

© Copyright 2011
Stephen A. Friedman

Resource Sharing in Modulo-Scheduled Reconfigurable Architectures

Stephen A. Friedman

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

Doctor of Philosophy

University of Washington

2011

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

Stephen A. Friedman

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

Lawrence Snyder

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

Resource Sharing in Modulo-Scheduled Reconfigurable Architectures

Stephen A. Friedman

Co-Chairs of the Supervisory Committee:

Professor William H.c. Ebeling
Computer Science & Engineering

Professor Scott Hauck
Electrical Engineering

This dissertation explores compiler algorithms for sharing resources in coarse-grained reconfigurable arrays (CGRAs). CGRAs are scalable, word-oriented architectures designed for executing high-performance computation kernels. Instead of having a single configuration like an FPGA or a program counter indexing arbitrary instruction words, a CGRA maintains several configurations on chip that are sequenced through by a modulo-counter, changing the configuration each cycle. This model of execution works well on pipelined compute-intensive loops with a large amount of instruction level parallelism and limited control flow. CGRAs provide high-efficiency, high-throughput computing, achieving an order of magnitude improvement in operations per cycle over conventional CPUs.

Even with their considerable strengths, CGRAs' flexibility and efficiency can still be improved through compiler-based resource sharing. In this dissertation I propose and demonstrate the following novel sharing techniques applicable to this execution model:

- *Sharing Static Routing* – Reducing the number of bits needed per configuration can save area and power, or be used to allow for more configurations, increasing flexibility. One way of reducing the number of control bits is to limit portions of the interconnect to a single, repeated configuration while the rest of the array is free to use multiple configurations. Towards this goal, I propose an extension to

the PathFinder/QuickRoute routing algorithms for supporting sharing of statically configured pipelined routing resources in a time-multiplexed system.

- *Predicate Aware Sharing of Compute and Routing Resources* – The basic modulo-scheduled execution model can efficiently pipeline and execute a simple loop. CGRAs often support complex control flow by reserving resources to perform all computations, and then ignoring the results of the untraversed control paths. To reduce this overhead, I propose a scalable hardware modification, hardware abstractions, and a set of Schedule/Place/Route algorithms capable of predicate-aware mapping. This system allows sharing of resources across operations executed under mutually-exclusive control flow – for example, reusing resources across then and else branches of an if construct. It achieves this sharing by exploiting otherwise wasted configuration memory.

These sharing techniques provide more efficient use of CGRA resources. Sharing static routing helps reduce the large configurations needed for CGRAs. Mutual-exclusive sharing reduces the burden of control flow, which can broaden the set of applications CGRAs can accelerate, and provide some flexibility to the programmer. It allows the programmer to handle infrequent or exceptional cases directly on the accelerator without forcing portions of the accelerator to remain idle waiting for those cases. These algorithms are implemented and evaluated across a suite of benchmarks to demonstrate the benefits of sharing in CGRAs.

TABLE OF CONTENTS

	Page
List of Figures	v
List of Tables	viii
List of Algorithms	ix
Glossary	x
Chapter 1: Introduction	1
1.1 Flexibility in Computation	1
1.2 The Space Between ASICs and CPUs	3
1.2.1 FPGA	4
1.2.2 VLIW	4
1.2.3 Modulo-Scheduled CGRA	5
Chapter 2: The Mosaic Project	8
2.1 Mosaic Tool-chain	8
2.2 Macah	8
2.3 Architecture Generation	11
2.4 Mapping Applications to CGRAs using SPR	12
2.4.1 Related Work	12
2.5 Abstract Representation	15
2.5.1 Scheduling	16
2.5.2 Placement	20
2.5.3 Routing	21
2.5.4 Resource-Performance Tradeoff	21
2.6 Simulation	22
Chapter 3: Practical SPR	24
3.1 Latency Padding	25

3.2	Latency Padding Effects	27
3.3	Dynamic Recurrence Clustering	29
3.4	Dynamic Recurrence Clustering Effects	30
3.5	Conclusion	30
Chapter 4:	Static Interconnect Sharing in SPR	32
4.1	Static and Dynamic Routing	33
4.1.1	Control-based PathFinder	34
4.2	Evaluation	39
4.2.1	Architecture	40
4.2.2	Benchmarks	40
4.2.3	Static Interconnect Sharing	41
4.3	Conclusion	44
Chapter 5:	Predicate Aware Resource Sharing in SPR	46
5.1	Predicate Aware Sharing	47
5.2	Representing Predicate Relationships - The CDT	49
5.2.1	CDT Structure	50
5.2.2	Altering the CDT	53
5.2.3	Tree Restrictions	55
5.2.4	Integrating CDT and Modulo Scheduled Pipelining	59
5.2.5	CDT Pruning and Verification	66
5.3	Representing Predicates	68
5.4	Hardware Support for Mutual Exclusion	71
5.4.1	Regions of Control	72
5.4.2	Modifying Configuration Retrieval	73
5.4.3	Exposing Configuration Control	75
5.5	Supported Abstractions	76
5.5.1	Control Dependence Tree	77
5.5.2	Aged and Aggregate Conditions	77
5.5.3	Regions	77
5.5.4	Predicate Gateway	78
5.6	Predicate Aware SPR Overview	78
Chapter 6:	Predicate Aware Scheduling in CGRAs	80
6.1	Predicate Aware VLIW Scheduling	81

6.2	Applying Predicate Aware Scheduling to CGRAs	83
6.2.1	Computing the Resource Minimum II	83
6.2.2	Alternative to PQS	84
6.2.3	Promotion for Dependence Trimming	85
6.2.4	The Region Abstraction and Scheduling	86
Chapter 7:	Predicate Aware Placement	87
7.1	The Problem	87
7.2	Local Sharing	89
7.3	Region Costs	92
7.4	Finalizing the Placement	95
Chapter 8:	Predicate Aware Routing	97
8.1	The Problem	99
8.2	Predicate-Aware PathFinder Costs	104
8.2.1	Predicate-Aware Signal Congestion	105
8.2.2	Predicate-Aware Control Congestion	106
8.2.3	Negotiated Routing Costs	120
8.2.4	Simple Optimizations	121
8.3	Routing Predicates	122
8.3.1	When to Request Predicates	122
8.3.2	Finding Needed Predicates	124
8.3.3	Selecting Predicates with Capacity Limitations	126
8.4	Predicate-Aware Signal Tree Routing	127
8.4.1	QuickRoute	127
8.4.2	Predicate Aware Multi-source/-sink Routing	129
8.4.3	Predicate Aware Multi-condition/-source/-sink Routing	133
8.4.4	Predicate Aware Route Re-use	141
8.4.5	Choosing the Routing Condition	143
Chapter 9:	Evaluation of Predicate Aware Mapping	146
9.1	Evaluation Architecture	146
9.1.1	Register File Limitations	148
9.1.2	Simplified Processing Elements	150
9.1.3	Intra-connect Capacity	151
9.1.4	Test Architecture Configurations	152

9.2	Evaluation Benchmarks	153
9.3	Scheduling Speculation versus Sharing	155
9.3.1	The Potential of PA-SPR	156
9.3.2	Balancing Sharing and Speculation	159
9.4	Performance Improvement in PA-SPR	162
9.5	Sharing in PA-SPR	166
Chapter 10:	Conclusions and Future Work	169
10.1	Summary	169
10.2	Conclusions	172
10.3	Future Work	176
10.3.1	Reducing Predicate Pressure	176
10.3.2	Alternate Hardware Implementations	177
10.3.3	Optimizing Predicate Usage	177
10.3.4	Merging Select Operations Into Routing	179
10.3.5	Power Implications of Predicate-Aware Sharing	180
10.3.6	Cross-iteration Mutual Exclusion	180
10.3.7	The Counter is Dead – Long Live the Counter!	181
10.4	Epilogue	181
	Bibliography	183

LIST OF FIGURES

Figure Number	Page
2.1 Mosaic project tool-chain.	9
2.2 A 2x1 cluster CGRA block diagram.	11
2.3 Example of a simple dataflow graph.	14
2.4 Unrolled datapath graph with mapped dataflow graph.	19
2.5 Scaling across architecture sizes.	22
3.1 II and Latency effects across different padding settings.	28
3.2 Effects of Clustering on II.	30
4.1 Dynamic and static mux representation.	33
4.2 Routes across different phases of a mux.	34
4.3 Congestion calculation from signal usage.	36
4.4 History updates for static control congestion.	38
4.5 Histogram of dynamic and static sharing. The channel width for these tests was set at the minimum routable channel width for the static sharing algorithm.	42
4.6 Channel width across the benchmarks for hardware, software, and no time multiplexing.	43
4.7 Fig. 7 from [VEWC ⁺ 09]. Area-energy product for 32- and 8-bit datapath word-widths and maximum supported II of 16, 64, and 128 configurations, varying the percentage of interconnect channels that are statically configured or dynamically time-multiplexed.	44
5.1 Examples of Program 5.1 using different conditional execution methods. Note: sum' refers to the new value of sum.	48
5.2 A simple example of a Control Dependence Tree for sequenced and nested IF statements.	51
5.3 Example of unstructured code requiring a full DAG CDG with divergence in a condition node.	56
5.4 Example of unstructured code requiring a full DAG CDG with divergence in a partition node.	57
5.5 Combining time into the Control Dependence Tree	60

5.6	Base CDT and example schedule for a kernel iteration.	61
5.7	Even though red and blue operations can share <i>within</i> an iteration, they cannot share <i>across</i> iterations. Operation 8 may share with 9, but not 16. . .	62
5.8	Calculations showing where sharing can and cannot happen.	63
5.9	Base CDT and example schedule for cross-iteration sharing.	64
5.10	Calculations showing where cross-iteration sharing can happen.	65
5.11	A loop with cross-iteration sharing.	65
5.12	The final fully expanded scheduled CDT along with the steady-state kernel schedule.	67
5.13	Example of pruning partition nodes that do not provide any mutual-exclusion.	68
5.14	Example of a CDT and the conflict relation it embodies.	69
5.15	Illustration of an aggregate execution condition. Two conflict vectors plus their delay are checked for conflicts then combined into an aggregate execution condition.	71
5.16	A basic configuration hardware design.	72
5.17	CGRA colored by configuration regions.	72
5.18	Configuration hardware diagram with potential modification points marked in blue	73
5.19	Combining counter and predicate bits.	75
6.1	Fig. 1 of [SMDL03]: Example code segment	81
6.2	Fig. 4 of [SMDL03]: IMS kernel (a) versus PAMS kernel (b) schedule	81
6.3	Fig. 8 of [SMDL03]: Predicate-aware reservation table	82
8.1	Joining and tunneling for mutually exclusive routes.	103
8.2	Code and Euler diagram of the space of possible conditions.	105
8.3	Revisiting the control congestion from Chapter 4 with the context of the CDT	108
8.4	An example CDT and routing using texture and color.	109
8.5	Predicate-aware control congestion with only the phase available.	110
8.6	Predicate-aware control congestion without the phase signals available. . .	111
8.7	Predicate-aware control congestion with tables for partial promotion with limited predicate availability.	112
8.8	Configuration slot expansion to configuration words.	114
8.9	Child slot override when there are partial promotion conflicts.	119
8.10	Comparison of original and predicate-aware versions of QuickRoute.	128
8.11	Compatible signal tunneling.	130
8.12	Single signal divergence and re-convergence with control congestion.	131

8.13	Example conflict resolution for routes.	132
8.14	Possible condition relationships between fanouts with the same source device.	135
8.15	Illustration of broken and valid route re-use.	136
8.16	Illustration of re-used versus independently routed mutually exclusive source fanouts.	140
9.1	Diagram illustrating conditional routing through a register file that cannot be fixed through post processing.	149
9.2	Diagram illustrating a simple retiming chain.	150
9.3	Complex and simplified Mosaic architecture processing element designs. .	151
9.4	Plot of the minimum II estimate across the tests for both the full PA-SPR with dependence trimming enabled and the standard SPR, with the potential advantage of PA-SPR highlighted in the middle plot.	157
9.5	Plot of the II advantage of PA-SPR after the scheduling has been completed.	159
9.6	Comparison of the schedule II to the minimum II for both predicate aware and regular scheduling.	160
9.7	Comparison of the scheduled II between PA-SPR with dependence trimming disabled and SPR.	161
9.8	Comparison of PA-SPR with dependence trimming to choosing the best of either PA-SPR with no trimming or SPR.	162
9.9	Plots showing the final advantage in II of PA-SPR over SPR.	163
9.10	Comparison of the increase in II during the placement and routing process between PA-SPR and SPR.	165
9.11	Comparison of the total increase in II during mapping in PA-SPR and SPR.	165
9.12	Plot of the sharing of compute elements in PA-SPR.	166
9.13	Plot of the sharing of routing muxes in the interconnect.	167

LIST OF TABLES

Table Number	Page
2.1 Example schedule with II 2 and length 4.	17
2.2 Example schedule with II 1 and length 4.	18
3.1 Summary of Benchmarks	25
8.1 Table of possible condition relationships for fanouts from the same source.	137
8.2 Table of condition relationships that support sink- and source-routed re-use.	139
8.3 Table of condition relationships that support only source-routed re-use. . .	139
9.1 Summary of Benchmarks	155

LIST OF ALGORITHMS

Algorithm Number	Page
2.1 Main SPR Control Loop	15
3.1 Main SPR Control Loop with Padding	26
4.1 Congested Mux History Update	37
5.1 Sum of absolute differences example.	47
8.1 Predicate Aware PathFinder	125
8.2 PathFinder (Negotiated Congestion)/QuickRoute	130
8.3 Predicate Aware PathFinder/QuickRoute integration	142

GLOSSARY

aggregate execution condition

A representation for the union of execution conditions designed for efficient conflict testing with other conditions .

compatible

Two operations that are mutually exclusive are said to be compatible because they may share architectural resources without the possibility of run-time conflict..

condition node

A node in the CDT that represents the condition under which a control-flow block will execute..

configuration region

The region of a reconfigurable architecture controlled by a single configuration memory.

configuration slot

A representation for a predicate that is available in a region and the configuration words it expands to in the region.

partial promotion

Speculative execution of an operation under a run-time condition that is true in a super-set of the executions where the original operation's condition was true, but not necessarily all executions.

partition node

A node in the CDT that represents a control construct that partitions control flow into mutually exclusive blocks..

predicate gateway

Targetable sink for routing predicates used by predicate aware routing.

recII

recurrence minimum II, a lower bound the schedulable II set by the largest recurrence loop.

resII

resource minimum II, a lower bound set by resource limits on the schedulable II.

Chapter 1

INTRODUCTION

This dissertation explores resource sharing in modulo-scheduled reconfigurable architectures. This introductory chapter will explain what a modulo-scheduled reconfigurable architecture is and present a view of the current computing landscape, which explains when and why resource sharing in such an architecture is beneficial. Starting at the broadest sense of digital computing devices, it then narrows down to the specific application space to which this work applies. The path taken from broad to narrow will help motivate this work because the trajectory taken will help highlight lightly explored areas of digital computation relative to more main-stream solutions.

1.1 Flexibility in Computation

In the broadest sense, a digital computation is accomplished by having a digital device carry out a set of manipulations on a set of inputs to produce some outputs. Instantiating a computation in a digital device is a task that relies on balancing conflicting costs. Though there may be many different costs, some of the most important are time to design the system, monetary cost to design the system, cost of an instance of the system, time of computation, silicon area, and power per computation.

Some of these costs are incurred only once or a few times relative to the number of times the device executes a computation, and so can be amortized when considering a per-computation cost. One way of amortizing these costs is to create a device for a single application that will be used repeatedly over the life of the device. An example of this is transforming data streams to be sent over radio protocols, as in a cellular telephone. Another way of amortizing these costs is to support a broad range of applications on the same device. An example of this is the processor in a desktop computer, which can rapidly switch between entertainment, finance, scientific simulation, communication, and

a myriad of other applications at the user's request. I will illustrate this range of flexibility by discussing two distant points in the range – Application Specific Integrated Circuits and general-purpose processors.

