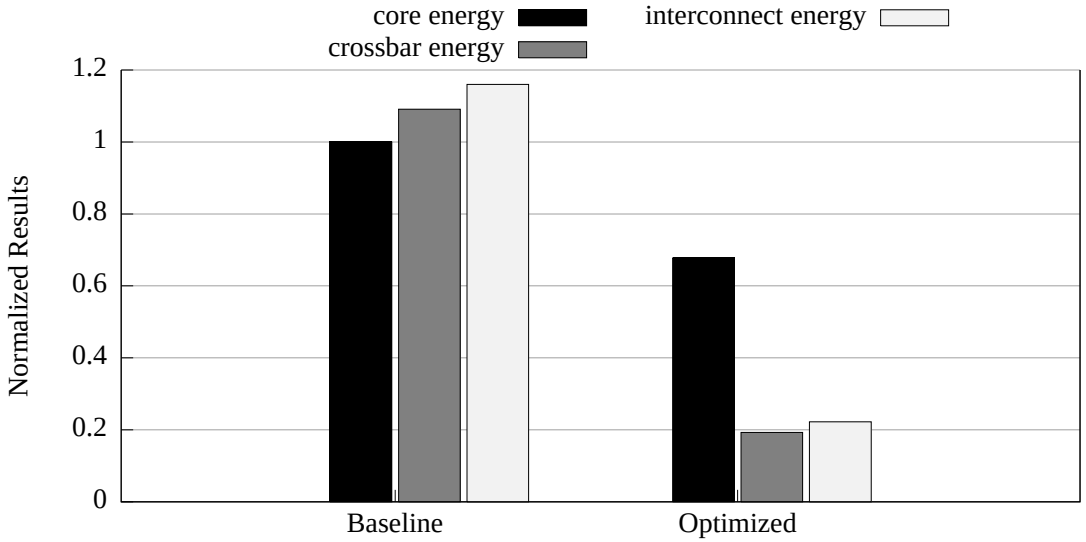


Figure 9.13: Average energy distribution for final baseline and optimized architectures. All energy results are normalized to the core energy for the baseline architecture.

thermore, as an existing commercial solution, unless there is a sufficiently large performance delta between a CGRA and an FPGA, a single advancement in the silicon process generation typically closes any performance gap. There is renewed hope that CGRAs may find future commercial development now that power consumption and energy efficiency are first-order design constraints. When comparing both performance and energy efficiency, CGRA's typical performance advantage over an FPGA should increase to more than an order of magnitude advantage over an FPGA. This would result in a gap that is not easily closed by waiting for a new process technology. This dissertation widens the combined performance-energy gap between CGRAs and FPGAs by providing a systematic methodology for CGRA design that is focused on further improving the architecture's energy efficiency without sacrificing performance.

9.6.2 Influencing other Spatial Architectures

Aside from the wholesale adoption or development of CGRA architectures, this work can have impact on the architecture and evolution of FPGAs or MPPAs. As noted earlier, FPGAs are no longer simple arrays of LUTs and wires; they are complex architectures that incorporate hard-macro blocks such as embedded multipliers and block RAMs. In an effort to reduce the cost of an FPGA's flexibility, it is possible to incorporate several design patterns from CGRAs into FPGAs. One such approach would be to coarsen a majority of the FPGAs datapath to process words of data (rather than bits of data) while staying with a configurable architecture, rather than a reconfigurable architecture that time-multiplexes on a cycle-by-cycle basis. Such an architecture could leverage these experimental results for both storage distribution and specialization of functional units. This evolution of FPGAs is potentially more likely because these new devices can leverage most of the existing FPGA ecosystem, including toolchains and programming models, while a CGRA requires a new toolchain.

Massively Parallel Processing Arrays are another spatial accelerator that has seen promising research results, but has yet to achieve significant commercial adoption. With lots of control domains, but simple processors, they are optimized for applications with tightly-coupled, thread-level parallelism. One potential source for that type of parallelism are computationally-intensive inner loops, similar to what CGRAs accelerate via pipelining. The cost of the MPPA's execution model is the overhead from the large number of control domains, a cost that CGRAs have shown does not need to be incurred for many applications. Embedding small CGRA-style compute fabrics into different regions of an MPPA, and using a reduced set of MPPA control domains is a potentially interesting architecture development that could improve the energy efficiency of MPPAs for loop-level parallelism. This type of exploration is just getting underway with the Mosaic 2 research project.

9.7 Future Work

Beyond the nitty-gritty details of optimizing a CGRA architecture for energy efficiency, this dissertation is part of a larger class of research that seeks to increase the opportunities for

using spatial accelerators to solve computationally-intensive applications. This work has focused on application domains that contain pipelinable loop-level parallelism, but many of the challenges that were encountered and resolved during the course of this research are systemic to all spatial accelerators. One guiding principle (for future research) is dictated by the limitations of today's semiconductor manufacturing processes, which demand that energy efficiency is the new world order of computer architecture design. Whereas in the past performance was king, it is now the ability to solve a problem while requiring less power that is driving both architectural and system-level innovations. In this vein, this research has explored the design space for one type of spatial accelerator. The same guiding principles and methodology can, and should, be applied to other classes of architectures.

FPGAs are a primary candidate for further exploration, especially when they are used for accelerating streaming data and other word-wide applications. While a tremendous amount of research has dramatically improved the energy efficiency of FPGAs over the last decade, architecturally they are still optimized for solving bit-level computations. For many classes of algorithms, this flexibility is wasted and translates into a significant amount of overhead. Coarsening the datapath of an FPGA is in fact fairly straight-forward; however, doing so in a manner that leverages decades worth of tools research and designer expertise has proven to be challenging. One interesting research path would be to explore a scalable FPGA architecture that has a native word-width of a small number of bits (*i.e.* 4-bits), that is coupled with a toolchain which allows each resource to be used as an expensive single-bit datapath, or chain multiple resources together to form a larger word-wide datapath. Such an effort could potentially either leverage or revisit older projects like Garp [83], Chess [84], or Elixent [85] and see what advantages these older designs have when energy efficiency is a primary concern. Using this approach, the entire word-wide datapath becomes more efficient than creating a word-wide datapath on a traditional FPGA. The trade-off is that single-bit operations and signals will underutilize the multi-bit resources and thus the single-bit control-path will incur an increased overhead. Developing and refining such an architecture would require a methodology and flow very similar to the one used in this work.

A sticky wicket that frequently appears with coarser datapaths is a push towards a fixed frequency architecture. The trade-off of balancing clock frequency versus logic depth

forces designers to adopt more highly pipelined design styles. As logic depth per cycle is decreased there are fewer resource that are likely to be underutilized within a given pipeline stage, but applications will require a high degree of pipelining. As a result of increased pipelining, more resources may lie idle if there are long loop-carried dependencies that cross multiple pipeline stages. This is one of many reasons that CGRAs use a time-multiplexed architecture. If the logic depth per cycle is increased, more computation is performed per cycle and thus a given application will typically have a shallower pipeline. As a result, loop-carried dependencies will typically cross fewer pipeline stages. However, there is additional resource fragmentation from the additional resources per logic stage, if they are unused. Some of these trade-offs can be better balanced by the flexible frequency approach of the FPGA, but it is paid for in the abundance of routing and storage resources required to support this flexibility. Therefore, coarsening the datapath of an FPGA is not just an issue of replacing LUTs with ALUs and wires with buses, but it will require an interesting design trade-off between a fixed versus flexible frequency architecture. The ramifications of a fixed frequency, pipelined architecture will also impact the CAD tools as they would have to accomodate pipelined routing instead of timing closure.