Application Specific Integrated Circuits (ASICs) represent one end of this flexibility range. Computations that can amortize the design, fabrication and dedicated hardware costs can be directly embedded into hardware that does nothing but carry out that computation as efficiently as possible. These applications require the least flexibility, and as a result a great amount of design cost can be used to produce something that has low per-computation cost – it is fast, low power, and can use a minimal amount of resources to directly execute the computation at hand.

At the other end of this flexibility spectrum are Von Neumann style general-purpose processors (CPUs). This type of device is highly flexible – it can compute any possible computation up to the system's intrinsic resource limits. This flexibility is obtained through reconfigurability, time multiplexing, and an extra source of input – the instruction stream. A very abstract way of understanding a processor is to view it as a hardware device that performs a single word-sized manipulation at a time, which is configured by a special instruction input. A CPU performs a large computation by time-multiplexing the processor across all of the manipulations for a given computation, and storing the intermediate results in a memory system. The instruction stream defines the sequencing for this time-multiplexed reconfiguration.

However, in a CPU this flexibility comes at a cost. The direct hardware implementation of a computation carries out independent manipulations at the same time, whereas a general-purpose processor must operate according to the serialization constraints of the instruction stream, leading to an increase in computation time. Advanced processors are able to eliminate some of the latency of this serialization in local windows of the instruction stream, but beyond the granularity of that window, they still serialize according to the instruction stream. Additionally, individual operations may take longer and consume more space – for example adding 4 to a 32-bit input with a general purpose 32-bit adder in a processor will take longer and use more hardware than dedicated increment by 4 hardware. Also, storing intermediate results in a memory system has space and

time overheads compared to sending an output value down wires (and possibly through pipeline registers) to its consumer in an ASIC. Even the word width imposes overhead. For example, if the width of the computations is 4-bit, then there is a hardware and power overhead in a 32-bit CPU for computing over the extra 28 bits.

General-purpose processors also increase flexibility in an incredibly powerful way, but perhaps one that is taken for granted – the ability of the input to alter the instruction stream and intermediate storage access. Viewing a processor plus instructions as a way to emulate a hardware instance of a computation, being able to manipulate the input stream is like performing hardware optimizations on the computation at run-time. By jumping around a section of code that generates unneeded results, the emulation is being pruned, as if the hardware was dynamically removed. Additionally, looping can be seen as dynamically duplicating hardware, one emulated copy per loop iteration. Altering accesses to intermediate storage (indirect memory addressing) is essentially dynamically changing the communication patterns of the emulated hardware. This can change the entire computation graph. Most programmers probably do not think of unbounded hardware when writing a while loop, or dynamic re-wiring when accessing an array, but that is the power of these constructs – they convert the wildly differing costs of hardware to directly execute these computations into a fixed hardware cost for the processor plus a wildly differing execution time per computation.

1.2 The Space Between ASICs and CPUs

The design space between ASICs and CPUs in terms of this broadly defined flexibility is quite large. It is also not a linear space, but multi-dimensional, where the different types of costs can be traded off against one another, for instance trading amortizable costs for per-computation costs based on the characteristics of the computations covered. This section will briefly touch on several architectures that represent different points in this space and their relations to one another. This is done merely to provide context, as these broad caricatures of architectures are median values in the design space, and there are

examples of real-world projects that make trade-offs which blur the boundaries between these architectures. This list is also not meant to be exhaustive, but illustrative.

1.2.1 FPGA

Field Programmable Gate Arrays (FPGAs) are often thought of as a sea of gates. They have look-up tables (LUTs) for implementing arbitrary Boolean functions, registers for storing intermediate results, and configurable wiring to connect them all together. The equivalent to instructions for an FPGA is a bit-stream, which is all of the bits needed to fill in the LUTs and set up the configurable wiring. For a particular computation, a circuit design process similar to that of an ASIC is used to generate a bit-stream, this bit-stream is loaded, and then the computation is run. Using an FPGA implementation is one of the most literal programmable ways to emulate an ASIC, and so requires a proportional per-operation computation cost, inflated by the overhead of reconfigurability. In general, FPGAs do not do time multiplexing except at the level of re-configuring between computations. Possible reasons for this are that the bit-stream (instruction) is so large that either the bandwidth needed to read them at a practical rate for fine-grained multiplexing is impractical or the storage overhead on-chip makes it impractical to store multiple configurations at one time. Exceptions to this do exist, but they are rare. FPGAs amortize the fabrication level design times and tooling costs across different computations, but retain the circuit level design costs and limitations of bounded hardware size.

1.2.2 VLIW

Very Long Instruction Word architectures (VLIWs) are a step from CPUs towards more parallel computations. Unlike a single-issue CPU that has one reconfigurable execution unit, VLIWs combine a small number of execution units that can operate in parallel. The instruction bandwidth is increased relative to a regular CPU, but if the bandwidth can be supported, this can lead to faster per-computation execution time at the cost of some more hardware and more complicated instruction generation. Most VLIWs provide uniform access to intermediate storage across all execution units through a large crossbar, which can become a scaling bottleneck. Thus, going past a small number of execution

units introduces hardware costs and cycle times from the quadratic scaling of a crossbar renders these wide-issue machines impractical.

1.2.3 Modulo-Scheduled CGRA

Coarse-Grained Reconfigurable Arrays or Architectures (CGRAs) represent a point in the flexibility space that is between FPGAs and VLIWs. This dissertation explores compiler techniques for making computation on CGRAs more efficient and flexible. Like a VLIW or CPU, a CGRA is built at a word-level, hence they are coarse-grained relative to the bit level of FPGAs. They are intended to have many more execution units than a traditional VLIW – where an FPGA is a sea of gates, a CGRA is a sea of execution units. They are usually tiled in some form, and so they have a scalable configurable interconnect more like an FPGA, and less like the all-to-all crossbar configuration of a VLIW. This large amount of parallelism and scalable interconnect architecture means that CGRAs are most appropriate for running computations similar to those placed on an FPGA. Relative to an FPGA, a CGRA is more appropriate for computations where the natural bit width is coarse enough that the bit-level configuration overhead of wide computations in FPGAs outweighs the overhead of using wide units on narrow computations in CGRAs.

Consider devoting equivalent die area to an FPGA or a CGRA. To route a word of data through the interconnect in an FPGA, each bit of that path will be individually configured along the entire path. In a CGRA, all the bits of the word will be routed together, so a single configuration can be used across all of the bit-lines at every configurable junction in the path. In contrast to this, an FPGA requires a separate configuration for each bit-line. This provides a word-sized reduction in the amount of configuration memory needed to route a word of data when moving from the fine-grained FPGA configuration to the CGRA. Similarly, there will be a large reduction when switching from a word-sized set of LUTs to a single hard-wired execution unit. Relative to a VLIW, a CGRA instruction/configuration can still be hundreds of times bigger, due to the large number of execution units and the explicit nature of interconnect control. This means that the instruction bandwidth that a CGRA can support is likely not large enough to fetch an arbitrary instruction every

cycle like a CPU, and instead must operate more like an FPGA, where the configuration is loaded at the beginning and re-used throughout the life of the computation.

Many CGRAs use their relative space advantage over FPGAs to incorporate multiple configurations on chip at a time. This can be seen as having a few instructions to loop through in the execution. The baseline for CGRAs used in this dissertation uses the simple method of directly looping through these configurations, one per cycle, during execution. This style of execution is common for inner loops in VLIW processors, and research into these processors developed techniques such as software pipelining and modulo scheduling to extract appropriate parallelism from programs written in high-level languages. The use of high-level languages can lower the development time of applications relative to the traditional circuit-level design of FPGAs [XSAH10].

This dissertation focuses on Modulo-Scheduled CGRAs. My work is part of the Mosaic project to explore this space through coupled high level language design, compiler back-end design, and architecture exploration, as described in Chapter 2. In this ecosystem, I want to make the best use of the on-chip instruction space. In this dissertation, I propose and analyze compiler back-end techniques for extending the flexibility of this on-chip instruction space. This includes both reducing the amount of on-chip instruction space needed and supporting flexible resource sharing to exploit the available space that may otherwise be wasted. In particular, this dissertation makes the following contributions, evaluating them in the context of SPR, the compiler back-end of the Mosaic tool-chain:

- *Practical VLIW/FPGA Algorithm Adaptation* – SPR is composed of scheduling adapted from VLIW compilers and placement and routing adapted from FPGA tools. Practical coupling of the algorithms is achieved through a novel latency padding technique, providing feedback between placement/routing and scheduling that yields throughput improvements in the final mappings. In addition, a dynamic clustering algorithm is proposed for clustering critical loops. This clustering improves throughput by adapting FPGA placement to the limitations of fixed-frequency, highly pipelined interconnect. These additions are described and evaluated in Chapter 3.

- *Sharing Static Routing* – Reducing the number of bits needed per configuration can save area and power, or be used to allow for more configurations to increase flexibility. One way of reducing the number of control bits needed is to provide only one configuration for a subset of the architecture that will be re-used across the time-multiplexed configurations of the rest. Towards this goal, I propose an extension to the PathFinder/QuickRoute routing algorithms for supporting sharing of statically configured pipelined routing resources in a time-multiplexed system. This extension is described and evaluated in Chapter 4.
- *Predicate Aware Sharing of Compute and Routing Resources* – The basic modulo-scheduled execution model can efficiently pipeline and execute a simple loop. More complicated control flow is often accomplished by reserving resources to perform all computations, and then ignoring the results of the untraversed control paths. To reduce the resource cost of complicated control flow, and thereby increase flexibility, I propose a Schedule/Place/Route system capable of predicate-aware mapping that allows sharing of resources across operations executed under mutually-exclusive predicates. An overview of the abstractions needed to support predicate-aware mapping is given in Chapter 5. In Chapter 6, I describe the adaptation of VLIW predicate-aware scheduling to CGRAs. In addition, I explore the trade-offs that occur when sharing leads to longer loop-carried dependencies, which can reduce throughput instead of improving it. In Chapter 7 I describe the changes needed to support predicate aware sharing in Simulated Annealing based placement. In Chapter 8, I describe a method for adapting both the PathFinder and QuickRoute algorithms to deal with the added complexity of predicate aware sharing. The modification to PathFinder is particularly interesting because it comes from a generalization of the static route sharing described in Chapter 4. The combined system is evaluated in Chapter 9.

Finally, Chapter 10 provides a summary of conclusions and future directions.

Chapter 2

THE MOSAIC PROJECT

The implementation and evaluation work in this dissertation was carried out as part of a larger research project at the University of Washington. This chapter provides an overview of the Mosaic project, along with introductory coverage of the different elements of the toolchain.¹

2.1 Mosaic Tool-chain

The Mosaic project is an exploration of architectures and programming tools with the goal of quantifying the architectural trade-offs and necessary innovations in tool support for CGRAs. The project consists of three parts: a new system-level language – Macah; an architecture-adaptive back-end mapping tool – SPR; and an architecture generation tool and characterization effort. Figure 2.1 shows a block diagram of the Mosaic project tools. The final goal is to produce a high-performance, low-power device design and a set of compiler tools that will ease the programming burden.

2.2 Macah

The front of the tool-chain is a set of benchmarks and a front-end compiler for a language called Macah that was developed as part of the dissertation work of Benjamin Ylvisaker [Ylv10]. Macah is a C-like language, borrowing most of its syntax from C. There are three significant differences between the Macah compiler in Mosaic and traditional C compilers that are important for the work in this dissertation: support for an explicitly marked kernel for acceleration, streaming I/O with relaxed re-ordering semantics, and automated flattening of nested control flow within a kernel.

¹The portions excerpted (2.1, 2.4, 3.1-3.4, 4.1-4.2.3) are based on an earlier work: SPR: An Architecture-Adaptive CGRA Mapping Tool, in Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'09) ©ACM, 2009. <http://doi.acm.org/10.1145/1508128.1508158>

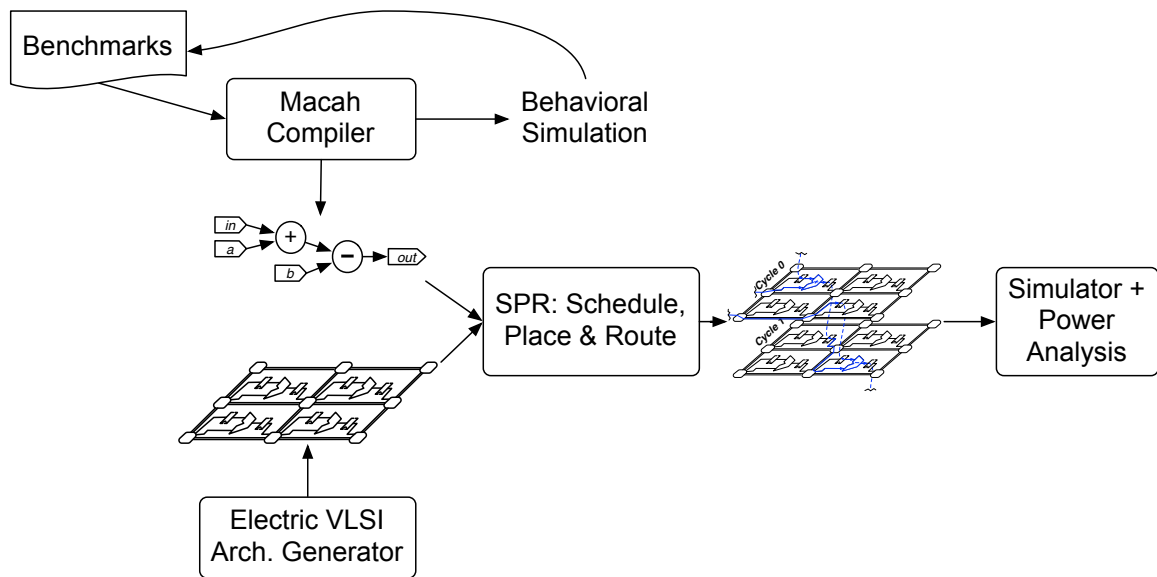


Figure 2.1: Mosaic project tool-chain.

In Macah, specific portions of an application that are intended for special hardware acceleration are denoted as a kernel block, demarked by the `kernel` keyword and enclosed within curly braces. The execution model assumes a general-purpose processor will execute the majority of the control heavy code, and upon encountering a kernel block, control will transfer to the accelerator to execute that kernel quickly and efficiently.

Communication of data between the CGRA and the general-purpose processor's main memory occurs through two methods. The run-time system transfers any live variables or arrays when control is transferred to the accelerator and back. Special streaming operations handle data transfer during kernel execution. A stream accessor is defined for each stream that specifies how data is read from memory and sent to the kernel, and how it is written from the kernel back to memory. Within the kernel, stream-receive and send operations will retrieve and write values to the streams. To allow for more aggressive loop-pipelining of the kernels, the ordering semantics of the stream accesses are relaxed from strict program order. Instead, accesses to the same stream will be ordered relative to each other, but there is no guarantee on the ordering of accesses between different streams.

In Mosaic, a kernel executes as a single data-flow graph in a heavily pipelined loop. To make this possible, Macah applies a generalized form of flattening to turn nested control flow into a single loop with predicated blocks. This allows the kernel to execute with a large amount of instruction level parallelism when mapped to a spatially distributed architecture with many compute units. The research that went into Macah developed enhanced loop flattening in parallel with the work presented as part of this dissertation. As a result, the capabilities of the Macah compiler have varied over the course of the research presented here. In particular, for the work presented in Chapter 3 and Chapter 4, this flattening wasn't fully developed, so the benchmark programmers hand-flattened the kernels. In addition, the programmers used manual back-substitution of the induction variables, as described in [TLS90], to overcome some inefficiencies in the automated loop-induction logic generation. As the Macah front end matured and the enhanced loop flattening support came to fruition, the need for hand-flattening was eliminated. The benchmarks were returned to versions with more traditional, easier to program nested loops and conditionals. These versions were used to evaluate resource sharing across mutually exclusive code paths in Chapters 5-9.

Macah currently supports three primary modes of compilation. The first mode is used for program development and debugging. It is a translation to C code for execution entirely on a general-purpose processor and debugging with traditional tools. The second mode creates a data-flow simulation of the kernel. The kernel portion drives a Verilog simulator in concert with the sequential code compiled for the general-purpose processor. This simulation exposes the parallelism of the kernel, which may help identify more concurrency bugs, as the execution semantics are a closer match to the final accelerator target. The final mode of compilation produces the intermediate data-flow graph that will be mapped to the CGRA architecture. The back-end CGRA mapping tool, SPR, uses this data-flow representation to map the program to the CGRA architecture.

As part of this process, Macah can also maintain and output control dependence information. This information indicates which operations were control dependent on values from other operations according to the original nested control flow, and additionally any control flow synthesized in the compilation process. Once a kernel has been flattened,

it is difficult to re-construct this information, so maintaining this information has proven useful in allowing SPR to share resources based on mutually-exclusive control.

2.3 Architecture Generation

The CGRA architectures targeted by this tool-chain are generated from a set of parameters and basic cells describing the compute units and their organization. The architecture generator was created as part of the dissertation work of Brian Van Essen [VE10] exploring power efficiency in CGRAs. The architecture generator is built as a plug-in to the open-source Electric VLSI Design System [SS].

The use of the Electric system provides a way to define the basic computation units that will be used in the architecture in a schematic-capture environment with Verilog annotations. The architecture generation plug-in then reads an architecture configuration file and uses these base cells to generate a full CGRA.

CGRAs generated in this manner consist of clusters of compute unit cells that are automatically connected via a full crossbar. These clusters are then connected together with a top-level grid-style interconnect. Each cluster is connected to a switchbox of the grid interconnect, and each switchbox is connected to the four neighboring switchboxes in the grid. A block diagram illustrates this in Figure 2.2.

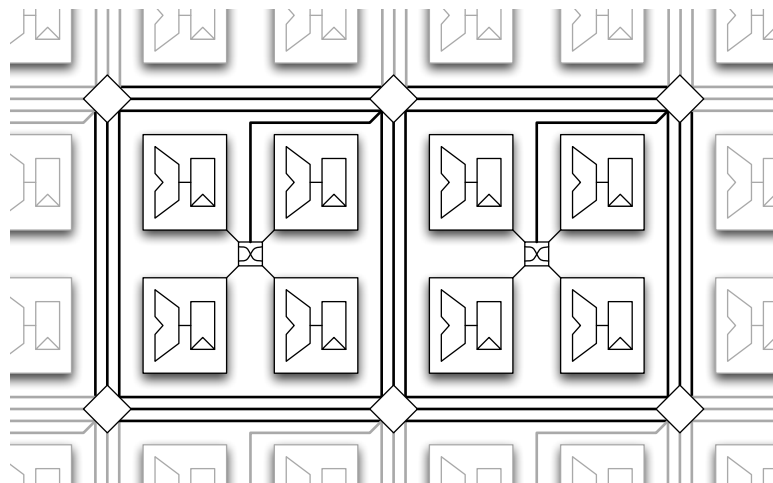


Figure 2.2: A 2x1 cluster CGRA block diagram.

2.4 Mapping Applications to CGRAs using SPR

Previously, a number of CGRA architectures have been proposed, including RaPiD [ECF96], ADRES [MVV⁺03a], MATRIX [MD96], Tartan [MG07], MorphoSys [SLL⁺00], and HSRA [TMJ⁺99]. These architectures sampled the possible design space and demonstrated the power, performance, and programmability benefits of using CGRAs.

Each of the previously mentioned CGRA projects required custom mapping tools that supported a limited subset of architectural features. We developed a new adaptive mapping tool to support a variety of CGRAs. We call this architecture-adaptive mapping tool SPR (Schedule, Place, and Route). SPR's support for features unique to CGRAs makes it a valuable tool for architecture exploration and application mapping across the CGRA devices that have and will be developed.

2.4.1 Related Work

Despite the large number of CGRAs that have been proposed in the literature, little in the way of flexible tools has been published. Most projects have mapping tools of some form, but they are tied to a specific architecture and/or are only simple assemblers that aid mapping by hand. The most flexible are DRESC [MVV⁺02] and the tool in [LCD03], both of which only support architectures defined using their limited templates.

Of the existing tools, DRESC is the closest to SPR, as it is also intended as a tool for architecture exploration and application mapping for CGRAs. DRESC exploits loop-level parallelism by pipelining the inner loop of an algorithm. Operators are scheduled in time, placed on a device, and routed simultaneously inside a Simulated Annealing framework. Their results indicate good quality mappings, but the slowdown from using scheduling, placement, and routing jointly within annealing makes it unusable for all but the smallest architectures and algorithms. DRESC only supports fully time-multiplexed resources, not the more efficient statically configured resources of architectures like RaPiD.

CGRA mapping algorithms draw from previous work on compilers for FPGAs and VLIW processors, because CGRAs share features with both devices. SPR uses Iterative Modulo Scheduling [Rau94] (IMS), Simulated Annealing [KGV83] placement with a cool-

ing schedule inspired by VPR [BR97a], and PathFinder [ME95] and QuickRoute [LE04] for pipelined routing.

IMS is a VLIW-inspired loop scheduling algorithm. IMS heuristically assigns operations to a schedule by specifying a start time for each instruction, taking into account resource constraints in addition to data and control dependencies. SPR uses IMS for initial operation scheduling, and we have extended IMS to support rescheduling with feedback from our placement algorithm, letting us handle the configurable interconnects of CGRAs.

FPGA mapping tools typically use Simulated Annealing for placement and PathFinder for routing. VPR, which has become the de facto standard for FPGA architecture exploration, is similar to SPR in that it seeks to be a flexible and open mapping tool that can provide high quality mappings and support a wide spectrum of architectural features. Unfortunately, it only applies to FPGAs. Given the demonstrate success of VPR, SPR adopts the same base algorithms for the placement and routing stages, though they have been extended to CGRAs by supporting multiplexing and solving the placement and routing issues that arise when using a fixed frequency device.

SPR uses QuickRoute to solve the pipelined routing problem. More recently, QuickRoute was extended to perform timing-driven routing [EH06] and have reduced memory complexity [CE06]. SPR does not yet incorporate these extensions, but they may be added in the future.

As shown by the authors of DRESC [MVV⁺02], mapping applications to CGRAs has similarities to the problems of scheduling computations on VLIW architectures and placing and routing computations on FPGAs. The difficulty comes in making these algorithms work together and adapting them to the particulars of CGRAs. For mapping, the application is represented as a *dataflow graph* (shown in Figure 2.3) and the architecture as a *datapath graph*. We describe our architecture representation in Section 2.5.

In DRESC, the authors chose to implement this mapping process as a monolithic scheduling, placement, and routing algorithm unified within a Simulated Annealing framework. Integrating placement and routing this way was shown to be significantly slower for FPGAs in Independence [SHE05] when compared to the separate stages of

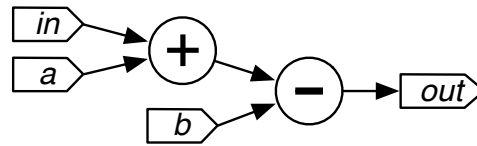


Figure 2.3: Example of a simple dataflow graph.

VPR [BR97a]. The slowdown could be even worse when including the scheduling for time-multiplexed coarse-grained devices, so the Mosaic tool-chain avoids the monolithic approach. Instead, the mapping process is divided into three closely coupled but distinct algorithms:

- Scheduling - ordering operations in time based on data and control dependencies.
- Placing - assigning operations to functional units.
- Routing - mapping data signals between operations using wires and registers.

To illustrate how these algorithms are combined, the main loop of SPR is shown in Algorithm 2.1. It uses IMS [Rau94], Simulated Annealing [KGV83] placement, and PathFinder [ME95] for negotiated routing. QuickRoute [LE04] provides the signal level routing for PathFinder to produce pipelined routes. This allows flexible use of interconnect registers during routing, rather than being limited to fixed register placement in register files.

The other interesting subroutines are described throughout the rest of this paper. The subroutine `unrollGraph()` of the datapath graph handles translating our architecture description into a graph suitable for placement and routing, and is discussed in Section 2.5.1.

SPR was designed with several assumptions based on the types of programs it will map and the range of current CGRAs. First, it assumes a kernel consists of a single loop to be pipelined. Currently, a kernel can be described in the Macah language using nested and sequenced loops, and the Macah compiler will turn them into a single loop dataflow graph [CFV⁺07]. Second, we assume we are mapping to a modulo-counter,

Algorithm 2.1: Main SPR Control Loop

```

1 begin
2   while iterate do
3     minII  $\leftarrow$  iterativeModuloSchedule(minII)
4     unroll datapath graph by minII
5     placeSuccess gets runSimulatedAnnealingPlacement()
6     if placeSuccess then
7       routeSuccess  $\leftarrow$  runPathFinderRouting()
8       if  $\neg$ routeSuccess then
9         increment minII
10    increment SPRIterations
11    iterate  $\leftarrow$  ( $\neg$ (placeSuccess  $\wedge$  routeSuccess)  $\wedge$  SPRIterations < maxIterations)
12 end

```

time-multiplexed architecture. That means the architecture can switch an entire configuration per clock cycle using a modulo-counter. Though some architectures support more complex control, this simple multiplexing is the most frequently implemented approach across a range of architectures. Finally, SPR currently assumes a fixed frequency device where routes cannot be constructed that violate that frequency. These assumptions are not fundamental, and may be lifted in future work, but they limit the scope of the SPR compilation problem to a manageable level while still applying to a broad set of CGRAs.

2.5 Abstract Representation

To achieve architecture-adaptability, SPR's architecture representation remains very abstract. An architecture is represented as a datapath graph, which is defined in Verilog out of a few primitives. Verilog modules prefixed with `primitive_` are used to represent arbitrary functional units. The Verilog is flattened into a directed graph. Primitive Verilog modules form the nodes and wires form the arcs. Two distinguished primitives receive special treatment:

- `primitive_register` - A register to be used by the router to route in time.
- `primitive_tap` - A configurable connection between two wires (pass gate).

Registers are distinguished so that QuickRoute [LE04] can use them for pipelined routing. Additionally, they are stored with improved efficiency by not representing them as nodes, but instead recording them as latency on arcs between nodes.

The connections in the interconnect that are controlled by configuration bits are represented as tap devices. A `primitive_tap` device is a dynamic connection, meaning it has an array of bits controlling its configuration allowing time multiplexing. A set of taps whose outputs are connected to the same wire are aggregated into a logical mux by SPR to ensure that two taps are never made to drive the same wire at the same time.

The user must do four things to support a new architecture. First, the user must describe the architecture in Verilog using primitive nodes as outlined above. Second, the user must create a function to estimate the cost of routing from one node to another. This is used for placement cost calculations and the A* search in QuickRoute. Additionally, many architectures support the notion of clusters with cheaper/faster local interconnect and more expensive global interconnect. SPR represents this by assigning every node a cluster coordinate, where SPR assumes anything with the same coordinate is in the same cluster. Third, the user must define a mapping between dataflow graph operation types and primitive functional unit types. This mapping is a many-to-many relation, for example mapping either an ADD or an OR operation onto an ALU functional unit, or mapping an ADD operation to either an ALU or an ADDER functional unit. Finally, the user must write a subroutine for translating the abstract internal configuration into an appropriate form for the architecture, such as a bit-stream. The user writes SPR plug-ins to handle the second through fourth items. This minimal amount of work to support a new architecture allows easy adaptation to a variety of CGRAs. The following sections provide a more detailed explanation of each stage, now that an overview and the abstractions of SPR have been presented.

2.5.1 Scheduling

The scheduling problem is addressed in SPR using the IMS [Rau94] algorithm. The result is a complete schedule that specifies when each operation can execute given the data dependencies and architectural resource constraints. The schedule repeats every II (Initi-

ation Interval) cycles, with a new iteration of the application loop starting each repetition. The II is determined by several things, and the reader is directed to [Rau94] for the details, but one that is important to our discussion is the maximum recurrence loop. When values from the current iteration are needed by future iterations, those values must be computed before the future iteration needs them. This is called a recurrence loop or loop carried dependence, and the largest recurrence loop is a lower bound on the II. This will be important when we are discussing our CGRA-specific extensions to the placer.

Using IMS allows us to easily trade off between resource constraints and throughput. To illustrate this, consider the following examples. Table 2.1 shows a possible schedule for our example dataflow graph from Figure 2.3. The target datapath graph in this example contains one ALU, one stream-in device, one stream-out device, and one constant device. This example requires two ALU operations and two constants per iteration. However, the architecture only has one of each, requiring the schedule to only start an iteration every other cycle, with a four cycle latency.

Table 2.1: Example schedule with II 2 and length 4.

Cycle	alu	str.i	str.o	cnst	
0		in[0]		a[0]	lt0
1	add[0]			b[0]	
2	sub[0]	in[1]		a[1]	lt1
3	add[1]		out[0]	b[1]	
4	sub[1]	in[2]		a[2]	lt2
5	add[2]		out[1]	b[2]	

If the resources are increased by adding an ALU and another constant, it is possible to use the schedule given in Table 2.2. The schedule length for a single iteration is still four cycles, but a new iteration initiates every cycle. This yields an II of one, thus doubling the throughput.

Operators that do not fall on the critical scheduling path may have some schedule slack that allows their start time to change without violating any dependency constraints. In these simple examples, b has some slack, and could be scheduled 1 cycle earlier. This schedule slack is preserved and communicated forward to the placer to provide flexibility

Table 2.2: Example schedule with II 1 and length 4.

Cycle	alu0	alu1	str_i	str_o	cnst0	cnst1	
0			in[0]		a[0]		lt0
1	add[0]		in[1]		a[1]	b[0]	lt1
2	add[1]	sub[0]	in[2]		a[2]	b[1]	lt2
3	add[2]	sub[1]	in[3]	out[0]	a[3]	b[2]	lt3

by allowing moves in time within the slack window. Initially, the schedule is tightly packed based on dependency and resource count constraints. Later, placement or routing may not be able to find a solution with this optimistic schedule, and re-scheduling will be done to lengthen it. This lengthening will produce more slack and possibly more virtual resources via the unrolling process described in the next section. This increases the chance of a successful place and route, at the cost of latency and/or throughput.

Modulo Graph Unrolling

Once the schedule meets dependency and resource usage constraints, placement and routing need to determine where operations execute and how they will communicate. However, there is a mismatch between the assumptions for scheduling and the assumptions for standard FPGA place and route algorithms. The scheduler assumes that all resources can be made to do a different operation in each cycle of the II; that is, it has virtualized the resources by a factor of II. Standard FPGA place and route algorithms do not support this type of virtualization. To overcome this difference, SPR unrolls the architecture graph II times, making one copy of the architecture for each cycle of the II. Each cycle within an II is referred to as a *phase*. SPR also re-maps connections with non-zero latencies so that they cross the appropriate number of phases, effectively routing forward in time. Since a modulo schedule is being used, we create a modulo graph by wrapping any connections beyond II phases back around to the beginning. This datapath graph transformation is legal as long as II is less than the depth of the chip's configuration memory, so the unrolling is limited to ensure this. Unrolling the graph turns the CGRA's time dimension into a third space dimension from the point of view of the placer and router, as shown

in Figure 2.4, allowing standard algorithms to be used. This figure illustrates the usual spatial routing on wires and through switch boxes, but registers actually route forward in time to the next cycle, shown with dotted lines.