Another candidate for further exploration are structured ASICs, which are a combination of a semiconductor fabrication process technology and a streamlined manufacturing / engineering design flow. Given the ever rising cost and complexity of traditional ASICs, a structured ASIC is a pre-fabricated substrate of primitive computing and communication blocks that is customized by specializing the top metal layers or interconnect vias. This allows a customer to purchase an ASIC-like device, where much of the cost and development effort has been amortized across multiple customers. To-date the computing resources of commercially available structured ASICs are very similar to FPGA resources, focusing on bit-level computation and flexibility. Using the Mosaic architecture as a design template, it would be possible to create a coarse-grained structured ASIC fabric that is optimized for datapath computations. Such an exploration could leverage anything from the design of the interconnect, storage, and functional units, to methods of incorporating time-multiplexing into the substrate of the structured ASIC. As an alternative, rather than taking design features from the CGRA and incorporating them into the structured ASIC,

it would be possible to build a CGRA that is also a structured ASIC. The advantages of building a CGRA architecture in conjunction with a structured ASIC process, are that the architecture could provide additional options for domain specialization that would then be assembled, or enabled / disabled at design time.

In addition to the broader impact on the field of spatial accelerators, there are two direct threads of future work that are enabled by this dissertation: further development and optimization of Mosaic CGRA architectures, and leveraging the Mosaic infrastructure. Future research possibilities along the first thread are broken down into three directions that are addressed in subsequent sections: direct optimization of the Mosaic CGRA, developing a hybrid architecture that combines CGRA and MPPA architectural elements, and using the Mosaic CGRA architecture as a virtual accelerator that is then targeted for using FPGAs. The second thread makes use of the infrastructure that was developed during the course of this research to evaluate the advantages of CGRAs with respect to other spatial accelerators, and is discussed in the last section.

9.7.1 Refining the Mosaic CGRA

The most immediate future work would be to continue to refine and optimize the Mosaic CGRA. The initial candidates for optimizations are the stream I/O ports and the embedded block memories in the array. Optimizing the stream I/O ports would require creating more complete models and then integrating them with resources outside of the core, such as the stream accessor engines or direct memory access (DMA) engines. For the memories, one open question that we have encountered is the development of an efficient method for logically partitioning large memories into smaller logical arrays. Automating this kind of partitioning is extremely challenging to do well, and requires an understanding of the algorithm's dataflow. Therefore, preliminary work would focus on providing a clear mechanism for an application programmer to annotate large arrays, so that the CAD tools can perform a guided partitioning. An additional challenge is to aggregate multiple smaller memories into one larger, logical array. Unlike the partitioning problem, this task is more amenable to automation by the CAD tools. Implementing a memory packer is more of an engineer-

ing challenge (than a research problem) that could leverage existing placement techniques, and an algorithm to provide translation between an application's array indexing and the physical location of the array.

9.7.2 Evolving the CGRA: moving towards an MPPA

The Mosaic 2 research project builds on the work in this dissertation and the rest of the Mosaic project. The architectural goals of the Mosaic 2 project is to develop a spatial accelerator that is a hybrid CGRA+MPPA. Particularly, its intent is to leverage the Mosaic CGRA's energy efficiency for exploiting loop-level parallelism and the control flexibility of an MPPA. The two key steps for this are carving up the CGRA into multiple control domains, and adding flow-controlled communication channels between independent domains of control within a larger CGRA fabric. To avoid incurring significant overhead, there are three aspects of the design that have to be balanced: the ratio of control domains to computing resources, the flexibility of each domain, and how much the influence of each domain overlaps with neighboring domains. Establishing the ratio of control domains to compute resources determines the size of a statically scheduled region in the architecture, and should be correlated to the characteristics of common application's dataflow graphs. To provide flexibility and robustness for the architecture, regions can use a composable design style that allows multiple control domains to be combined into a single statically scheduled region. Other associated research challenges are developing CAD tools to floorplan the architecture and coordinate use of resources shared between multiple control domains.

9.7.3 Using the Mosaic CGRA as a Virtual Accelerator Platform

While the Mosaic CGRA architecture is an energy efficient spatial accelerator, it can also be viewed as a target for accelerator languages such as Macah, which has also been developed as part of the Mosaic project. The intention of the Macah language is to provide a C-level language for spatial accelerators. It provides C-style management of resources and parallelism, and is primarily targeted towards CGRAs. One approach to making FPGAs more accessible as an application accelerator would be to program them via Macah, and use

a variant of the Mosaic CGRA architecture as a virtual accelerator platform that is then implemented on the FPGA. Specifically, the current Mosaic CGRA architecture could be redesigned and optimized for executing on an FPGA platform. Then, rather than synthesizing an application directly for the FPGA fabric, it would be compiled from Macah to this new, FPGA-friendly, Mosaic CGRA architecture. To execute the application kernel, the Mosaic CGRA would be synthesized to the FPGA and the application would be loaded into the Mosaic CGRA's configuration memory. Such an approach could provide superior application performance on an FPGA by fine-tuning the CGRA mapping to the FPGA architecture, similar to how high performance floating-point libraries have been developed for FPGAs [86]. Furthermore, because the CGRA fabric is developed as a library, it offloads problems such as timing closure from the application developer onto the library designer, by using pre-defined and pre-optimized hardware structures. As a result, the use of a virtual accelerator platform for FPGAs would relieve the burden of hardware design from the programmer, allowing them to focus on the challenge of targeting a spatial accelerator.

9.7.4 Establishing the advantages of a CGRA

Comparing the effectiveness of spatial accelerators, especially those with significantly different architectures, is a difficult and ongoing problem. Considering that CGRAs, MPPAs, GP-GPUs, and FPGAs, are all optimized to exploit specific (and possibly different) types of parallelism, providing an effective and fair comparison of two architectures is tricky, but mostly time consuming. However, providing a benchmark suite for comparing CGRAs to FPGAs, GP-GPUs, and MPPAs would be extremely valuable. The key challenges for such a suite would be to provide implementations for each platform with similar levels of algorithmic and tool optimizations. Once the implementation and the mapping of each benchmark to each architecture achieve parity across architectures, it is possible to measure how effective each architecture is at exploiting different types of parallelism. Furthermore, identifying the types of parallelism within each benchmark and correlating it with performance on each accelerator would provide valuable insight for future architects.

9.8 In closing

The landscape of interesting applications demands increasing performance with a decreasing energy budget, and is ready for a new type of spatial accelerator to address this challenge. This need for energy-efficient computing ranges from battery powered embedded system up to large-scale supercomputers. Coarse-grained reconfigurable arrays have tantalized the community with significant performance advantages, but now offer a compelling solution by combining high-performance and high energy efficiency for exploiting loop-level parallelism. This work provides an exploration into the design-space of CGRAs and identifies architectural optimizations that further increase the value-proposition of CGRAs by dramatically improving their energy efficiency.

BIBLIOGRAPHY

[1] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments," in *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 315–326.

[2] Z. Baker, M. Gokhale, and J. Tripp, "Matched Filter Computation on FPGA, Cell and GPU," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 23-25 April 2007, pp. 207–218.

[3] A. Lambrechts, *et al.*, "Design style case study for embedded multi media compute nodes," in *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, 5-8 Dec. 2004, pp. 104–113.

[4] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2007, pp. 13–23.

[5] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Reed Taylor, "PipeRench: A virtualized programmable datapath in 0.18 micron technology," in *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference*, 12-15 May 2002, pp. 63–66.

[6] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, A. Jung Ho, P. Mattson, and J. D. Owens, "Programmable stream processors," *Computer*, vol. 36, no. 8, p. 54, 2003, 0018-9162.

[7] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," in *26th International Symposium on Computer Architecture (ISCA99)*, 1999.

[8] A. DeHon, "The Density Advantage of Configurable Computing," *Computer*, vol. 33, no. 4, pp. 41–49, 2000.

[9] M. Gokhale, J. Frigo, K. McCabe, J. Theiler, C. Wolinski, and D. Lavenier, "Experience with a Hybrid Processor: K-Means Clustering," *The Journal of Supercomputing*, vol. 26, no. 2, pp. 131–148, 2003.

[10] J. L. Tripp, H. S. Mortveit, A. A. Hansson, and M. Gokhale, "Metropolitan Road Traffic Simulation on FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. Los Alamitos, CA, USA: IEEE Computer Society, April 2005, pp. 117–126.

[11] R. Scrofano, M. Gokhale, F. Trouw, and V. Prasanna, "Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 23–34.

[12] B. Ylvisaker, B. Van Essen, and C. Ebeling, "A Type Architecture for Hybrid Micro-Parallel Computers," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, April 2006, pp. 99–110.