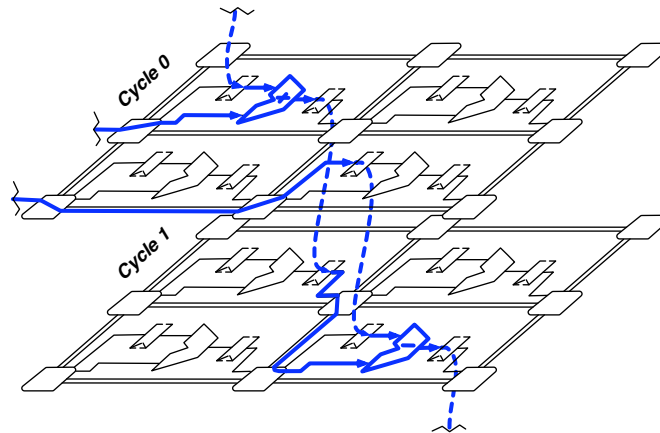


Figure 2.4: Unrolled datapath graph with mapped dataflow graph.

SPR maintains information about the unrolled nodes' correspondence to physical devices. Each unrolled copy of the graph corresponds to a specific phase of the II. Additionally, SPR annotates all dataflow graph nodes with their start time and slack from the schedule. This extra information allows the placement to restrict moves to phases of a device that preserve a legal schedule, but still generate moves in both space and a window of time.

At this point, it is important to note the difference between what is termed a stateful and a stateless device. For stateless devices, such as an ALU, increasing the II adds another virtual device, because it provides another schedule slot for the physical device. However, this does not work for some devices, such as memories, and we denote those as stateful. An example of this would be a small block RAM, because no matter how much the compiler “unrolls” the graph, the same data will be in the same physical memory. With an increase in II, the schedule gets more read and write accesses to the block RAM, but it does not increase the storage capacity of the memory. For these stateful devices, SPR handles the constraints of keeping only one state element in a device, but can virtualize

the accesses to the device. It groups these accesses so they are mapped to the same physical device.

2.5.2 Placement

Like most FPGA tool flows, SPR's placer uses Simulated Annealing [KGV83]. When using the Simulated Annealing framework, three key problem-specific components must be defined: the cooling schedule, move function, and cost function. The cooling schedule of VPR [BR97a] is used because it was shown to work well for FPGAs, and once we have unrolled our architecture, it is very close to a standard FPGA placement problem.

The move function in SPR is more complicated than that for an FPGA. The scheduling and stateful element constraints are enforced through the move function by only generating moves that respect these constraints. SPR starts generating a move by choosing a random dataflow node. SPR creates a shuffled list of the physical datapath nodes with a compatible type, and chooses the first one. After that, it chooses a random phase from the set of phases in the current schedule slack for the node. As a final check, SPR checks that the dataflow node at the destination datapath node is compatible with the phase and type of the current datapath node, and if so, it generates a successful swap move. If not, it tries different destination datapath nodes until it finds a swap or possibilities in the list are exhausted. In the latter case, SPR chooses a different initial dataflow node, re-starting the process. If the entire shuffled list is not used, SPR caches it for use in future move creation. In addition to this simple swap move, a more complicated clustering move function has been implemented, and is described in Section 3.3.

The last thing that needs to be defined is the cost function. SPR uses a routability-driven cost function, with routability estimates defined on a per-architecture basis. For the architecture used in the evaluations here, this cost function takes a current location, a destination, and a latency and estimates number of muxes needed to reach that destination with the exact latency. Given a cost function to estimate the routing cost, the cost of a placement is the sum of the routing cost over all connections in the architecture, plus a penalty for each unroutable connection. These unroutable connections arise because SPR targets fixed frequency devices, and if a route must traverse a large portion of the chip,

there will be some forced registering along the way to keep clock frequencies high. If a connection between two operations does not have the latency needed to meet the forced register delay constraints, it is marked as “broken.” These broken connections incur a penalty cost proportional to the amount of latency that would need to be added to meet the delay constraint. The base penalty value is an option that defaults to the maximum value of an integer divided by 200, to provide some headroom in calculations.

2.5.3 Routing

Routing the signals between the operators in a scheduled and placed dataflow graph requires finding paths containing zero or more registers. To accomplish this, SPR uses QuickRoute [LE04], a fast, heuristic algorithm that solves the pipelined routing problem.

By using PathFinder [ME95] with QuickRoute, SPR has a framework that negotiates resources conflicts. As a general conflict solver, PathFinder can be applied to a range of problems that can be framed as a negotiation. Originally, PathFinder was used to optimize FPGA routing by negotiating wire congestion. When applied to our unrolled architecture graphs, the original PathFinder will work unmodified for dynamically configurable resources, where the configuration can be changed on every tick of the clock.

2.5.4 Resource-Performance Tradeoff

One problem with using a system like an FPGA for an accelerator is that if the computation doesn't fit on the particular chip that is available, it won't run without adjustment of the application. Because SPR is designed for use in time multiplexed systems, more virtual hardware resources can be made available at the cost of slower execution. The benefits of this time multiplexed flexibility are illustrated in Figure 2.5. In this case, the number of compute-unit clusters are varied across the X axis of the graph. As the same application is mapped to architectures with fewer resources, there is still a valid mapping, but the II's are increased to make more resources available, with a corresponding decrease in throughput. Similarly, larger applications can be placed on the same size architecture at the cost of throughput. In the case illustrated in Figure 2.5, the application size is in-

creased by using more coefficients in the FIR. The 40 coefficient FIR consists of 255 nodes and 506 nets.

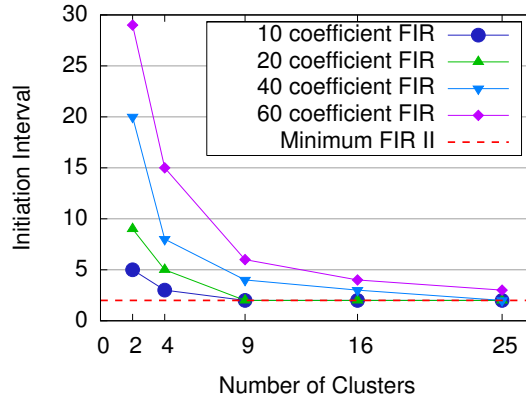


Figure 2.5: Scaling across architecture sizes.

This flexibility through virtualization means larger computations can be accomplished at the cost of time, a common tradeoff for software meant to run on general purpose processors, but one that is often more difficult in spatial architectures such as FPGAs. In an FPGA-based system, changing the problem size in terms of what actions are performed in parallel is often a designer task. With time-multiplexed virtualization support, it can be transformed into a compiler task. The rest of this dissertation takes its inspiration from this added flexibility, and explores new ways to allow a compiler to make this trade-off. In particular, it investigates emulating time multiplexing in routing structures which do not actually have the hardware to do that time multiplexing, and it also investigates extending time multiplexing to data dependent multiplexing. If successful, these compiler extensions can lead to hardware savings and more complex control support, broadening the application base suitable for CGRA acceleration.

2.6 Simulation

Once an application has been mapped to a generated architecture, a configuration for that architecture is produced. The generated architectures are created in Verilog, and a co-simulation system has been built that will allow a Verilog simulator to run a virtual

CGRA alongside the general purpose processor portions of an application in another process. The simulation and sequential execution communicate through Verilog PLI calls, and a full run-time has been built for transferring live values back and forth and allowing stream based communications. This allows full application simulation. Comparing the results of this simulation to those obtained by compiling the Macah program to straight C code provides for functional verification of the mapping results. Additionally, the simulation can be instrumented to estimate performance and power measurements, as in [VE10].

Chapter 3

PRACTICAL SPR

SPR is a combination of a VLIW scheduling algorithm, an FPGA placement algorithm, and an FPGA routing algorithm. The initial implementation simply runs these in sequence, and if the mapping fails in placement or routing, the minimum II is set one higher than the II for the failed attempt, and the process starts over from the scheduling stage. This can rapidly lead to high IIs, yielding poor performance. This chapter introduces two techniques designed to help minimize the number of times the II is increased. The first is a latency padding technique that provides specific feedback from the placer to the scheduler indicating which connections are problematic. These problematic areas are scheduled with more slack in the next iteration to allow the placer more flexibility in meeting the constraints of a fixed frequency device with configurable interconnect. The second technique is a new dynamic clustering method used during placement to help keep the most critical communications local to an architecturally defined cluster.

These techniques were implemented in SPR in Java using the schedule, place, and route algorithms described in the previous chapter. A suite of 8 benchmarks were used for the evaluation, which constitute a set of algorithms with loop-level parallelism typically seen in embedded signal processing and scientific computing applications. With only a few benchmarks, a two-tailed paired T-test with $p = .1$ was used for establishing significance in the results. The list of benchmarks, along with the count of operation nodes, signaling nets, minimum recurrence II, and minimum latency is given in Table 3.1. Note that [FCVE⁺09] was printed with an out-of-date table relative to the rest of the results, the table here contains the corrected values.

The experiments used for these techniques targeted a 2-D grid CGRA architecture inspired by island-style FPGAs and grid style CGRAs. It is an array of 16 clusters containing 4 ALUs, 4in/4out stream accessors, 4 structures for holding configured constants,

Table 3.1: Summary of Benchmarks

Kernel	Nodes	Nets	II	Latency
Motion Estimation	413	774	5	37
Smith-Waterman	446	993	6	129
2-D Convolution	434	869	5	58
Matched Filter	378	745	5	30
Matrix Multiply	196	402	5	14
CORDIC	157	336	2	33
K-Means Clust.	449	998	7	30
FIR	255	506	3	46

and 2 local block RAMs. Devices in a cluster are connected by a crossbar. The clusters are connected with a 16-track pipelined grid interconnect using Wilton switchboxes [Wil97].

3.1 Latency Padding

The initial schedule optimistically assumes only operator computation latency and ignores data movement latency to get the tightest schedule possible. Unfortunately, the placer cannot always meet this optimistic schedule because some longer-range connections will have forced latency in them due to registering. The placer in SPR is time-aware – it can move operations in both space and time. If there is slack available in the schedule, the placer can shift the slack around by moving operations in time. This gives the placer the flexibility to migrate the slack and use it to handle the forced latency of long wires. However, if there is not enough slack, it needs to communicate this to the scheduler.

Latency padding performs this communication. Latency padding inserts extra latency requirements in the dependency graph seen by the scheduler. Padding of n between a producer and consumer operation forces the scheduler to separate the completion of the producer operation and the start of the consumer operation by n cycles in the schedule.

Once the placer runs to completion, any connections that the architecture-specific cost function marks as unroutable are candidates for padding. The placer finds the minimum amount of extra latency needed route the connection by repeatedly querying the cost function with higher latencies. SPR adds that amount of latency to the connection as

padding that the scheduler views as extra forced delay in the dependencies. The scheduler will create a schedule with the appropriate amount of delay between the source and sink. Once that schedule is transferred back to the placer, the placer will see the extra time between the source and sink as slack that it can use to move operations in time. Section 3.2 covers several options for how and where to add padding, along with their effects. The pseudo code in Algorithm 3.1 outlines the main iteration loop of SPR, along with the added padding step, just after the placement.

Algorithm 3.1: Main SPR Control Loop with Padding

```

1 begin
2   while iterate do
3      $\text{minII} \leftarrow \text{iterativeModuloSchedule}(\text{minII})$ 
4     unroll datapath graph by minII
5      $\text{placeSuccess} \leftarrow \text{runSimulatedAnnealingPlacement}()$ 
6     if  $\neg \text{placeSuccess}$  then
7       | pad schedule
8     else
9       |  $\text{routeSuccess} \leftarrow \text{runPathFinderRouting}()$ 
10      | if  $\neg \text{routeSuccess}$  then
11      | | increment minII
12      | increment SPRIterations
13      |  $\text{iterate} \leftarrow (\neg(\text{placeSuccess} \wedge \text{routeSuccess}) \wedge \text{SPRIterations} < \text{maxIterations})$ 
14 end

```

The subroutine `padSchedule()` implements the latency padding technique which communicates the need for extra slack in placement back to the scheduling stage. The goal of this padding is to directly add slack to problematic areas of the computation. This added padding will affect the scheduling of all down-stream operations and could affect up-stream operations and the II through recurrence relationships. Thus, once SPR adds padding, the mapping process must go all the way back to the scheduling stage and start again. The padding tells the scheduler where to insert more slack, and then the placer uses that extra slack to span the long latency interconnect. However, the next time through the placer may need the padding on a different connection due to the random

nature of simulated annealing. Fortunately, padding appears as slack to the placer. As the placer moves operations to earlier start times, it effectively propagates the slack to the operation's outputs, and as it moves operations later, it propagates the slack to the inputs. Ideally, the connections that need the slack in this following pass are nearby the newly inserted slack, so after some annealing, the slack can migrate to where it is needed.

3.2 Latency Padding Effects

Latency padding is an educated guess as to where more latency in the schedule will aid in placement and routing. It is not an exact guess, because adding latency for a given placement will result in a new schedule and a new placement, which may have different latency needs. Additionally, because the placer works in both time and space, the latency may be moved by the placer to make better use of it.

Given a "broken" connection (a single connection that is unroutable due to forced latency constraints), there are several possibilities for adding latency that may fix it on the next scheduling and placement pass. One possibility is to add padding latency only to the connection that is broken, effectively spreading out the operations on both sides of the connection. We will call this *connection padding*.

Another option is to pad by reserving more time for the source operation as a whole. This has the advantage that there will be slack on all outputs for the operation, which means the operation will be able to be moved in time easily by shifting slack from all outputs to all inputs. This gives the placer a little more flexibility in the next pass, but there is potentially more latency added to the system as a whole. We call this *operator padding*.

In addition to choosing what to pad, the placer must decide the amount to pad. Even though the placer knows amount of latency that is needed for routability in the current placement, that may not be optimal amount of padding due to the opportunity for re-scheduling. For example, if there are several broken connections, adding padding to only one may fix all connections if the placer can move the slack to a common ancestor of the sinks. Another possibility is that adding less than the full amount of padding may work,

because in the next pass, the relative scheduling of the operations will change, affecting the placement and the routing.

To get a minimal amount of padding, one could add only 1 cycle of latency at a time and re-run the schedule and placement until it succeeds. These extra iterations cost extra compilation time. To minimize the number of extra iterations, one could pad by the full amount needed to meet the interconnect minimum delay constraints. This work focuses on applications where throughput matters more than the overall latency. Thus, it should be fine to pad by the full amount where the extra latency only affects the overall program latency, not the throughput. However, padding a recurrence cycle could increase the II and decrease the throughput. Therefore, conservative padding could be worth extra compiler run-time for recurrence cycle connections.

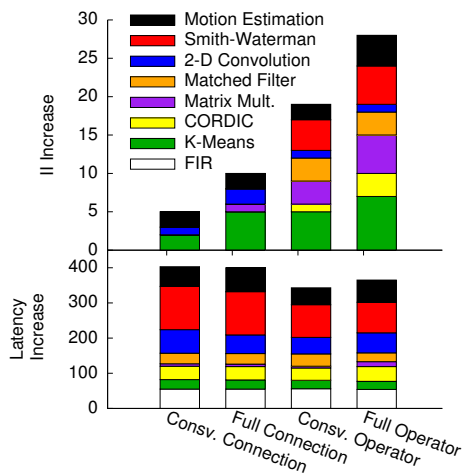


Figure 3.1: II and Latency effects across different padding settings.

shown in Figure 3.1. Note that some applications do not show up in the bars, and in this case, the II was not increased over the initially scheduled value.

Connection based padding is the best option for keeping throughput high, with the conservative padding producing slightly better results, as expected. Conservative connection based padding only results in an average 1.11x increase in II, compared to a 1.74x

The experiments test two amounts of padding, which are denoted *full* and *conservative*. For full padding, the placer inserts the full amount of latency that is needed to make connections routable according to the cost function. For conservative padding, the placer only inserts 1 cycle of padding on any recurrence cycle connections, but still inserts the full amount on other connections.

Between padding connections or operators and choosing either conservative or full padding in recurrence cycles, there are 4 options for how to do latency padding. The results of experiments run using these 4 possibilities are

increase with full operator padding. However, if latency is the primary concern, then conservative operator based padding is the best solution.