[13] M. Butts, A. Jones, and P. Wasson, "A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2007, pp. 55–64.

[14] D. Helgemo, "Digital signal processing at 1 GHz in a field-programmable object array," in *Proceedings of the IEEE International [Systems-on-Chip] SOC Conference*, 17-20 Sept. 2003, pp. 57–60.

[15] S. Kelem, B. Box, S. Wasson, R. Plunkett, J. Hassoun, and C. Phillips, "An Elemental Computing Architecture For SD Radio," Elemental CXI, Tech. Rep., 2007. [Online]. Available: http://www.elementcxi.com/

[16] D. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, "Architecture design of reconfigurable pipelined datapaths," in *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, Atlanta, 1999, pp. 23–40.

[17] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[18] C. Ebeling, C. Fisher, G. Xing, M. Shen, and H. Liu, "Implementing an OFDM Receiver on the RaPiD Reconfigurable Architecture." in *International Conference on Field-Programmable Logic and Applications*, ser. Lecture Notes in Computer Science, P. Y. K. Cheung, G. A. Constantinides, and J. T. de Sousa, Eds., vol. 2778. Springer, 2003, pp. 21–30.

[19] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[20] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 2006, pp. 21–30.

[21] K. K. W. Poon, S. J. E. Wilton, and A. Yan, "A detailed power model for field-programmable gate arrays," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 2, pp. 279–302, 2005.

[22] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon, "HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. ACM Press, 1999, pp. 125–134.

[23] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, 1997, pp. 22–28.

[24] M. Mishra and S. C. Goldstein, "Virtualization on the Tartan Reconfigurable Architecture," in *FPL 2007. International Conference on Field Programmable Logic and Applications, 2007.*, 2007, pp. 323–330.

[25] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996, pp. 157–166.

[26] G. Sassatelli, L. Torres, J. Galy, G. Cambon, and C. Diou, "The Systolic Ring: A Dynamically Reconfigurable Architecture for Embedded Systems," *Field-Programmable Logic and Applications*, vol. 2147, pp. 409–419, 2001.

[27] U. Nageldinger, "Coarse-grained Reconfigurable Architectures Design Space Exploration," Ph.D. dissertation, Kaiserslautern University, 2001.

[28] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. New York, NY, USA: ACM Press, 1988, pp. 318–328.

[29] B. R. Rau, "Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*. ACM Press, 1994, pp. 63–74, San Jose, California, United States.

[30] C. Ebeling, D. Cronquist, and P. Franklin, "Configurable computing: the catalyst for high-performance architectures," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 1997, pp. 364–372.

[31] D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and Compiling Applications for RaPiD," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. IEEE Computer Society Press, 1998, pp. 116–125.

[32] A. M. Ghuloum and A. L. Fisher, "Flattening and parallelizing irregular, recurrent loop nests," *SIGPLAN Not.*, vol. 30, no. 8, pp. 58–67, 1995.

[33] C. D. Polychronopoulos, "Loop Coalescing: A Compiler Transformation for Parallel Machines," in *Proc. International Conf. on Parallel Processing*, August 1987, pp. 235–242.

[34] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," in *Proc. IEEE Computer Society Fourth International Computer Software and Applications Conf.*, October 1980.

[35] K. Kennedy and K. S. McKinley, "Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution," in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. London, UK: Springer-Verlag, 1994, pp. 301–320.

[36] K. Puttegowda, D. I. Lehn, J. H. Park, P. Athanas, and M. Jones, "Context Switching in a Run-Time Reconfigurable System," *The Journal of Supercomputing*, vol. 26, no. 3, pp. 239–257, Nov. 2003.

[37] H. Schmit, "Incremental reconfiguration for pipelined applications," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, vol. 00. Los Alamitos, CA, USA: IEEE Computer Society, 1997, p. 47.

[38] H. H. Schmit, S. Cadambi, M. Moe, and S. C. Goldstein, "Pipeline Reconfigurable FPGAs," *The Journal of VLSI Signal Processing*, vol. 24, no. 2, pp. 129–146, Mar. 2000.

[39] A. Abnous and J. Rabaey, "Ultra-low-power domain-specific multimedia processors," in *Workshop on VLSI Signal Processing*, vol. IX, 30 Oct-1 Nov 1996, pp. 461–470.

[40] R. A. Bittner, Jr., P. M. Athanas, and M. D. Musgrove, "Colt: an experiment in wormhole runtime reconfiguration," in *Proc. of SPIE 2914 – High-Speed Computing, Digital Signal Processing, and Filtering Using reconfigurable Logic*, J. Schewel, Ed. Bellingham, WA: SPIE – The International Society for Optical Engineering, 1996, pp. 187–195.

[41] S. Singh and P. James-Roxby, "Lava and JBits: From HDL to Bitstream in Seconds," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001, pp. 91–100.

[42] M. B. Taylor, *et al.*, "The Raw microprocessor: a computational fabric for software circuits and general-purpose programs," *Micro, IEEE*, vol. 22, no. 2, p. 25, 2002.

[43] M. D. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal, "Scalar operand networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 2, p. 145, 2005.

[44] M. Gokhale and J. Stone, "NAPA C: Compiling for Hybrid RISC/FPGA Architecture," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.

[45] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 49–56.

[46] D. Pellerin and S. Thibault, *Practical FPGA Programming in C.* Prentice Hall PTR, April 2005.

[47] A. Sharma, K. Compton, C. Ebeling, and S. Hauck, "Exploration of Pipelined FPGA Interconnect Structures," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, 2004, pp. 13–22.

[48] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an architecture-adaptive CGRA mapping tool," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* New York, NY, USA: ACM, 2009, pp. 191–200.

[49] B. Ylvisaker, A. Carroll, S. Friedman, B. V. Essen, C. Ebeling, D. Grossman, and S. Hauck, ""Macah: A "C-Level" Language for Programming Kernels on Coprocessor Accelerators ", 2008." University of Washington, Tech. Rep., 2008.

[50] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671–680, 1983.

[51] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications.* London, UK: Springer-Verlag, 1997, pp. 213–222.

[52] L. McMurchie and C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs," in *Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays.* Monterey, California, United States: ACM Press, 1995, pp. 111–117. [Online]. Available: http://doi.acm.org/10.1145/201310.201328

[53] S. Li and C. Ebeling, "QuickRoute: A Fast Routing Algorithm for Pipelined Architectures," in *IEEE International Conference on Field-Programmable Technology*, Queensland, Australia, 2004, pp. 73–80. [Online]. Available: http://ieeexplore.ieee.org/iel5/9585/30303/01393253.pdf?tp=&arnumber=1393253&isnumber=30303

[54] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100 –107, july 1968.

[55] Sun Microsystems and Static Free Software, "Electric VLSI Design System." [Online]. Available: http://www.staticfreesoft.com/

[56] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," in *IEEE International Symposium on Computer Architecture*, 2000, pp. 83–94.

[57] A. Chandrakasan and R. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proceedings of the IEEE*, vol. 83, no. 4, pp. 498–523, 1995.

[58] M. Barocci, L. Benini, A. Bogliolo, B. Ricco, and G. De Micheli, "Lookup table power macro-models for behavioral library components," in *IEEE Alessandro Volta Memorial Workshop on Low-Power Design*, Mar. 1999, pp. 173 –181.

[59] S. Gupta and F. Najm, "Power modeling for high-level power estimation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 1, pp. 18 –29, Feb. 2000.

[60] R. A. van de Geijn and J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[61] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 1998, pp. 191–200.

[62] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic transformations in the implementation of K-means clustering on reconfigurable hardware," in *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM Press, 2001, pp. 103–110.

[63] C. Zhu, X. Lin, L. Chau, and L.-M. Po, "Enhanced Hexagonal Search for Fast Block Motion Estimation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 14, no. 10, pp. 1210–1214, October 2004.

[64] T. Oliver, B. Schmidt, and D. Maskell, "Hyper customized processors for bio-sequence database scanning on FPGAs," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 2005, pp. 229–237.

[65] M. Haselman, R. Miyaoka, T. K. Lewellen, and S. Hauck, "Fpga-based data acquisition system for a positron emission tomography (PET) scanner," in *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2008, pp. 264–264.

[66] B. Ylvisaker, ""C-Level" Programming of Parallel Coprocessor Accelerators," Ph.D. dissertation, University of Washington, September 2010.

[67] F. J. Bouwens, "Power and Performance Optimization for ADRES," MSc Thesis, Delft University of Technology, August 2006.

[68] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, "Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, S. B. . Heidelberg, Ed., vol. Volume 4419/2007, March 2007, pp. 1–13.

[69] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, "Resource Sharing and Pipelining in Coarse-Grained Reconfigurable Architecture for Domain-Specific Optimization," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 12–17.

[70] S. Wilton, N. Kafafi, B. Mei, and S. Vernalde, "Interconnect architectures for modulo-scheduled coarse-grained reconfigurable arrays," in *Proceedings of the 2004 IEEE International Conference on Field-Programmable Technology*, 6-8 Dec. 2004, pp. 33–40.

[71] A. Papanikolaou, F. Starzer, M. Miranda, K. De Bosschere, and F. Catthoor, "Architectural and Physical Design Optimizations for Efficient Intra-tile Communication," in *Proceedings of the 2005 International Symposium on System-on-Chip*, 17-17 Nov. 2005, pp. 112–115.

[72] H. Lange and H. Schroder, "Evaluation strategies for coarse grained reconfigurable architectures," in *International Conference on Field-Programmable Logic and Applications*, 24-26 Aug. 2005, pp. 586–589.

[73] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in *International Conference on Field-Programmable Logic and Applications*, vol. 2778, Lisbon, Portugal, 2003, pp. 61–70, 2003.

[74] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD - Reconfigurable Pipelined Datapath," in *International Workshop on Field-Programmable Logic and Applications*, R. W. Hartenstein and M. Glesner, Eds. Springer-Verlag, Berlin, 1996, pp. 126–135.

[75] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures," in *IEEE International Conference on Field-Programmable Technology*, 2002, pp. 166–173.

[76] A. Lambrechts, P. Raghavan, M. Jayapala, B. Mei, F. Catthoor, and D. Verkest, "Interconnect Exploration for Energy Versus Performance Tradeoffs for Coarse Grained Reconfigurable Architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 17, no. 1, pp. 151–155, Jan. 2009.

[77] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and single-driver wires in FPGA interconnect," in *IEEE International Conference on Field-Programmable Technology*, Dec. 2004, pp. 41–48.

[78] N. Clark, A. Hormati, and S. Mahlke, "VEAL: Virtualized Execution Accelerator for Loops," in *IEEE International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 389–400.

[79] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 183–198.

[80] J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1989, pp. 26–38.

[81] B. Van Essen, R. Panda, C. Ebeling, and S. Hauck, "Managing short versus long lived signals in Coarse-Grained Reconfigurable Arrays," in *International Conference on Field-Programmable Logic and Applications*. Milano, Italy: IEEE, Aug. 31 - Sept. 2 2010, pp. 380–387.

[82] N. McVicar, C. Olson, and J. Xu, "Power Exploration for Functional Units in Coarse-Grained Reconfigurable Arrays," University of Washington, Tech. Rep. UWEETR-2010-0005, 2010. [Online]. Available: https://www.ee.washington.edu/techsite/papers/refer/UWEETR-2010-0005.html

[83] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *Computer*, vol. 33, no. 4, p. 62, 2000, 0018-9162.

[84] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 1999, pp. 135–143.

[85] A. Marshall, "Elixent adds routing to tackle control in signal processing," *Electronics Systems and Software*, vol. 3, no. 3, pp. 47–47, June/July 2005.

[86] W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood, "A re-evaluation of the practicality of floating-point operations on FPGAs," in *IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1998, pp. 206 –215.

[87] T. Miyamori and K. Olukotun, "REMARC: reconfigurable multimedia array coprocessor," in *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 1998, p. 261.

[88] A. Alsolaim, J. Becker, M. Glesner, and J. Starzyk, "Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communication Systems," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, pp. 205–214.

[89] A. Abnous, H. Zhang, M. Wan, G. Varghese, V. Prabhu, and J. Rabaey, "The Pleiades Architecture," in *The Application of Programmable DSPs in Mobile Communications*, A. Gatherer and A. Auslander, Eds. Wiley, 2002, pp. 327–360.

[90] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in *13th International Conference on Field Programmable Logic and Applications*, vol. 2778, Lisbon, Portugal, 2003, pp. 61–70.

[91] P. Benoit, L. Torres, G. Sassatelli, M. Robert, and G. Cambon, "Dynamic hardware multiplexing for coarse grain reconfigurable architectures," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 2005, pp. 270–270.

[92] P. Benoit, L. Torres, G. Sassatelli, M. Robert, G. Cambon, and J. Becker, "Dynamic Hardware Multiplexing: Improving Adaptability with a Run Time Reconfiguration Manager," *IEEE Computer Society Annual Symposium on VLSI: Emerging VLSI Technologies and Architectures*, vol. 0, pp. 251–256, 2006.

[93] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera reconfigurable functional unit," in *Symposium on FPGAs for Custom Computing Machines*, 1997, pp. 87–96.

[94] A. Wang, "The Stretch Architecture: Raising the Level of Productivity and Compute Efficiency," Keynote Speech, 6th WorkShop on Media and Streaming Processors, December 2004.

[95] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*. New York, NY, USA: ACM, 1994, pp. 172–180.

[96] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, Dec. 1991.

[97] H. Schmit, B. Levine, and B. Ylvisaker, "Queue machines: hardware compilation in hardware," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, pp. 152–160.

[98] B. A. Levine and H. H. Schmit, "Efficient Application Representation for HASTE: Hybrid Architectures with a Single, Transformable Executable," in *IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2003, p. 101.

[99] R. A. Sutton, V. P. Srini, and J. M. Rabaey, "A Multiprocessor DSP system using PADDI-2," in *Design Automation Conference*, vol. 00. Los Alamitos, CA, USA: IEEE Computer Society, 1998, pp. 62–65.

[100] A. Yeung and J. Rabaey, "A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput DSP algorithms," *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences*, vol. i, pp. 169–178 vol.1, 5-8 Jan 1993.

[101] D. Chen and J. Rabaey, "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 12, pp. 1895–1904, Dec 1992.

[102] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, D. Truong, T. Mohsenin, and B. Baas, "AsAP: An Asynchronous Array of Simple Processors," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 43, no. 3, pp. 695–705, Mar. 2008.

Appendix A

DESIGN PRINCIPLES OF COARSE-GRAINED RECONFIGURABLE ARRAYS AND OTHER SPATIAL ARCHITECTURES

This appendix explores several coarse-grained reconfigurable array research projects to identify key architectural features and common design pattens. Some details of each architecture have been omitted in order to highlight the features of interest for designing a new CGRA. Each architecture is introduced below, with a brief statement of its design philosophy, while its salient features are summarized in Tables A.1, A.2, and A.3. In general these architectures share a number of common features that were discussed in §2.1. Table A.1 places each architecture in the context of these common features. Table A.2 is an overview of the available resources for each architecture. Finally, the interesting features of each architecture are shown in Table A.3 and presented in §A.1.

MATRIX MATRIX [25] is designed to 1) blend configuration-path, control-path, and datapath resources, 2) support static and dynamic configuration, and 3) provide a scalable number of control domains. Resources are efficiently balanced by constructing configuration-path, control-path, and datapath from the same pool of processing elements.

MorphoSys MorphoSys [19] is designed for general data-parallel applications, but with a specialization on imaging application domain.

REMARC The Reconfigurable Multimedia Array Coprocessor [87] is designed to offload multimedia kernels from a standard RISC processor.

DReAM Dynamically Reconfigurable Architecture for Mobile Systems [88] was a research project intended to accelerate digital baseband processing for mobile radio devices.