3.3 Dynamic Recurrence Clustering

Many architectures group resources into clusters with more flexible and lower latency interconnect. This is found in architectures like HSRA, MorphoSys, and Tartan. As we mentioned before, the largest recurrence loop in an application sets a lower bound on the application's Π . This means that these recurrence loops are effectively the critical path. In order to keep the throughput high, we want to make sure any critical loops in the application take advantage of the faster interconnect offered by clustered nodes in the architecture.

The basic idea behind the clustering is that when attempting to move an operation from one cluster to another, nodes from the same recurrence loop may need to be moved to the new cluster as well. This is to avoid higher Π 's due to inter-cluster communication in the critical loop.

The clustering algorithm starts out by marking all edges in recurrence loops as clustering edges. Then, when the placer is generating an inter-cluster move, it checks to see which neighboring nodes should be moved to the new cluster as well. Here, neighbors are defined as any other nodes a particular node is directly connected to. If the clustering edges to any neighboring nodes in this cluster would become unroutable, the placer attempts to include those neighboring nodes in the move to the destination cluster. To include the neighboring nodes, the placer attempts to either move the neighbor to the destination cluster or swap the neighbor with an operation already in the destination cluster.

Of course, the included neighbor's neighbors may be part of a recurrence loop and may need to be moved as well, so this process is repeated recursively until no new nodes are added. Any operations that have already been added to the move are marked so a recursively generated move will not try to move the same operation again. The placer limits consideration of neighbors to nodes that start in the same cluster, and moves/swaps

are only generated between the original source and destination cluster, so the biggest move generated will be a full swap of two clusters. The key here is that clustering only occurs when a connection would otherwise end up being unroutable, which would result in latency padding and an increase in II. This way, the placer only uses clustering where necessary, and recurrence loops with enough slack can still spread across clusters.

3.4 Dynamic Recurrence Clustering Effects

To test the effects of the dynamic recurrence clustering, SPR was run across the benchmarks with it both on and off. The results are shown in Figure 3.2.

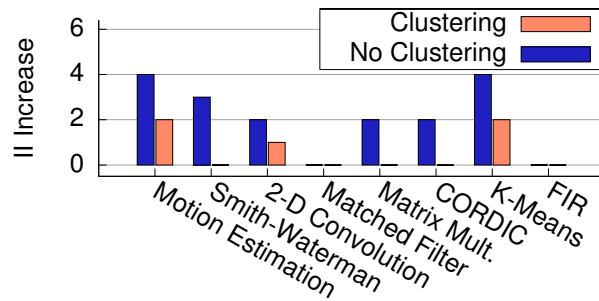


Figure 3.2: Effects of Clustering on II.

Clustering achieves improved IIs, translating into improved throughput, in six of the benchmarks. For the remaining two, the results were the same for both methods. Averaged across all benchmarks, this translates into a significant improvement of 1.3x in throughput. The lack of bars for most of the clustering cases in the chart indicates that for most of the benchmarks the placer and router were able to achieve the minimum schedulable II when clustering was turned on.

3.5 Conclusion

In this chapter, I demonstrated two extensions to SPR that help bridge the gap between VLIW and FPGA compilation algorithms. These extensions are needed to couple them into a practical CGRA architecture-adaptive mapping application for exploring the

CGRA architecture space. I demonstrated that the latency padding technique is successful coupling the VLIW-style scheduler and FPGA-style placer to meet the constraints of a fixed frequency device with configurable interconnect. After evaluating several methods of padding, we found that conservative padding on a per-unroutable connection basis achieved the best throughput. I also developed a new dynamic clustering method for placement that achieved an average improvement of 1.3x in throughput of mapped designs.

Chapter 4

STATIC INTERCONNECT SHARING IN SPR

This chapter introduces an enhancement to the PathFinder algorithm for targeting architectures with a mix of time-multiplexed and statically configured interconnects. The enhanced algorithm is able to successfully share statically configured interconnect in a time-multiplexed way, achieving an average channel width reduction of .5x compared to non-shared static interconnect.

A mux in the fully time-multiplexed portion of the interconnect will be called a dynamic mux. A mux in the statically configured interconnect will be called a static mux. A dynamic mux has a configuration entry that is switched on every phase of the modulo schedule, while the static mux only has a single configuration entry that will be used throughout the lifetime of the application.

Dynamic connections are more flexible, but have higher area and power requirements for storing and switching the configuration. Static connections can be more area and power efficient, but at the cost of flexibility. An example that demonstrates the difference between a static and dynamic muxes is shown in Figure 4.1. Including support for both static and dynamic muxes allows SPR to route in interconnects with a mixture of these resources, like those found in the RaPiD [ECF96] and MATRIX [MD96] architectures.

Dynamically time-multiplexed resources are the standard type of routing resources used by SPR as it has been presented thus far, and are represented using `primitive_tap` nodes. Muxes constructed of `primitive_tap` devices are unrolled into independent instances, as described in Section 2.5.1. A `primitive_stap` device is a static connection, meaning it has a single bit controlling its configuration for the life of the application. It will also be unrolled into independent instances to track signal usage at the output of a mux on a per-phase basis, but all of the unrolled instances from the same mux will share a single structure for tracking the desired configuration setting, corresponding to

the limitation of a single setting in the architecture. With this representation, a mix of static and dynamic interconnect can be used in SPR architecture descriptions; however, a single mux must be made either entirely of dynamic or entirely of static taps that share the same output wire.

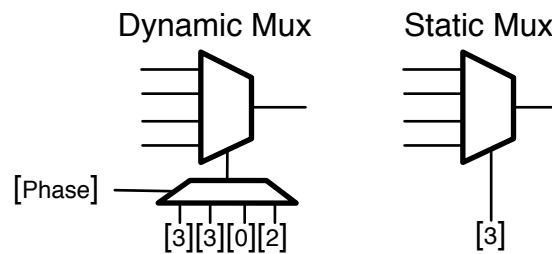


Figure 4.1: Dynamic and static mux representation.

The most straight-forward use of these statically configured resources is to let a single signal use them once, ensuring there will never be a conflict over which tap (input) should be active in the multiplexor. However, the work in this chapter will demonstrate that the negotiated congestion routing framework use by SPR, given the correct congestion metric, can be used to allow signals in different phases to share the static multiplexor by agreeing to use the same setting across all phases. This is done by separating costs for signal usage over time and for the configuration settings, and tracking congestion between the two separately. This provides the negotiation framework the foothold needed to promote agreement of tap usage across different phases.

4.1 *Static and Dynamic Routing*

When applied to the modulo-unrolled architecture graphs, the original PathFinder will work unmodified for dynamically configurable resources, where the configuration can be changed on every tick of the clock. Reconfigurable systems such as RaPiD [ECF96] use more area-efficient and power-efficient interconnect for portions of the system that are set up statically before a computation. We have extended PathFinder to allow sharing of both static and dynamic muxes between signals in different clock cycles. To see what

this sharing means, consider the routes shown in Figure 4.2 across different phases of the same mux.

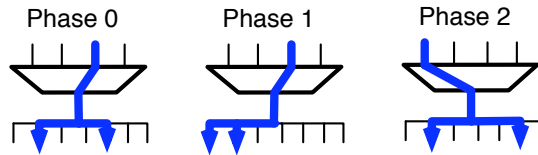


Figure 4.2: Routes across different phases of a mux.

If this is a dynamic mux, all three signals can share this mux. A static mux would allow sharing of the signals in the first two phases because they share the same input, even though their destinations may be different. The signal attempting to use the mux in Phase 2 would need to be re-routed. Without enhancement, PathFinder is unable to support this type of sharing in mixed static/dynamic architectures.

4.1.1 Control-based PathFinder

In the enhanced version of PathFinder presented in this chapter, static and dynamic resources appear the same to the signal level QuickRoute algorithm. The difference lies in the computation of the congestion costs during the congestion negotiation. The problem with using standard PathFinder is that virtual copies of static muxes appear as completely disjoint routing resources to PathFinder. Even though all virtual copies of a static mux need to have the same input configuration to have a valid mapping, PathFinder will obviously route through different inputs in different phases. On the other hand, dynamic muxes have no constraints on the settings between phases, and so by simply unrolling the graph, the original PathFinder formulation supports dynamic muxes. A straightforward PathFinder extension for supporting static muxes is to simply allow only one signal to ever use a particular static mux, effectively not unrolling it. This can be accomplished by summing the signal counts across the phases. This will only allow 1 signal to ever use a static resource. However, if signals in different phases can all use the same setting for the mux – route through the same input – then multiple signals can all use that mux even though it only supports a single configuration.

Two observations lead to the new PathFinder formulation which can time-multiplex signals across static muxes. The first is a simple optimization to PathFinder. Notice that PathFinder only needs to negotiate between signals for the use of a mux output port, not all of the individual wire segments and registers driven by that output port. This is because by choosing which signal will occupy the output port of a mux, we have implicitly chosen that the same signal will occupy all wire segments connected to the port, either directly or indirectly through a series of registers. This allows for a dramatic reduction in the number of resources that must be tracked with PathFinder negotiation information.

The second observation is when routing in an interconnect made up of muxes, wires and registers, the only routing choices to be made consists of selecting the input of a mux to pass to the output – what should the value of the configuration be for that mux. Each available configuration allows the router to choose a different direction of travel for a signal. This makes the relationship between static and dynamic routing resources more apparent. In the dynamic case, there is a separate configuration available for each phase of the II, so the router can choose a different input in every phase. In the static case, the router can only choose to have one input used for the life of the program. As long as the route for every signal going through a mux uses the same input, the single configuration can be “shared” across the signals. Thus, PathFinder must negotiate for use of the shared configuration across unrolled instances of a static mux.

Now there are two different kinds of congestion to allow for this new negotiation. The first is the original PathFinder notion of *signal* congestion: two electrical signals cannot be sent along the mux output wire at the same time. The second is *control* congestion: two signals using a statically configured mux cannot require two different configurations in different phases, but both can use the output wire in different phases.

In the original PathFinder, congestion led to two types of cost: the immediate sharing cost and the history sharing cost. Now that there are two different types of congestion, there are a total of four costs to be monitored. We will begin with the signal and control immediate sharing costs. The immediate sharing cost for signal congestion remains unchanged from PathFinder, and is the excess number of signals attempting to use a mux in a given phase.

The immediate sharing cost for control congestion is the excess number of configurations used by a mux across all phases. For a dynamic mux, which can have a different control setting in each phase, this will always be zero. However, for a static mux, only one setting is available, so any excess settings needed by signals add to the immediate sharing cost.

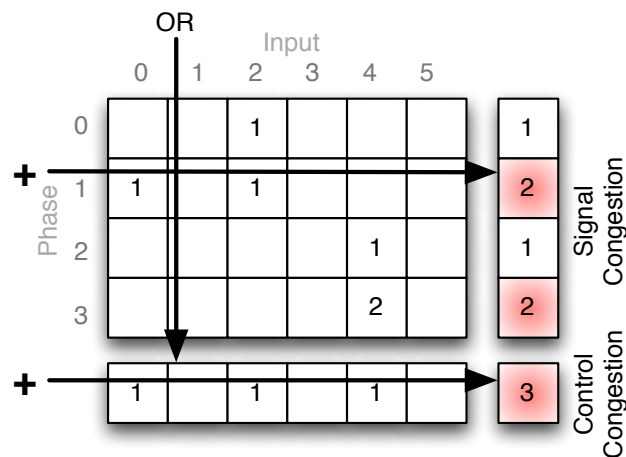


Figure 4.3: Congestion calculation from signal usage.

An example of computing the congestion for the different types of sharing on a 6 input static mux with an II of 4 is depicted in Figure 4.3. The large table in the middle counts the signals routed through a particular mux. Each column represents a different mux input, and each row represents an unrolled phase instance of the mux. The right-most column shows the results of calculating the congestion, and the bottom-most row is an intermediate result in the congestion calculation. The top half of the diagram illustrates the original PathFinder signal congestion costs for each unrolled instance of the mux. By adding up the number of signals routed through the inputs for a given phase, we see how many will be on the output in that phase. This leads to signal congestion on the phase 1 mux and the phase 3 mux because in both cases there are two signals trying to use the mux output simultaneously.

The bottom half of the diagram illustrates the new control congestion for a static mux. First, each input is checked across all phases to see if it is ever used, performing a logical OR across phases. Then the results of the usage check are summed across the inputs to see how many unique configurations are used over the whole table, obtaining 3 in this example. Because this is a static mux, a maximum uncongested value is 1 input, so the value of 3 here indicates control congestion. Using both types of congestion in PathFinder is straightforward, as PathFinder simply needs to penalize congested resources and iterate until there is no congestion of either type.

Now consider the signal and control history sharing costs. Again, the history sharing cost for signal congestion from PathFinder is unchanged. The history cost for control congestion is a little more subtle. For a fully dynamic mux, there is no control congestion history to maintain.

Algorithm 4.1: Congested Mux History Update

```

1 begin
2   foreach sig of signalsOnCongestedMux do
3     foreach phase of II do
4       foreach input of mux do
5         if ( $sig.input \neq input \vee (sig.input == input \wedge sig.phase == phase)$ )
6           then
7             increment historyControlCost[phase][input]
8         end if
9       end foreach
10    end foreach
11  end

```

The goal of the negotiation process is to encourage the signals using the mux in different phases to use the same inputs for a single static mux. The router must compute the history cost update by looking at the mapped signals across all phases of a mux with control congestion. Pseudo code for this update is given in Algorithm 4.1. This update is only applied to muxes with control congestion. For each signal using one of the unrolled muxes, there are two basic increases to the control cost. The first is an increase to the cost of any input other than the one the signal is currently using, represented by the condition

$sig.input \neq input$. The second is an increase to the cost of the current input and phase that a signal is using, represented by the condition $(sig.input == input \wedge sig.phase == phase)$.

The reasons for these two increases will be discussed in the context of an example where two signals, A and B, are using a 6 input static mux with an II of 4. The resulting history cost increases are depicted in Figure 4.4. Inputs with history cost increases from A are shaded in light red and increases from B are shaded in dark blue. Inputs that are shaded by both will have their cost increased by twice as much as those shaded by a single color.

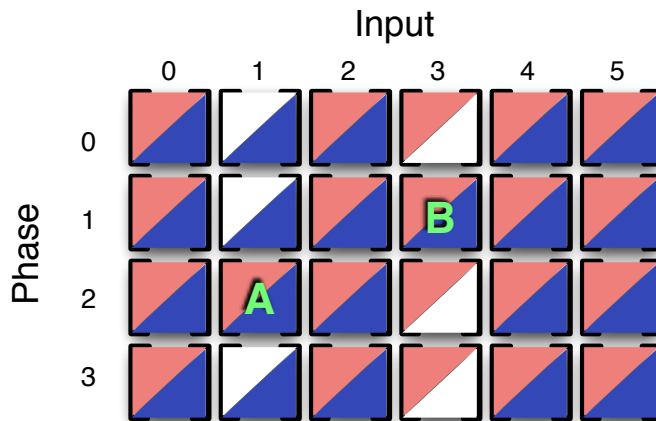


Figure 4.4: History updates for static control congestion.

For the condition $sig.input \neq input$, each signal increases the cost of using a configuration (or input) other than its own. This makes that same input in other phases relatively less expensive in future iterations. As the cost increases, either signal A or B could find a cheaper alternate route through a different mux, or through an input compatible with the other signal. For example, A might find an alternate route that uses input 3.

For the condition $(sig.input == input \wedge sig.phase == phase)$, each of A and B increase the cost of their input in the phase they use it. To see why, consider what would happen if signals did not increase the cost of their current input. Suppose the depicted congestion of A and B is the only congestion in the current routing. At this point, the paths A and B are taking are the least cost paths from their sources to their respective sinks. When each signal only increases the cost for inputs it is not using, the cost of the currently used

inputs will increase at the same rate. The cost of all others will increase at twice that rate. Since the currently used inputs are penalized by the same amount, neither signal has incentive to take a longer path to use compatible inputs. The only way this congestion will be resolved is if A or B move to a different mux altogether, not by both using the same input.