Pleiades The Pleiades [39, 89] CGRA is actually an architectural template, rather than a single instance of an architecture. Processing elements in Pleiades are arranged in 2-D, but lack the strict layout of an array. The interconnect of Pleiades is a generalized-mesh, a result of a heterogeneous set of processing elements. The Maia CGRA [89] is an instance

Architecture	Topology	Word-width (bits)	Interconnect	Control Domains	Compute Model	Reconfiguration
MATRIX	2-D array	8	Nearest Neighbor (x2), H/V L4-Bypass, HBUS, VBUS	1-N	VLIW, SIMD, MSIMD	Static-Dynamic
MorphoSys	2-D array	16	Mesh, Intra-quadrant, HBUS, VBUS	1	HSIMD, VSIMD	Dynamic
REMARC	2-D array	16	Mesh, HBUS, VBUS	1	VLIW, HSIMD, VSIMD	Dynamic
DReAM	2-D array	8	Mesh, Grid (Island)	1	VLIW	Dynamic
Pleiades (Maia)	2-D gen.	16	Generalized-Mesh	1	VLIW	Dynamic
ADRES	2-D array	32	Mesh, HBUS, VBUS	1	VLIW	Dynamic
Systolic Ring (DHM)	Ring	16	Fwd/Rev pipelined ring	1 or N	VLIW, MIMD	Dynamic
PipeRench	Linear array	8	Feed-forward pipeline, Horiz. crossbar, VBUS	1	VLIW	Dynamic
RaPiD	Linear array	16	Segmented tracks	4	VLIW	Static-Dynamic

Table A.1: CGRA Trademark Characteristics

Architecture	Avg. # PEs	ALUs per PE	Mem per PE	Configurations (Contexts)	Reconf. Mode
MATRIX	100	1	1	256	Time-multiplexed
MorphoSys	64	1	1	32	Context-switched
REMARC	64	1	1	32	Context-switched
DReAM	36 - 144	2	2	NR	Context-switched
Pleiades (Maia)	5	NR	NA	NR	Context-switched
ADRES	16 - 36	1	reg-file	128	Context-switched
Systolic Ring (DHM)	8	1	reg-file	Global - NR / Local - 6	Context-switched
PipeRench	256	1	reg-file	256	Pipeline reconfiguration
RaPiD	8-32	4	3	NR	Time-multiplexed

Table A.2: CGRA Resources and Reconfiguration Mode (NR - not reported, NA - not applicable)

Arch. Features	Section	MATRIX	MorphoSys	REMARC	DReAM	Pleiades	ADRES	RaPiD	PipeRench	Systolic Ring
Scalable Datapath	§A.1.1	√			√			√	√	
Scalable Configuration Resources	§A.1.2	√								
Multiple Control Domains	§A.1.3	√						√		√
Simplifying Compilation	§A.1.4		√		√	√	√	√	√	
Minimizing Cfg. Overhead	§A.1.5	√	√	√				√	√	
Optimizing Data Movement	§A.1.6		√	√				√	√	√
Domain specific operators	§A.1.7	√	√	√	√	√				
Integration with Seq. Proc.	§A.1.8		√	√		√	√			
Preserving Arith. Precision	§A.1.9		√					√		
Flexibility in Sched. & Exec.	§A.1.10				√	√	√			
Choreographing External I/O	§A.1.11		√					√		
Reducing Dynamic Power	§A.1.12					√				
Runtime Binding of Resources	§A.1.13								√	√

Table A.3: Architectures and list of features by section number

of Pleiades for speech coding applications.

ADRES Architecture for Dynamically Reconfigurable Embedded System [90, 75, 68] is a template architecture that is designed to simplify the task of programming for CGRAs by providing a shared memory model between the host processor and CGRA fabric.

Systolic Ring with Dynamic Hardware Multiplexing Dynamic Hardware Multiplexing (DHM) [91, 92] and the Systolic Ring [26] architecture are designed to manage task-level parallelism in a CGRA.

PipeRench The PipeRench [37, 38, 17, 5] architecture was designed to implement pipeline reconfiguration and create a platform for hardware (or pipeline) virtualization. The computation model is much more restrictive than other CGRAs surveyed here, with all feedback constrained to within a pipeline stage, it is practically limited to a feed-forward pipelined computation.

RaPiD RaPiD or reconfigurable pipelined datapaths [30, 31, 16] is designed to execute deeply nested, pipelined, computation loops (kernels). RaPiDs ability to efficiently execute nested and sequenced loops is unique among the field of surveyed CGRAs.

A.1 CGRA Design Principles & Architectural Features

This section presents a survey of key architectural features from existing CGRA research and commercial designs. Each of these features had been proposed to address specific weaknesses or expand the capabilities of CGRAs. I propose the following set of design principles that unify individual features based on the architectural constraints for which they optimize. Note that the details of some principles partially overlap, but each principle optimizes different aspects of CGRA design.

A.1.1 Scalable Datapaths (Extensible ALUs)

A scalable datapath is one that operates efficiently on data words larger than the native width of the CGRAs fabric. Structures such as ALUs, barrel shifters, and embedded memories can be designed for scalability. DReAM provides an example of a scalable embedded memory. Within each processing element are a pair of RAM units. One of the RAMs has a FIFO controller and dedicated communication to adjacent processing elements in the same

column. These communication paths allow DReAM to chain multiple RAMs and controllers together to create a deeper FIFO.

A scalable or extensible ALU can be constructed by cascading the carry-in and -out logic from one ALU to another, as well as shift-in and -out logic from one shifter to another. The carry and shift logic already exist internal to ALUs, to support multi-bit operations, and just needs to be made available externally. Exposing this logic is the simple part; the challenging part is how to route the output from one ALU to the input of another without undue delay. Special consideration is often given for this, since any interconnect delay factors directly into the speed of computing "primitive" arithmetic operations.

Architectures such as MATRIX, PADDI-2 (PADDI-1), and PipeRench use dedicated wiring between adjacent processing elements for these carry and shift signals. This approach provides high performance and avoids congestion with normal datapath communication, but these dedicated communication resources are wasted if an application does not have large multi-word operations. The other alternative, used by RaPiD, is to route these signals onto general purpose communication resources. This has the advantage that fewer resources are wasted when native word-width arithmetic operations are being executed. The disadvantage is that the "intra-ALU" delay is typically larger than systems using dedicated resources. RaPiD addresses this additional delay by pipelining the execution of these multi-word operators.

A.1.2 Scalable Configuration Resources

Partitioning transistor resources between the configuration-path and datapath is a common design challenge in CGRAs. Adding more configuration resources can increase the system's flexibility or add additional control domains, which may improve the system's performance on kernels with high control complexity. However, these resources typically come at the cost of datapath resources, which reduces the raw computational horsepower of the system. On the other hand, reducing the configuration resources can limit the flexibility of the system and potentially the number of kernels that can be efficiently executed. As mentioned earlier, FPGAs offer a high degree of flexibility in trading datapath resources for configuration-path

resources, at the expense of high configuration overhead.

MATRIX is a CGRA that, like an FPGA, unifies the set of configuration and datapath resources. To achieve this, it augments the processing elements with some logical reduction operators, a NOR plane (*i.e* one half of a PLA), and the ability to feed results into the configuration-path. This makes it possible to construct multiple control domains, and provide data driven feedback into the configuration of the processing elements and interconnect.

A.1.3 Multiple Control Domains

Control domains correspond to a thread of control within an application, as described in Section 2.1.2. Supporting multiple threads of control can increase the type of parallelism that can be exploited efficiently within a CGRA. One method of implementing multiple control domains is illustrated in MATRIX, where the configuration logic isn't fixed by the architecture and is implemented using the same set of resources as the datapath. Therefore, as the application demands it, more control domains can be constructed by the application. The limit to the number of domains is theoretically bound by the total number of processing elements, but in practice the number is substantially lower because datapath resources are still needed. This versatility enables MATRIX to augment its normal VLIW mode of operation with both SIMD and multiple-SIMD (MSIMD) modes. Multiple-SIMD is the combination of multiple control domains each executing in SIMD mode. The Systolic Ring architecture has a single global control domain and per processing element local domains. This permits both VLIW and MIMD execution modes.