When A and B *do* increase the cost of their input in the phase they use it, each signal will notice that “the grass is greener” on the input the other signal is using. Whichever signal has a cheaper alternate route to the other input will eventually move first, and the congestion will be resolved, sharing the same input of the static mux in different phases. In this way, we can have several signals sharing the static resources cooperatively.

Note that if the router only considered history costs, the routes could oscillate, with A and B both switching each iteration. However, the immediate sharing cost avoids this oscillation – as soon as the first signal moves, the immediate sharing cost is eliminated. Either the second signal must choose the same path it currently uses, or pay a significant immediate sharing cost to switch to another input. This use of history costs to negotiate long-term congestion, and immediate costs to eliminate oscillations is a major feature of the original PathFinder algorithm. A more thorough treatment of the convergence of PathFinder can be found in [CS00, CSEM00]. SPR re-routes only congested signals and QuickRoute always chooses the cheapest path, which corresponds to the combination of *ReassignAll* = **FALSE** and **SelectCheapest** where the authors of [CS00, CSEM00] show the negotiated-congestion technique converges.

4.2 Evaluation

To evaluate the enhancements, and to provide a tool for the Mosaic architecture exploration [EH11], these techniques were implemented in SPR in Java using the schedule, place, and route algorithms described in the previous sections. SPR is tested on a suite of 8 benchmarks, which are algorithms with loop-level parallelism typically seen in embedded signal processing and scientific computing applications. With only a few benchmarks, a two-tailed paired T-test with $p = .1$ was used for establishing significance in the results.

The architectures are defined as structural Verilog suitable for simulation, generated by the Mosaic Architecture Generator plug-in to the Electric [SS] VLSI Design System.

4.2.1 Architecture

The experiments targeted a 2-D grid CGRA architecture inspired by island-style FPGAs and grid style CGRAs. It is made up of a 2-D array of clusters containing 4 ALUs, 4in/4out stream accessors, 4 structures for holding configured constants, and 2 local block RAMs. Internally, devices in a cluster are connected by a crossbar. The clusters are connected to each other with a pipelined grid interconnect that can vary in the number of static and dynamic channels. The grid interconnect employs Wilton style switchboxes [Wil97]. An architecture made up of 16 clusters with a 16 track interconnect was used unless otherwise indicated. In total, the test architecture contains 288 functional units, though the SPR algorithms should scale to thousands of functional units on the basis of their lineage from existing VLIW and FPGA algorithms. SPR will eventually support pipelined functional units which take multiple clock cycles for computation, but single cycle operations are assumed for the simple test architecture.

4.2.2 Benchmarks

The benchmarks were written in Macah with the main loops designated as kernels. These kernels were translated by the Macah compiler into a dataflow graph [CFV⁺07]. The dataflow graph nodes are primitive operations. The nets are either routable connections or dependency constraints, such as sequential memory accesses, that must be respected by the scheduler. A tech mapper translates compiler-specific nodes into SPR readable generics and maps from dataflow graph node types to devices in the architecture.

The benchmarks include three simple signal processing kernels: *fir*, *convolution*, and *matrix multiply*, and five more complex kernels from the multimedia and scientific computing space: *K-Means Clustering*, *Smith-Waterman*, *Matched Filter*, *CORDIC* and *Heuristic Block Motion Estimation*. Relevant statistics for the benchmarks were given in Table 3.1.

4.2.3 Static Interconnect Sharing

This section evaluates the new static congestion negotiation algorithm. In the experiments in this section, the applications from the benchmark suite are mapped to two different architectures, one with a fully dynamic global interconnect, and one with a fully static global interconnect. In both architectures, the cluster crossbar is fully dynamic, providing a way to multiplex signals onto and off of the global interconnect. The experiments measure sharing by counting the number of signals mapped to a physical mux in different phases.

The fully dynamic run gives the baseline for the amount of sharing that would happen in the most flexible system possible, given the application. With a fully dynamic interconnect, there are no constraints put on signals sharing a mux in different phases. It is *sharing neutral*, and should neither encourage nor discourage sharing. On the other hand, the static interconnect has two forces at work to perturb the amount of sharing. They both originate from the constraint that two signals who share a mux in different phases must use the same input. The extra costs that encourage signals to use the same input can work to lower the amount of sharing in routing rich architectures, because when there is a conflict, it is easier for one signal to use a slightly longer but alternate route through empty muxes than to use the same input as the competing signal. As the routing resources become more scarce, there will be fewer empty muxes to use and some sharing will be required. Once two signals have negotiated to use the same input for one mux, that means both signals are also using the same upstream mux. In fact, this holds transitively, so if a mux is shared, the same amount of sharing will be forced on all upstream static muxes until a dynamic mux is reached. This will tend to increase sharing.

The baseline dynamic sharing is an average of 1.35 signals/mux for the benchmarks at a fixed channel width of 16. As a check to ensure this number is reasonable, we calculated the expected value of sharing given the II, global interconnect utilization, and assuming a uniform distribution of sources and sinks, and found it to be 1.21 signals/mux. The measured sharing should be slightly higher because the actual distribution is not uniform. The fully static run achieves an impressive 1.32 signals/mux, which is not a significant

difference from the dynamic case according to the paired T-test. This demonstrates that with a fixed amount of interconnect, possibly over-provisioned, the new sharing algorithm is able to achieve a signal density on restricted hardware which is on par with what is achieved on a more flexible architecture.

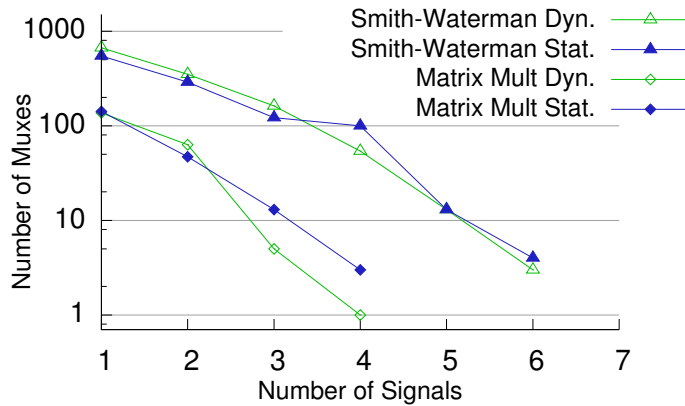


Figure 4.5: Histogram of dynamic and static sharing. The channel width for these tests was set at the minimum routable channel width for the static sharing algorithm.

Examining sharing at the stress case of minimum routable channel width begins to differentiate more between static and dynamic resources. In this case, a higher utilization should lead to higher amounts of sharing. When the channel width is reduced to the minimum routable width per benchmark, there is an increase in sharing in the static interconnect to 1.52. This can be compared to the sharing obtained by running the benchmarks with a dynamic interconnect sized to the same per-benchmark minimum static channel widths. The dynamic case has lower sharing at 1.47, a significant difference from the static case according to the paired T-test. A possible explanation for this is that in the stress case, the upstream chaining is causing the higher sharing when using static resources. A histogram of the number of signals sharing a mux is plotted for two applications on both static and dynamic interconnect in Figure 4.5. Plotting the results for all of the benchmarks results in a very cluttered graph, so only the two benchmarks that provide a rough upper and lower bound on the sharing are presented here.

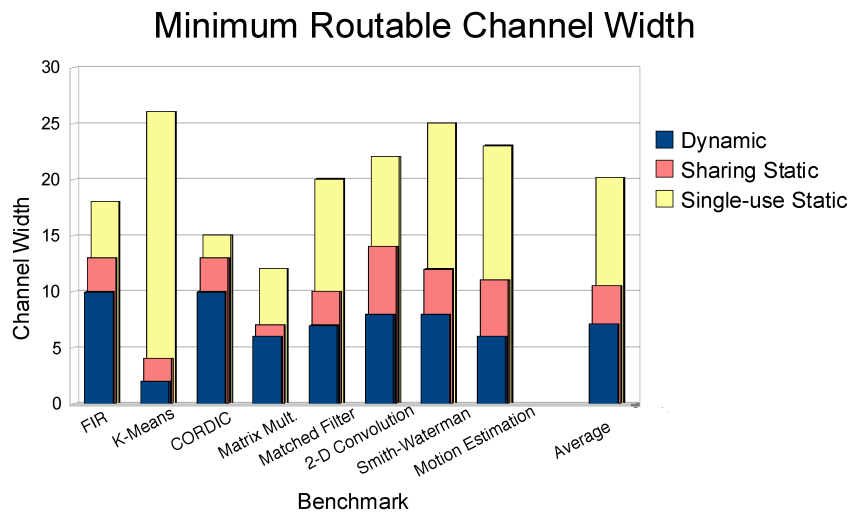


Figure 4.6: Channel width across the benchmarks for hardware, software, and no time multiplexing.

Experiments were run to find the minimum routable channel width across three configurations of SPR and architecture:

- Regular SPR mapping to an architecture with a fully dynamic interconnect.
- SPR with the static sharing enabled mapping to an architecture with a static global interconnect.
- SPR with static sharing disabled mapping to an architecture with a static global interconnect. Each static resource could only be used by one signal.

The results across the benchmarks are shown in Figure 4.6, along with the average. The average minimum channel width when using a dynamic interconnect is 7.13 channels. Using a static interconnect with no sharing increased the width all the way to 20.1 channels. By allowing sharing, the width greatly reduces to 10.5 channels on a fully static interconnect. This means that employing sharing provides all of the power and area savings of a statically configured interconnect while reducing the associated channel width to .5x of what is required when sharing is not allowed.

4.3 Conclusion

In this chapter, I demonstrated that a new sharing enhancement to PathFinder [ME95] can effectively exploit static interconnect in a time-multiplexed system. Given an architecture with statically configured routing resources, using the sharing algorithm provides clear benefits over statically allocating a single signal to statically configured resources, obtaining results that are closer to that of a fully dynamic system.

However, this does not indicate when an architecture should use static resources over fully dynamic ones. An investigation of this for the grid portion of the CGRA interconnect is presented in [VEWC⁺09]. The area-energy product was measured while sweeping the ratio of statically configured to dynamically time-multiplexed channels in the global interconnect across several architecture configurations. The results from that work are shown in Figure 4.7. These results indicate that shallow time-multiplexing depths and large word sizes favor fully time-multiplexed interconnects. As the word size is reduced and the time multiplexing depth is increased, eventually it becomes profitable to include statically configured channels.

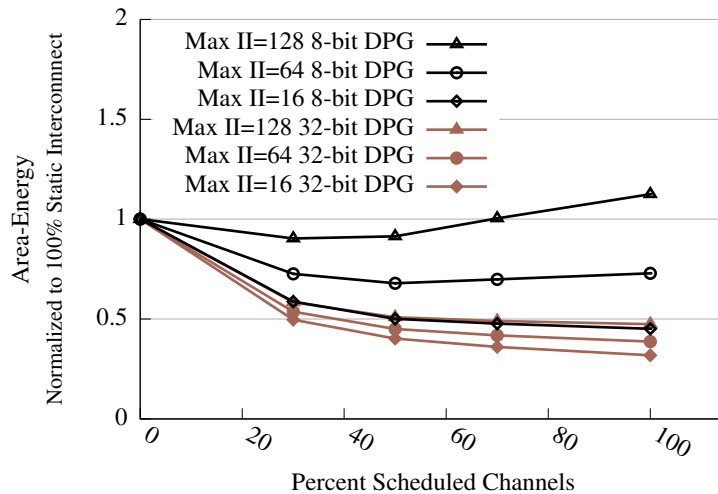


Figure 4.7: Fig. 7 from [VEWC⁺09]. Area-energy product for 32- and 8-bit datapath word-widths and maximum supported II of 16, 64, and 128 configurations, varying the percentage of interconnect channels that are statically configured or dynamically time-multiplexed.

As observed in [VEWC⁺09], one of the reasons that the statically configured resources are less efficient energy-wise is that any signals that share a particular static multiplexor will also share all downstream multiplexors, even if the signals are intended for separate destinations. This leads to a larger energy load for shared signals, which is significant in the global interconnect with large drivers. It may be that static resources are more beneficial in local communication and single-bit control oriented data-paths, but quantification of this benefit is left for future work.

The focus in this dissertation will now change to a different dimension of sharing. In spatial architectures like CGRAs and FPGAs, control flow is often handled through a form of if-conversion, computing both branches and selecting the appropriate result at the end. However, the program structure indicates that at run-time certain sets of these paths are mutually-exclusive – only one result of the set will ever need to be computed at a time. The following chapters explore what is required to share resources across these mutually-exclusive paths for a performance benefit. The next chapter will lay the foundations needed for extending SPR to support this sharing. Chapters 6-8 will introduce the extensions to scheduling, placement and routing respectively, which will be evaluated in Chapter 9.

Chapter 5

PREDICATE AWARE RESOURCE SHARING IN SPR

The previous chapters covered the primary algorithms behind SPR, how they are integrated, and how to extend them to share routing resources when full time-multiplexing hardware is only available in portions of the architecture. This chapter is the beginning of a shift towards supporting a different type of sharing in SPR. The next few chapters will focus on sharing resources across operations whose execution is guaranteed to be mutually exclusive at run-time.

A variety of ways to effectively apply modulo-scheduling to CGRAs have been explored [MVV⁺03a, PFM⁺08, OEPM09, FCVE⁺09]. With modulo-scheduled spatial architectures, the same configurations/instructions are repeated in a cycle, and any control is handled through some form of if-conversion, with parallelism extracted through software pipelining. This leads to two inefficiencies. The first is that resources are reserved to compute all control paths, even though only one path can be taken for a given iteration of a loop. The second deals with producing hardware that will span a variety of applications. Some applications will have a large initiation interval, leading to deep configuration memories that can support their large II. However, high throughput applications are those with a small II. For these applications, the over-provisioned configuration memory is simply wasted. It would be nice to be able to put that extra memory to use, perhaps to better handle more complex control flow in high-throughput kernels.

The goal of the next few chapters is to mitigate these inefficiencies. By extending an approach for VLIWs to CGRAs (predicate aware scheduling [SMDL03]), and introducing new constraints, abstractions and algorithms for placement and routing, I will demonstrate that it is possible to share resources across control paths by taking advantage of this otherwise wasted configuration memory. In predicate aware scheduling, the compiler is aware of the predication used to convert control dependencies to data dependencies. It

uses this information to schedule operations that are mutually exclusive at run-time onto the same physical resource, effectively sharing it across the control paths.

This chapter will provide the foundations for exploring predicate aware sharing in CGRAs. It begins with an example to illustrate predicate aware sharing on a simple program, and contrasts this sharing with execution in a general purpose processor and the typical modulo-scheduled scheme of CGRAs. Then it discusses an extension to the hardware model that will allow for run-time alterations of the modulo scheduled configurations. Finally, it describes how the predicate relationships are represented in SPR. Chapters 6, 7, and 8 will cover adding predicate awareness to the scheduling, placement and routing stages of SPR respectively.

5.1 Predicate Aware Sharing

Conditional execution is the ability of the hardware to choose which operations to execute based on run-time data. There are lots of useful signal processing applications that do not require conditional execution, such as a simple filter that always outputs a linear function of its input. However, we are interested in supporting a broader set of applications, including those that use heuristics. Even very simple applications can benefit from conditional execution. The example we will use is computing the sum of absolute differences. Pseudo-code can be found in Algorithm 5.1.

Algorithm 5.1: Sum of absolute differences example.

```

1 for  $i=0; i < n; i++$  do
2    $\text{diff} \leftarrow a[i] - b[i];$ 
3   if  $\text{diff} > 0$  then
4      $\text{sum} \leftarrow \text{sum} + \text{diff};$ 
5   else
6      $\text{sum} \leftarrow \text{sum} - \text{diff};$ 

```

Program Counter – In the Von Neumann model, the program counter for sequencing gives us the ability to handle conditional execution. In the case of Algorithm 5.1, that means conditionally executing either the *then* or *else* blocks of the *if* statement. Using run-

time data to update the program counter allows the processor to skip past any portion of code. An example in pseudo-assembly is shown in Figure 5.1(a) for the conditional if-else statement in Program 5.1.