A very different use of control domains is exhibited in the RaPiD architecture and introduced in Section 2.1.2. In the previous discussion of multiple control domains, each domain was independent and disjoint from each other, which was ideal for managing thread level parallelism. RaPiD proposed the use of multiple, tightly-coupled, control domains that could synchronize the execution of multiple sequenced loops. Additionally, each control domain provided a loop stack data structure that enabled a single domain to efficiently handle nested loops. The composition of these two features enabled RaPiD to execute

nested and sequenced loops within a single kernel very efficiently, thus expanding the range of loop-level parallelism that could be expressed as pipeline parallelism.

A.1.4 Designing to Simplify Compilation

A particular challenge of CGRAs is the task of programming them. From an application developer's perspective the problem is that they lie in between traditional hardware design and software programming. The basic arithmetic and logical operations are similar to traditional sequential processors, but the spatial aspect of their communication and the need to schedule, place, and route between operators is more like hardware design. One common approach to addressing this problem is to develop sophisticated tools that can transform sequential algorithms and code into spatial implementations. Vectorizing compilers have shown how to extract parallelism for many digital signal processing and scientific applications and map it to vector processors, but do not address the problem of mapping to spatial architectures. While the goal of automatic transformation is noble, the general case of trying to map from sequential to spatial implementations has met with limited success due to the intractability of the problem.

Several CGRA research efforts have attempted to improve the situation by making the architectures more amenable to compilation. One of these examples is the ADRES project. Observing that spatial communication is analogous to message passing in parallel programming, ADRES created a CGRA fabric that provides a shared memory model with the host processor. The host VLIW processor and CGRA fabric share the same multi-ported register file. This removes the need for the compiler to orchestrate marshaling the data between the host and CGRA. Furthermore, ADRES eliminates the overhead of transferring the data, thus providing additional opportunities to accelerate applications. Finally, issues of data synchronization are avoided when alternating execution of the host and the fabric. This approach is similar to those found in other fine-grained spatial architectures such as Chimaera [93], Stretch [94], or PRISC [95].

Another challenge of efficiently using a CGRA is finding an application kernel of sufficient size such that execution on a CGRA will provide overall acceleration. One tool that

helps the compiler form larger kernels is the support of predicated logic. Predicated logic takes a flag with each operation that indicates it should execute when that guard has been satisfied. Predication permits the compiler to perform if-conversion, hyperblock formation, and support prologue and epilogue code for loop pipelining [75]. Support for predication includes generating predicates from arithmetic operations, transporting predicates between processing elements, and observing predicates within stateful and stateless resources within the processing elements. ADRES and DReAM are examples of CGRAs that support predication.

Another challenge common to CGRAs is the fixed number of resources available to the application. CGRAs have a limited amount of configuration memory, and this limit will change from generation to generation of the CGRA. As a result, an application has to be compiled for a specific instance of a CGRA and its associated resource limits. Two approaches to mitigate this problem are pipeline virtualization used in PipeRench and online reloading of configuration memory in MorphoSys and Pleiades. Pipeline virtualization allows an application to be described as a compute pipeline of virtually unbound depth, although the width was limited by the architecture. These computational pipelines could then be mapped to a range of instances of PipeRench without recompilation. A key restriction of pipeline virtualization is that feedback must be contained within a single pipeline stage, thus producing a feed-forward pipelined algorithm. MorphoSys, DReAM and Pleiades offer online reloading of configuration memory to reduce the overhead of switching between kernels, by overlapping configuration loading with execution. As described in Section 2.1.2 online reloading also permits the host processor to feed a larger configuration into a limited size configuration memory, over time, which means that applications for MorphoSys are not limited to the size of their configuration memory.

As described in Section A.1.3 the RaPiD architecture provides multiple tightly-coupled control domains for the execution of nested and sequenced loop bodies. This same feature simplifies the task of the application programmer and compiler as they do not need to resort to using loop flattening or fusion to provide a single loop for the application kernel. Unlike many spatial architectures RaPiD is arranged as a 1-D linear array. Using dimensionality reduction techniques, many multidimensional algorithms can be transformed into linear

algorithms which may increase the similarity of communication patterns in the application with the communication channels in the architecture. Increasing this similarity can reduce the communication overhead when mapping application to architecture. However, these transformations typically add several levels of loop nesting and sequencing, which increase the value of multiple tightly-coupled control domains for RaPiD.

A.1.5 Minimizing Configuration Overhead

> There is no way to use **more** bits to encode an algorithm than as an FPGA configuration.
>
> –Mike Butts, Ambric Inc.

Having to move, manage, and store configuration data is an unfortunate overhead of reconfigurable architectures. For example, the configuration data for modern FPGAs is on the order of tens of millions of bits. Configuration memory and decoding logic takes up space and consumes energy, which strongly encourages minimizing configuration data. Additionally, with online reconfiguration, one dominating factor is the amount of time to move a new configuration into the CGRA fabric. For this reason, a number of CGRAs have proposed interesting methods for minimizing the size of the configuration data.

As described in Section 2.1.2, within a control domain the execution of a dataflow graph is choreographed by a single controller creating either a SIMD or VLIW mode of execution. To preserve flexibility, but reduce the amount of configuration data needed per processing element, some architectures have created a horizontal or vertical SIMD mode (HSIMD, VSIMD). In this mode a configuration instruction is broadcast to all processing elements within a single row or column. One example of this is found in REMARC, where a global instruction pointer is broadcast and a processing element then rebroadcasts the instruction in its local configuration store to its row or column. An alternative approach is found in MorphoSys, where the centralized controller broadcasts instructions to each row or column.

Another approach to minimizing the amount of configuration needed was illustrated in PipeRench. PipeRench uses a LUT as the core computing unit and supports an 8-bit data

path. Configuring a (16+)-input LUT would require 65+ kilobits per processing element. PipeRench observed that common computing patterns in image and signal processing performed the same operation on all 8 bits of the data word. Therefore a 3-LUT, two input selects and a carry-in were sufficient to encode most of the desired operations for each bit, and 8 3-LUTs could share the same configuration. This optimization reduced the configuration overhead down to 8 bits for the 8× 3-LUTS in each PE at the cost of some flexibility in available operations. Additional hardware, such as a carry and shift chain, was included in the PE to enable standard arithmetic operations.

The RaPiD project observed that the VLIW word to configure a CGRA was still a potential bottleneck and introduced a blend of traditional FPGA static configuration and CGRA dynamic configuration. The insight was that most of a pipelined application dataflow graph's behavior was static and could be configured once per kernel. Dynamic reconfiguration was still necessary to implement part of the DFG, but it was a small percentage of the total configuration. The application's configuration was then partitioned into a static configuration for the entire kernel and a set of dynamic configurations. Reducing the amount of dynamic configuration dramatically reduces the amount of configuration memory needed. A further optimization of RaPiD's was to introduce a sophisticated control path that decoded a compressed dynamic configuration. This reduced the size of the configuration memory further, at the cost of some logic complexity. MATRIX offers a similar blend of static configuration and dynamic reconfiguration. In particular the I/O for each processing element can be either statically scheduled or dynamically controlled by a control domain.

A.1.6 Optimizing Data Movement

Communication within the CGRA fabric is one area that is ripe for optimization. The time required to execute a dataflow graph consists of the time for computation plus the time to communicate within the CGRA fabric. The pattern of communication is defined by the producer-consumer relationship of operations in the DFG, but the time required to move data around within the fabric is a function of the structure of the interconnect and the relative positions of the producer and consumer. The variance in time for executing

an arithmetic computation is fairly small for a given process technology. The cost for moving data around, however, varies much more from CGRA to CGRA. As a result, some architectures focus on minimizing either the amount or the cost of data movement.

One example of optimizing data movement is avoiding matrix transposes for 2-D matrix operations. Both REMARC and MorphoSys make use of a 2-D mesh interconnect, horizontal and vertical buses, and HSIMD and VSIMD execution modes to compute a 2-D (I)DCT in place.

While CGRAs are typically most efficient when executing pipelined loops, most do not required it. In the PipeRench CGRA, pipelined execution is a integral component of the reconfiguration mechanism (Section 2.1.2) and computing model. As a result, PipeRench has optimized its communication mechanism for pipelined execution. In a feed-forward computing pipeline all of the data at each pipeline stage must move forward in the pipeline, or be lost. To accommodate this, PipeRench developed a pass register file to provide a pipelined interconnect between each stage. During each cycle of execution, the contents of the pass register file are passed from one stage to the next and the processing element can overwrite a single entry. This mechanism allows earlier pipeline stages to efficiently communicate with later stages. Similar to PipeRench, pipelined execution is an integral part of the Systolic Ring architecture. The ring interconnect provides feed forward communication between processing elements. To accommodate feedback in a kernel there is a secondary, pipelined interconnect in the reverse direction.

The RaPiD architecture has two features for minimizing the cost of data movement. One feature is a segmented-track interconnect, which resembles a fragmented partial crossbar and is similar to the interconnect in FPGAs. The segmented-track interconnect provides the communication between PEs and spans the entire fabric. Each component within the PE reads and writes to a subset of the tracks. The tracks are divided into subsections and adjacent tracks are offset from each other so that the subsections are staggered. Different sets of tracks are divided so that the subsections are of varying length. Finally some of the subsections are joined by programmable bus connectors that can pass data in one direction or the other, depending upon configuration. Overall, the segmented-track interconnect provides a flexible set of communication patterns without the overhead of a full or partial

crossbar. One consideration raised by the interconnect is fitting within a fixed cycle time. To address this, pipelining registers are added to the bus connectors between long track segments.

Another optimization of RaPiD is the inclusion of a rich set of pipeline retiming registers within each processing element. Retiming [22, 96] is a technique used to balance a pipelined dataflow graph and can be used to ensure that communication fits within the cycle time. By providing a mechanism to balance different paths of a pipelined dataflow graph these registers allow the interconnect to be more heavily pipelined.

A.1.7 Domain Specific Operators

The standard set of primitive operators are the textbook arithmetic, logical, and comparison operators. Examples include addition, subtraction, multiplication, AND, OR, NOT, equality, greater-than, and less-than.

Domain specific operators (DSOs) are a common approach to optimizing the performance of a CGRA for an application domain. Two main flavors of domain specific operators are composite and fundamental operators. A composite operator is one that combines multiple primitive operators into a single operator, the most common example being a multiply-accumulate (MAC). A fundamental operator has some intuitive, and typically complex, behavior, such as the Hamming distance in KressArray [27] or find-first-one (FFO)[1], which can be composed from multiple primitive operators, albeit inefficiently.

One of the biggest performance advantages of composite operators is not a reduction in hardware resources, but the use of dedicated communication within the composite operator, which reduces interconnect pressure and improves timing. Incorporating support for fundamental operators not only reduces an application's operator count but can alleviate pressure on the interconnect and reduce critical paths by consolidating tightly coupled operations into a single operator.

Examples of domain specific operators are:

- Euclidean distance function, which computes the absolute value of the difference of

[1]FFO is an operation frequently used in floating point arithmetic for renormalization.

two operands in MorphoSys.

- Extended mathematical operations such as minimum, maximum, average, and "absolute value and add" are common multimedia operations that are used in REMARC. MorphoSys provides extended registers and internal routing to create a multi-cycle multiply-accumulate (MAC). Pleiades also includes MAC units and vector minimum and maximum operations.

- "Arithmetic shift right and add" is used in REMARC to efficiently perform multiplication by a constant. DReAM used a LUT, shifter, and an adder to perform constant multiplication efficiently, when the constant was slowly changing over time.

- Extended logical and bit operations such as "logical shifts with Boolean AND and OR", which are used to provide efficient bit level operations on a coarse-grained datapath as in REMARC. A count-ones operator is used in MorphoSys to implement a bit-correlator. The Element Computing Array [15] offers count-ones and zeros, plus count-leading-ones and zeros.

- A Spreading Data Path (SPD) performs a correlation over complex numbers for N iterations. DReAM used the SPD primarily for QPSK-modulation but it could also be used for despreading, or synchronization.

- Control- and configuration-path logic such as reduction operators and a NOR plane enable MATRIX to convert word-wide data to single bit data. The NOR plane can be used to implement random logic or one half of a programmable logic array (PLA).

- Add-compare-select (ACS) is a fundamental operation from Viterbi decoding that is used in the Pleiades templates.

A.1.8 Integration with sequential processor

If an application is composed of computationally intensive kernels that are surrounded by a non-trivial amount control dominated code, it can be difficult to execute efficiently on a

CGRA. For this type of application, it is common to partition execution of the application between the CGRA and an external sequential processor, sometimes referred to as a host processor. In this case, any acceleration achieved by the CGRA is diminished by the overhead of interfacing with the host processor. Minimizing this overhead leads to a number of architectural features presented here.

A local, or tightly-coupled, coprocessor is on-chip and directly attached to the host processor, receiving data directly from the host processor's register file or caches. This model is analogous to floating point coprocessors in older sequential processor designs. The main benefit is that it reduces the CGRA-to-host overhead by avoiding communication through the processor's memory or I/O hierarchy. An example of this level of integration is seen in REMARC, which provides extensions to the MIPS ISA. An additional feature of REMARC is a load (store) aligner that can unpack (pack) a 64 bit value into byte values per column of the array. The load (store) aligner is used to transport values from the host processor's register file into the CGRA fabric.

MorphoSys is another tightly-coupled coprocessor; it provides extensions for the TinyRISC sequential ISA. Furthermore, MorphoSys includes a frame buffer for double buffering data that is fed into and out of the CGRA fabric.

Pleiades provides local coupling to a host processor by integrating the host directly into the "generalized"-mesh of the CGRA. This approach is similar to a system-on-chip (SoC) on-chip network, without the routing overhead of a general network.

ADRES takes the level of integration of a local coprocessor one step further by sharing the register file between the host processor and fabric. In addition to reducing the overhead of communication, this integration also provides a shared memory space as detailed in Section A.1.4.

A.1.9 Preserving Arithmetic Precision

Fixed point arithmetic is prevalent in CGRAs due to the expense of floating point arithmetic units. Combining fixed point arithmetic with a fixed bit-width datapath can lead to either data overflow or a loss of precision. A common example is with multiplication, where the

width of the result is equal to the sum of the widths of the operands. Most CGRAs choose to restrict the output to be the same width as the input. This approach is the simplest, requires the least resources, and if the operands are small, then overflow doesn't occur.

For applications with a large dynamic data range, this approach is insufficient. There are two common approaches for solving this problem. The first approach is to preserve all of the bits within the processing element, but strip excess bits when transmitting the result out of the processing element. This approach requires more resources within the processing element, doesn't impact the interconnect, and works well for a sequence of operations within a single processing element. The second approach, which is generally more applicable, is to output the result as two datapath words, which then can be transmitted out of the processing element either serially or in parallel. MorphoSys is one example that uses the first approach to support feedback for the ALUs. Internal to the processing element, the ALU result registers are extended width to avoid loss of precision during a multiply-accumulate operation. RaPiD utilizes the second approach, with parallel outputs for both results.

A.1.10 Flexibility in Scheduling & Execution

Compiling an application kernel to a CGRA requires mapping operations to operators in space and scheduling of operations, with respect to communication latencies, in time. Mapping in space is the traditional CAD place and route problem from logic design and is beyond the scope of this report. Scheduling operations in time can be done statically or dynamically. In a static schedule all operations are assigned a phase during which they execute and if, for some reason, an operand is not ready, then the entire application has to stall or garbage is computed. In a dynamic schedule, all operations are assigned a phase to execute in, but if an operand is not available then that operator and all dependent operators can stall, independent of the rest of the application. Static scheduling minimizes the amount of hardware needed for runtime synchronization and makes it easier to understand and debug the fabric's execution, but performs poorly on applications with dynamic control flow or unpredictable communication. Because many interesting applications have very regular control flow and communication, most CGRAs are statically scheduled.