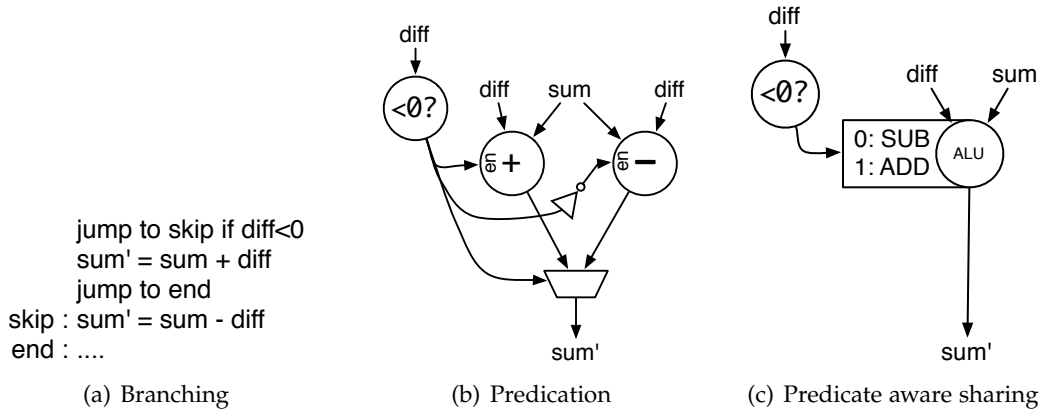


Figure 5.1: Examples of Program 5.1 using different conditional execution methods. Note: sum' refers to the new value of sum .

Predication – When moving to a statically defined modulo-schedule in a CGRA, the instruction stream is no longer easily altered. This is because for CGRAs, the equivalent of one instruction can be a very large configuration. In order to maintain scalability, it is better to have locally generated modulo counters instead of a global broadcast. This means there is no longer a central program counter that can easily be modified.

For spatial architectures, a common method of supporting conditional execution is through predication, sometimes combined with speculative execution. This method is illustrated in Figure 5.1(b). For predication, the operations for both paths of execution are set up to run in parallel. A control line indicating whether the difference is positive or negative is routed to enable inputs on both operations, so only the appropriate operation will execute. Then, the same control line is sent to a mux responsible for choosing the appropriate result to pass forward in the computation. A common optimization in spatial architectures is to remove the guard condition from any operations that have no external side effects, such as the add and subtract in this example [MLC⁺92, Cal02]. This allows

them to be scheduled earlier, effectively being speculatively executed with the result being selected from the computed alternatives.

Predicate aware sharing – Predication can be seen as having a particular resource choosing between two operations at run-time, a useful one and a *no-operation*. The functional units would be used more efficiently if they could instead switch between two useful operations. An example of this for a particular CGRA, RaPiD, is given in [ECF⁺97a], in the section that discusses mapping motion estimation to the hardware. As part of motion estimation, a sum-of-absolute-difference is calculated. In the example netlist, the sign result from the difference calculation is used to control whether the next ALU performs an add or a subtract. In this case, specifying that a datapath signal directly controls the type of operation of the functional unit is done by the programmer.

This method is illustrated in Figure 5.1(c). Here, the square extension to the ALU represents the local configuration memory for that ALU, with the subtract opcode stored at the 0 entry, and add stored at 1. The result of the comparison of *diff* to 0 is used to select between these instructions. By making the compiler aware of the mutual exclusion between the add and subtract that is controlled by the comparison, we can lift the burden of creating configurations like the RaPiD example from the programmer, and automatically generate them. This method can be extended to choose between more than just two conditional paths of execution.

The benefit of this method over predication is that functional units are not wasted for the control paths that are not used. However, it increases the compiler complexity, as the compiler must find compatible mutually exclusive code paths to map onto the same hardware. It also increases demand for configuration memory. The next few chapters will propose ways to allow the compiler to exploit the over-provisioned configuration memory in high throughput applications to enable this type of sharing.

5.2 Representing Predicate Relationships - The CDT

In order for SPR to manage sharing of mutually exclusive operations, it requires a way of representing the predicate relationships that define the mutual exclusion. In this work,

the extra information needed to determine the compatibility of operations for sharing purposes is provided in the form of a Control Dependence Graph (CDG). More specifically, there are several places where a tree structure is assumed, as opposed to a more general directed acyclic graph. Restricting the graph to a tree structure allows one to express the same hierarchical control structures as the predicate regions from [FO83], with sub regions constructed as children of enclosing regions. This representation corresponds to predicate dependencies in code structured entirely out of nested blocks.

This dissertation will refer to the structure as the Control Dependence Tree (CDT) to indicate that the structure is limited to a tree. The Macah front end used in this work directly tracks and creates the CDT defined by the structured code when performing enhanced loop flattening on the kernels. This is the representation used by SPR.

The work of [FO83] was later extended to a more general form, which was presented as a Program Dependence Graph [FOW87], combining control and data dependencies for arbitrary programs. Further extension to incorporate single-static assignment was presented in [OBM90], where it is called the Program Dependence Web. Efficient ways of representing the control dependencies in this more general framework is presented in [CFS90]. In [MLC⁺92], the authors used a similar graph structure for tracking the predicates in hyper-blocks and called it the Predicate Hierarchy Graph. General systems have been created for extracting and answering queries about the predicate structure of code [JS96, SHA00]. With the exception of [FO83], all of this previous work was done using a more general DAG structure. In this work, the use of a tree structure does not significantly limit available sharing, as described in Section 5.2.3, and adds opportunities for optimizing the algorithms, so it is more appropriate than the more general DAG structures.

5.2.1 CDT Structure

In a CDT, operations are associated with the run-time enable signal that indicates whether or not that operation should execute in this iteration – the predicating signal. This is a one-hot representation, where operations in the *then* portion of an *if-then-else* structure are controlled by one signal, and the *else* portion is controlled by a second signal that is

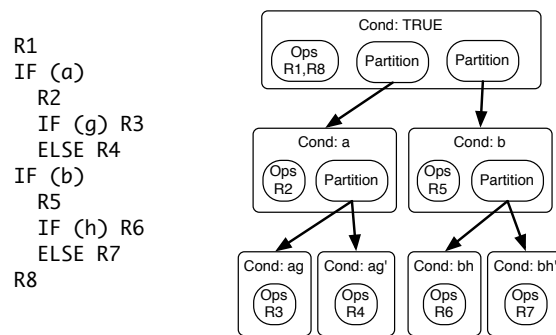


Figure 5.2: A simple example of a Control Dependence Tree for sequenced and nested IF statements.

never asserted at the same time as the first. However, they may both be un-asserted for an iteration where the entire *if-then-else* structure is not reached. This one-hot representation is beneficial in spatial architectures – especially with a multi-way branch such as a *case* structure with no fall through – because it allows the predicates to be spread and computed close to the locations where they are needed.

An example CDT along with the code structure it comes from is given in Figure 5.2. Each of the predicate signals that enable a control-flow block correspond to a condition node in the CDT. In Figure 5.2, condition nodes are drawn with rectangular sides. Within a condition node, there will be operations that execute “under” that predicate – executing only when that predicate is true. This is represented in the CDT by having *operation nodes* in the CDT that are children of the predicate node they execute under – the nodes with circular sides and marked with Ops in Figure 5.2. There will also be further partitions, for example a nested *if-then-else*. These are partition nodes, shown as the circular-sided nodes in Figure 5.2 labeled Partition. The *then* and *else* conditions of a nested *if-then-else* will each represent a nested predicate signal – and so each will have a condition node in the tree that is mutually exclusive with the other’s. The condition nodes labeled ag and ag’ are an example of this. A single condition may guard the execution of several *if-then-else* partitions, representing sequenced nested structures, such as the two *if* statements in Figure 5.2. Instead of being executed under a particular condition, they are always executed, so their partition nodes are in a special condition node labeled TRUE. The

children condition nodes of separate partition nodes are not guaranteed to be mutually exclusive with each other.

This node structure gives rise to depths in the CDT with alternating types, condition node levels and operation/partition node levels. The root of the tree is the “unconditional” condition node, represented as TRUE. This is a condition level. The next level down will have operations that run on every iteration of the kernel, and the top level partitions of the kernel. This is an operation/partition level. Each partition will have a set of children condition nodes, yielding the next condition level, and so on. In Figure 5.2, the operation/partition node level is drawn within each parent condition node level. This fundamental two level structure was present in the graphs of [FO83], with partition nodes represented by the adjacency of related sub regions, and the condition level represented by the enclosure of region boxes. Through similar mappings, the same two level structure can be demonstrated in the work of [FOW87, CFS90, MLC⁺92, JS96].

The partition nodes are the source of the mutual exclusion information used for resource sharing. They represent the nesting of code blocks guarded by mutually exclusive predicates. A child and all the descendants of that child will be mutually exclusive with the other children/descendants from the same partition node, as only one of the corresponding code paths will be chosen at run-time. As shown in [JS96], determining that two operations, X, Y are mutually exclusive can be done by finding a partition node P , such that X and Y are descendants of P 's children condition nodes C_a and C_b , respectively, such that $C_a \neq C_b$. For example, in Figure 5.2, operations from R3 and R4 are mutually exclusive with each other because they are children of different condition nodes, ag and ag' , who's common ancestry begins at the partition node under shown within the condition node a . Pairs of operations that execute and signals that communicate under mutually exclusive condition nodes in the CDT are themselves mutually exclusive, and will sometimes be referred to as being compatible because they may share resources in the architecture without fear of collision at run-time.

With the CDT limited to a tree structure, determining whether such a partition exists is reduced to finding the least common ancestor of X and Y and determining if it is a partition node or a condition node. To see this, consider node L as the least common

ancestor of X and Y . This means that the paths from X and Y to the root of the tree are disjoint from X and Y up to the children of L , and are equivalent from L on to the root. If L is a partition node, then clearly X and Y are mutually exclusive because they share the partition L as an ancestor, but they are descendants of disjoint paths below L , and must be descendants of different condition nodes, C_1 and C_2 . If L is a condition node, then X and Y are descendants of the same condition nodes from L up to the root, so there can be no common partition node above L with differing children condition nodes. Additionally, below L , X and Y will be descendants of disjoint children nodes, but they will also be descendants of disjoint partition nodes, so there can be no common partition node with differing children condition nodes below L , so there is no partition for X and Y . Without some alternate semantic analysis, we must conclude that X and Y may not be mutually exclusive at run-time. For example, in Figure 5.2, the least common ancestor of $R3$ and $R4$ is a partition node, so they are mutually exclusive. However, the least common ancestor of $R3$ and $R5$ is the condition node at the root of the tree, so they are not mutually exclusive.

5.2.2 *Altering the CDT*

If an operation's execution is to be dynamically controlled at run-time, the signal that controls the execution must be communicated to the hardware performing the execution. In a large architecture, this communication may be expensive and limited. Due to spatial locality, some controlling signals may be more readily available than others. For these reasons, it is useful to be able to alter the conditions that an operation executes under, but it must be done in a way that preserves correctness.

If an operation's controlling signal is not available, one solution is to execute the operation unconditionally. This will maintain correct execution, but prevent predicate aware sharing with other operations. This is reasonable because it corresponds to the execution model supported by a non-predicate aware version of SPR. This method of operation promotion was first presented in [TLS90] as eager execution through predicate lifting, and later used in [MLC⁺92] where it was referred to as promotion. It is referred to as promotion here to reflect the movement of operations towards the root of the CDT. All operations promoted to unconditional execution are executed speculatively, and the required result

is chosen from the speculative results to be passed forward in the computation. Any operations with external side effects, such as memory writes, must be guarded by the control predicate that indicates whether or not the control path of that operation is being followed on this iteration.

This promotion can be generalized to partial promotion as long as there is a method of guarding the execution of operations with side-effects. An operation can be promoted in the CDT to any condition node on the path from where it currently resides to the root node. This is because ancestors in the CDT represent a superset of the dynamic execution conditions for operations. For example, in Figure 5.2, The code block guarded by *a* will be executed in any loop iteration where the code block guarded by *g* executes, and possibly more. In [TLS90] this is referred to as lifting, and can be done from any predicate to one that dominates it. Any time the operation would have originally executed, it still will, though it may execute in more instances as well. If R3 is promoted to condition *a* in the CDT, this is equivalent to moving the R3 code up next to the R2 code under the *if* statement guarded by *a*. As long as any side effects are prevented when *g* is not true, this is a safe transformation. Note that this dissertation deals with dataflow graphs that were constructed for unconditional spatial execution, so all operations with side-effects already have a predicate input to ensure they only execute when it is appropriate.

This generalized promotion will reduce opportunities for sharing, but not necessarily down to no sharing as in the case of promoting to unconditional execution. When an operation is promoted, it will rise above some partition nodes. Unfortunately, SPR can no longer use any sharing that was facilitated by those partition nodes. For example, in Figure 5.2, if R3 is promoted to the operation node under *a*, it can no longer share with R4 because their least common ancestor is now the condition node *a*.

The flexibility of promoting operations, and the resulting elimination of the dependence between a predicate and the promoted operation, can be exploited by SPR in several ways. It can be used to reduce the critical recurrence Π during scheduling. It can be used during placement to reduce the need to communicate a predicate across an architecture. It can be used during routing to share routes even when the appropriate predicate signals are not locally available. In general, it gives the compiler the ability to make trade-offs

between the resources to route the predicate signals across the architecture and the resources needed to speculatively execute operations when the predicates are not available for sharing.

5.2.3 *Tree Restrictions*

Restricting the representation of condition dependencies to a tree structure does limit the relationships that can be expressed. The directed-acyclic graph (DAG) representation generalization allows re-convergence of paths from the root to a descendant node, whereas a tree does not. In a CDT, the run-time condition an operation runs under is described by the logical and of all of the conditions from the operation's parent condition node to the root. Allowing re-convergence in this graph represents the logical or of the conditions along the converging paths. This means that the operations will execute in the union of these conditions.

According to [FOW87], the hierarchical tree representation of [FO83] is limited to "structured" programs, those "built of blocks, loops, and conditionals, each of which is single entry and single exit." In this section, I will examine what this limitation implies in the context of sharing resources in CGRAs. I will do this by examining the sharing available in a full DAG-based CDG, and showing the loss of sharing opportunity when converting the DAG to a tree while maintaining program correctness.

Simplified example program codes requiring a DAG-based CDG are shown in Figure 5.3(a) and Figure 5.4(a). Blocks of statements are abstracted to regions labeled R1 through R7. Predicate conditions are represented by lower case letters a, b, g, and h. The conditions for alternate branches of a partition are denoted with an apostrophe ('), and logical AND and OR operations are denoted in the CDT with multiply/add notation. Jump target labels are given by X and Y. In the example, GOTO statements are used to create the control flow that will require a full DAG.