Dynamic scheduling improves performance on applications with unpredictable behavior or communication, but requires some form of coordination and flow control to avoid resource conflicts (structural hazards). Dynamic scheduling is typically supported through some form of handshaking mechanism. Common examples include buffering with presence bits (or valid bits) for data words, or handshaking protocols, such as taking data / data valid (TD/DV), that signal the transfer of valid data. PADDI-2 uses a hardware handshake on communication that can stall all participants when one is not ready. Another need for dynamic scheduling is for support of variable length operators. Both DReAM and Pleiades use data tokens and handshaking protocols to support variable length multiplications and arithmetic operations, respectively. Ambric is an MPPA that uses a hardware handshake on all communication registers to synchronize execution.

Another approach that extends the flexibility of a statically scheduled architecture is the use of predication, as shown in ADRES. Support for predication enables a compiler to reduce the dynamic control flow of a application kernel through techniques such as if-conversion. Predicated operations are necessary to prevent erroneous execution of stateful (or state-updating) operations. Predication can also be used to minimize dynamic power by avoiding unnecessary execution of functional (or pure) operators.

A.1.11 Choreographing External I/O

Given the high number of arithmetic units within a CGRA and the relatively limited amount of bandwidth into and out of the fabric, keeping the processing elements busy is a daunting task. One fundamental limitation of CGRAs is that data reuse in the application must be high to achieve high levels of resource utilization within the fabric [12], *i.e.* the computation to communication ratio must be high. While the application's computation to communication ratio is beyond the control of the CGRA, most CGRAs leverage a streaming interface or DMA engine that attaches to the periphery of the fabric. RaPiD is one of very few projects to explicitly address the demands on the external I/O from the fabric. At a high level, streaming I/O in RaPiD is similar to most CGRAs: a set of address generators iterate through some access pattern and push (or pull) data from memory into (or out of) FIFOs

that are connected to the fabric. The challenge for RaPiD is that the multiple control domains (Section A.1.3) provided are intended for execution of nested and sequenced loops rather than independent threads of computation. To avoid stalling the fabric, it is important for the address generators to follow the execution patterns of the control domain, and so in RaPiD a modified version of the configuration controller is used for address generation.

Another optimization is to minimize the latency of I/O as shown in MorphoSys, which introduced a double buffered frame buffer. The frame buffer has a very wide interface directly to the fabric and is accessed every cycle. Its double buffering means that the host processor and off-chip memory can fill the buffer while computation is occurring, thus overlapping computation and communication.

A.1.12 Reducing Dynamic Power Consumption

While CGRAs are often viewed as being power efficient when compared to sequential processors, one drawback to spatial processing is that the interconnect consumes a large percentage of the dynamic power, as shown by Poon et al. [21]. One approach to reducing the interconnect power is presented in Pleiades [39]. In the Pleiades communication network, each arc of the applications dataflow graph is mapped to a dedicated channel, which preserves any temporal correlation within a data stream and minimizes the amount of switching activity.

Another significant component contributing to dynamic power is the global clock found in synchronous systems, which can be as high as 28% to 40% of the system's energy, as noted by [39, 21]. Asynchronous systems provide the hope of reducing the overall clock power, but are challenging to design, debug and add overhead in the asynchronous handshaking logic. A hybrid approach to reducing the clock overhead is to use a globally asynchronous, locally synchronous (GALS) system. GALS systems replace the bulk of the large synchronous global clock tree with asynchronous logic, but preserve the simplicity and efficiency of a synchronous system within local structures. Pleiades is an example of a GALS system, where each processing element's local clock generator is driven by the arrival of an asynchronous data token. AsAP is an MPPA that uses GALS to improve energy efficiency. In the AsAP implementation, to reduce dynamic power, the clock generator in each processor is able to

drain the processor pipeline and shut down when either the input or output FIFOs stall.

A.1.13 Runtime Binding and Allocation of Resources

Traditionally the mapping from a dataflow graph to the fabric resources (or datapath graph) is done at compile time and expects access to the entire fabric. Ideally it would be possible to simultaneously map multiple kernels to a fabric, but contemporary tool flows requires that all kernels be known a priori and mapped simultaneously, which is impractical in practice. One approach to overcoming this limitation is to rebind a configuration to the available fabric resources at runtime. The Systolic Ring DHM provides such a runtime binding system. In order to make this rebinding lightweight and efficient the Systolic Ring is divided into topologically isomorphic regions. Computation is contained within a set of isomorphic regions and communication that crosses the boundary of the region must be encoded in a position independent format, *e.g.* cardinal directions (N,S,E,W). Application kernels are then mapped onto unused portions of the fabric by performing simple transformations on the configurations as they are loaded into the fabric. In the Systolic ring, these transformations are rotations on the external I/O. Another use of runtime binding is to perform dynamic loop unrolling, mapping multiple instances of the same loop onto the fabric as resources allow.

The PipeRench project is another CGRA that has topologically isomorphic regions. In PipeRench the region is a pipeline stage and this property was used to enable pipeline reconfiguration and hardware virtualization for the platform. Derivative work on a Queue Machines [97], HASTE [98], and Tartan RF [24] projects leveraged the isomorphic regions to perform a similar runtime binding as the Systolic Ring DHM.

A.2 Design Principles & MPPA Architectures

The exploration of spatial architectures continues here with an examination of some massively parallel processing arrays. The details of each architecture are presented in Table A.4. The common components of MPPAs are very similar to those presented in Section 2.1 for CGRAs. In Table A.4 the number of processors is equal to the number of control domains. For each architecture I have tried to identify key architectural features that fit

within the design principles for CGRAs, and marked them in Table A.5. A summary of each architecture is presented below, with references to the CGRA design principles when they encompass the MPPA architectural features.

PADDI-2 Programmable Arithmetic Devices for high speed DIgital signal processing [99, 100] is an early example of a MPPA. As described in Section 2.2, PADDI-2 has a control domain per processing element, which enables a MIMD programming model rather than the VLIW or SIMD of CGRAs. PADDI-1 [101], from which PADDI-2 is derived is a CGRA and provided VLIW style execution. Topologically, processing elements are partitioned into clusters, with a shared bus for communication. Clusters are interconnected via a crossbar.

AsAP Asynchronous Array of Simple Processors [102] is designed to efficiently compute many large DSP tasks through fine-grained task-level parallelism. Each processor in the array supports a multiply-accumulate operation (Section A.1.7). The communication topology is a hybrid static-dynamic mix, where inputs are statically configured and outputs are dynamically configured, which helps reduce the amount of configuration data (Section A.1.5).

Ambric Ambric [13] is designed to implement a Structural Object Programming model that is very similar to Communicating Sequential Processes. To provide this model, each processor executes a single object; all communication is through a self-synchronizing FIFO channel. The grid interconnect is statically configured, which helps reduce configuration data (Section A.1.5), and two processors and two embedded RAMs are grouped into a cluster that has a dynamic crossbar interconnect. Processors are partitioned into "control" and "DSP" processors, where every DSP processor in the array supports a multiply-accumulate operation (Section A.1.7) and has more local memory. One distinct feature is that processors can execute instructions from local RAM, or directly from a channel. To minimize the configuration data (Section A.1.5), Ambric supports a special SIMD mode, where instructions are replicated through a tree of channels. Using a 32-bit datapath in a multimedia application runs the risk of under utilizing resources if the application has sub-word operands. To improve efficiency for sub-word execution, the datapath was made to scale (Section A.1.1) to half-word and quad-word operations, which are available on the DSP processors.

Architecture	Topology	Word-width (bits)	Interconnect	Control Domains	Compute Model
PADDI-2	hierarchical	16	1st lvl bus, 2nd lvl crossbar	48-64	MIMD
AsAP	2-D array	32	Mesh	36	MIMD
Ambric	2-D array	32	Mesh, Grid	336	MIMD

Table A.4: MPPA Architectural Details

Arch. Features	Section	PADDI-2	AsAP	Ambric
Scalable Datapath	§A.1.1	√		√
Minimizing Cfg. Overhead	§A.1.5		√	√
Domain specific operators	§A.1.7		√	√
Flexibility in Sched. & Exec.	§A.1.10	√		√
Reducing Dynamic Power	§A.1.12		√	

Table A.5: MPPAs and list of features by section number