These two figures represent the two possibilities for having re-convergent paths in a CDG. In Figure 5.3(a), the paths diverge going from a single condition node to separate partition nodes. In Figure 5.4(a), the paths diverge from a single partition node to separate condition nodes. In both examples, the paths re-converge at a condition node. This is the

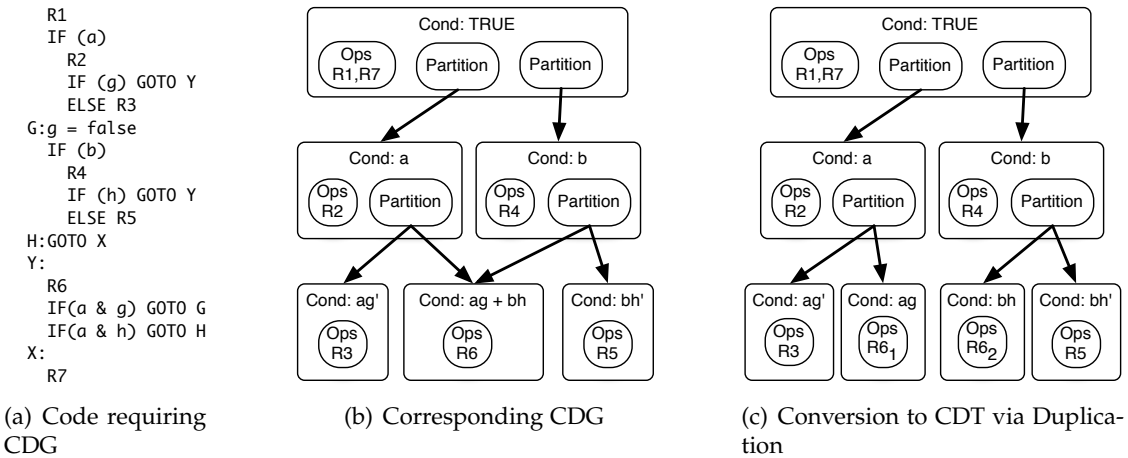


Figure 5.3: Example of unstructured code requiring a full DAG CDG with divergence in a condition node.

only reconvergence possible, as it corresponds to regions that are executed under the union of run-time execution conditions. The reconvergent CDGs are shown in Figure 5.3(b) and Figure 5.4(b).

In a reconvergent CDG, determining what operations may share resources is no longer a simple matter of determining if their least common ancestor is a partition node or not. This can be seen by examining the R6 region. When the paths diverge at a condition node, the code in region R6 will not be mutually exclusive with any other region. It is possible that conditions a, g, b and h are all true in the same iteration, so even though the LCA of R6 and R5 is a partition node, they may execute in the same iteration. The same applies to R6 and R3. In fact, this is a problematic construction for standard spatial speculative execution because R6 may need to be executed twice in a single iteration. The pipelined modulo-scheduled execution assumes each operation executes only once per iteration, so the region would need to be inlined, effectively duplicating it for each of the GOTO statements.

The CDG that results from this duplication is shown in Figure 5.3(c). The modifications required to allow for pipelined modulo-scheduling will likely split these reconvergences and eliminate them from the CDG, turning it into a tree. In fact, after duplication, R6₁ is mutually exclusive with R3 and R6₂ is mutually exclusive with R5, allowing the

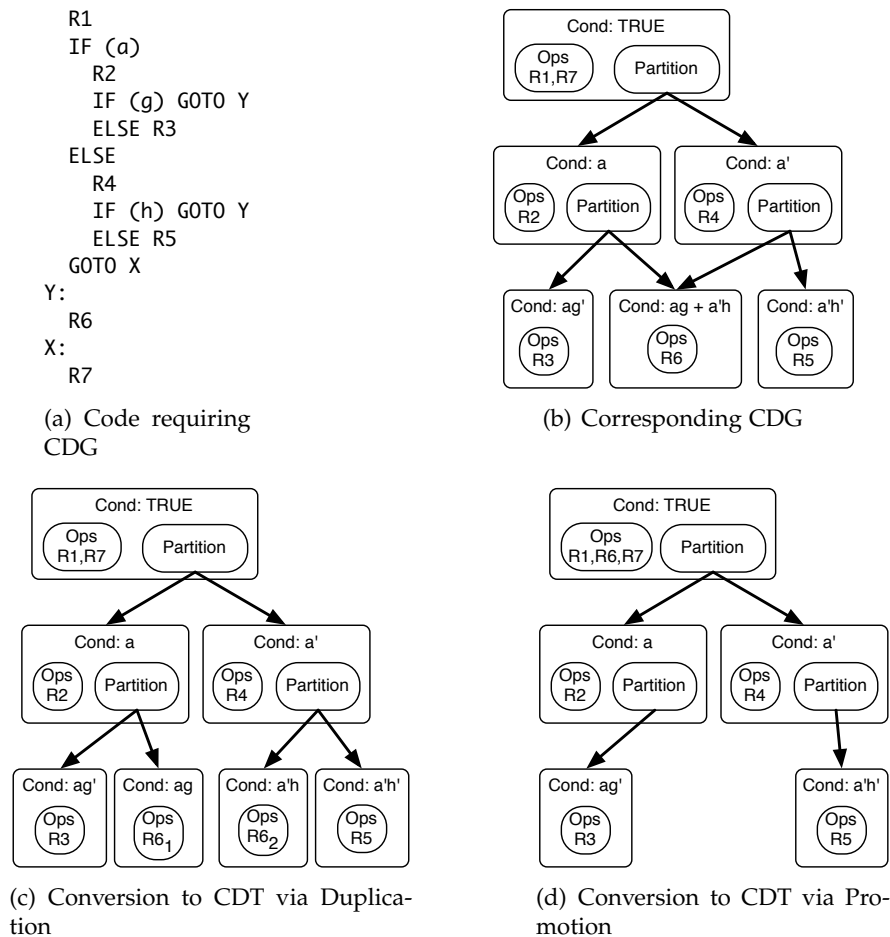


Figure 5.4: Example of unstructured code requiring a full DAG CDG with divergence in a partition node.

respective pairs to share resources. If they are not split, simply checking the type of the LCA no longer determines if two conditions are mutually exclusive, as the previous example illustrates. Instead, the set of all common ancestors that act as divergence points on paths to conditions determine what is mutually exclusive. In order to guarantee only one of the conditions is true in an iteration, all of the divergence points must be partition nodes to ensure that only one of the paths below each is true in a given iteration. In the case shown in Figure 5.4(b), one of the divergent nodes is a condition node, eliminating the possibility of sharing $R6$ with either $R3$ or $R5$. Thus, we can be sure we are not giving up any opportunity for sharing by restricting our sharing algorithms to trees. Whenever

one of the divergence nodes on a set of reconvergent paths is a condition node, sharing is not possible even in the full CDG representation, so the CDT is not losing any sharing opportunity.

When the paths diverge at a partition node, as in Figure 5.4(b), sharing becomes possible. The method of determining mutual exclusion in [MLC⁺92] can be used in these more general graphs to find possible sharing. Mutual exclusion is indicated by constructing an unsatisfiable Boolean equation by ANDing together the predicate Boolean expressions for two operations. The Boolean expression for a particular operation is formed by ANDing together all conditions on the path from the root to the operation. For multiple paths, each path expression is ORed together. For example, in the CDG given in Figure 5.4(b), R6 is mutually exclusive with R5, where the corresponding equation is $((a \wedge g) \vee (a' \wedge h)) \wedge (a' \wedge h')$ and is unsatisfiable.

For this example, R6 is mutually exclusive with R3 and R5, but not R1, R2, R4, or R7. Additionally, R6 can be promoted up to the TRUE condition, but not to the *a* condition or the *a'* condition. These relationships are important to resource sharing and scheduling, placement and routing. If it were possible to retain these relationships in a tree structure with no other drawbacks, then the restriction to a tree structure would not pose a significant limitation. To the best of my knowledge, there is no such tree structure, so instead I present two alternative ways of turning a general DAG of this form into a tree, along with the relative drawbacks.

The first solution is to duplicate the subtree below the reconvergence as before, shown in Figure 5.4(c). This retains the mutually exclusive relationships, and preserves the most potential for sharing. It increases the promotion flexibility because each duplicate of R6 can be promoted to the corresponding *a* or *a'* condition, as well as all the way to the unconditional case. The drawback to this method is extra resource usage if the opportunities for sharing cannot be exploited.

The second solution is to preemptively promote R6 up to the divergence point, as shown in Figure 5.4(d). This eliminates all sharing below the promotion, but avoids the extra resource usage from duplication of R6.

Characterization of the practical trade-offs between these two options is beyond the scope of this work. To exploit the algorithmic simplifications of a tree representation, this work will only consider structured programs where the CDG is limited to a tree. Algorithms for handling general DAGs or choosing the appropriate tree simplifications are left as future work.

5.2.4 Integrating CDT and Modulo Scheduled Pipelining

The CDT indicates which operations may share resources within the same iteration. However, modulo scheduling imposes more constraints on what may share and adds more flexibility at the same time. The insights described in this section are important to designing appropriate algorithms for using the CDT to share resources in a modulo-scheduled system.

Modulo Schedule Phase and the CDT

The CDT specifies the run-time control signals generated in the datapath that predicate each operation. Adding a schedule further constrains when the operation may execute. In a modulo-scheduled system, this can be viewed as adding the phase of the counter in as a new root partition of the CDT. This is because each node only executes when the original condition is true AND the phase counter matches the scheduled phase for the node. This is illustrated in Figure 5.5.

It is this top level partitioning that allows the time-multiplexed resource sharing in a modulo-scheduled system. Because the phase condition is the top level condition and it fully partitions all operations, the only test needed to see if two operations are mutually exclusive based on the phase partition is to check that their phases are not equal, effectively binning the operations by phase. This is why the modulo-graph unrolling formulation works so well for sharing physical resources. In fact, it is such a simple test relative to the more general case of checking mutual exclusion in the CDT that it is probably worth keeping the modulo-graph unrolling. In the future, SPR could potentially just treat the phase as any other condition to be used in the predicate aware framework. The potential of the latter idea is briefly explored at the end of Chapter 10.

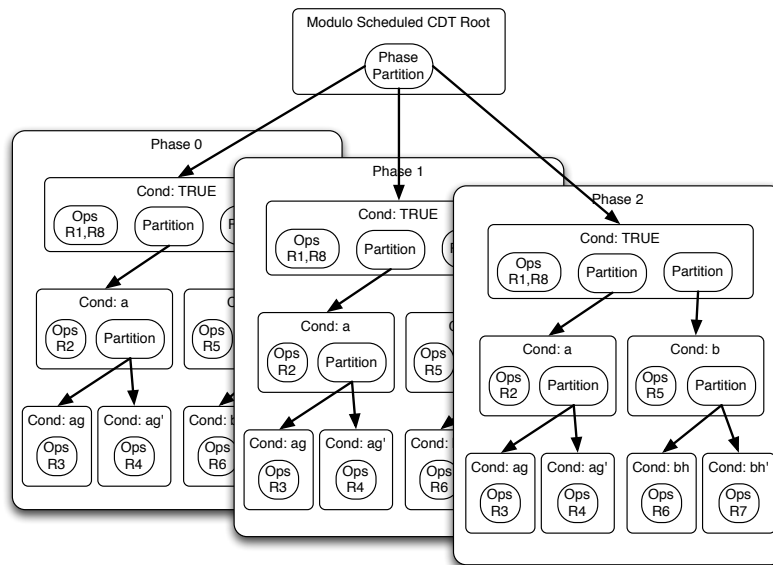


Figure 5.5: Combining time into the Control Dependence Tree

The illustration of Figure 5.5 assumes that the schedule length is the same as the II. A schedule that is longer than the II leads to further duplication of the CDT within each phase, once for each start time that aliases to that phase. This duplication is not a partition, and operations from these duplicates cannot share except in special circumstances. This is explained in more detail along with an example in the next section.

Modulo Variable Expansion for Predicates

Modulo-Scheduled pipelining increases parallelism by pipelining executions of the kernel loop body. As a result, what was originally a single variable in the kernel will have multiple live copies, up to one per loop iteration in the pipeline. This is known as modulo-variable expansion [Lam88]. Conditions in the CDT represent the currently live control flow blocks. Just like variables, there is conceptually a separate live CDT associated with each loop iteration in the pipeline, and each condition can take on different values for each live iteration.

This is illustrated in the following example, where the CDT and schedule are shown in Figure 5.6 and two possibilities for sharing that are affected by the modulo-variable

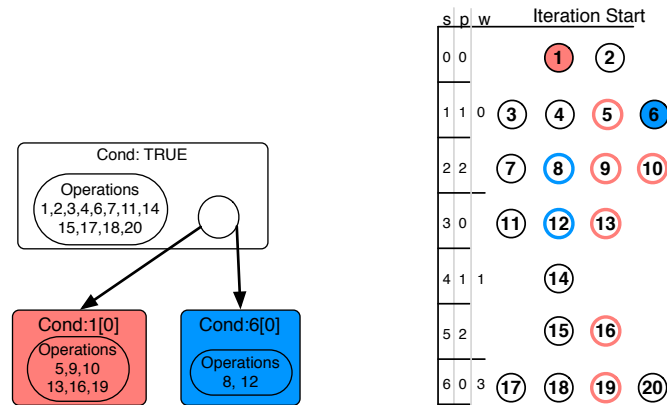


Figure 5.6: Base CDT and example schedule for a kernel iteration.

expansion are examined in Figure 5.7. First, consider the CDT shown in Figure 5.6(a), and the associated scheduled loop iteration in Figure 5.6(b). The CDT has an unconditional node at the root in white, with a list of operations that execute unconditionally. Below that there is a partition into conditions 1[0] ■ and 6[0] ■. The conditions are named by an operation number, a delay in brackets, and a color marker. The operation number is for the operation that generates the predicate representing the condition. The delay is the number of iteration initiations between the loop body execution where the predicate is generated and the loop body execution where the value is used. Often, this is zero, when the test immediately guards the control flow block. Sometimes the value is stored for several iterations before being used as the test in a control flow block. This latter case will not be considered until the next section. The color marker is used to associate conditions with operations in the schedule for easy reference.

The schedule on the right represents operations as numbered circles, and the schedule can be seen on the scale, where *s* denotes start time, *p* denotes phase, and *w* indicates wave. This example is a schedule of a loop with 20 operations with an II of 3. A multiplicity of operations in a row means that they are scheduled at the same time, and the columns have no particular meaning. The predicate-generating operations are shown as filled circles of the appropriate color in the schedule, and the circle of any conditionally executed operations are colored according to the condition under which it executes. Since the length of this schedule is longer than the II, there will be multiple iterations of the

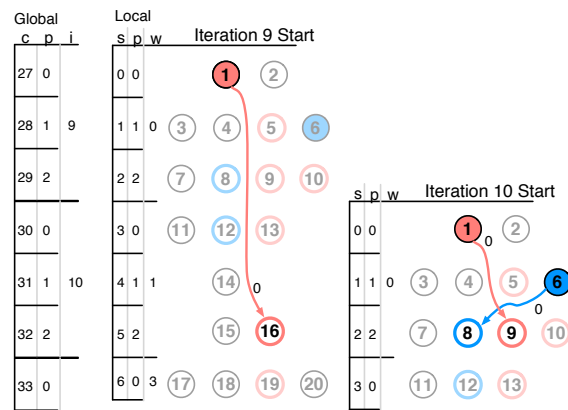


Figure 5.7: Even though red and blue operations can share *within* an iteration, they cannot share *across* iterations. Operation 8 may share with 9, but not 16.

loop live at the same time in different stages of completion. The number of waves in the schedule tells us the number of iterations that will be live at any given phase, here three live iterations for phase 0 and two for phases 1 and 2.

To show how the modulo-variable expansion of predicates affects sharing, two of the iterations are shown in Figure 5.7. The local time scales are next to each set of operations, with the global time scale on the left illustrating that this is an excerpt from some time into the computation. In the global timescale, c represents total cycles from the beginning of the computation, p is the phase, and i is the count of iterations that have been initiated. Iterations 7, 8, and 11 will also have operations live during the time period shown, but they are left out of the diagram for clarity. Operation 16 from iteration 9 and operations 8 and 9 from iteration 10 are highlighted, along with the operations that generate the predicates for them. The dependencies from predicate generating operations to operations 16, 8 and 9 are shown, along with the iteration delays on that dependence. The iteration delays are all zero, which means the operations depend on the predicates from the same iteration.

In this example, if operation 1 from iteration 9, denoted $Op1_9$, generated a value of true, then $Op16_9$ should execute. However, there are at least two live copies of the result of operation 1 available when $Op16_9$ will execute – the result of $Op1_9$ and the result of $Op1_{10}$. The compiler must ensure that the proper live value controls the execution of

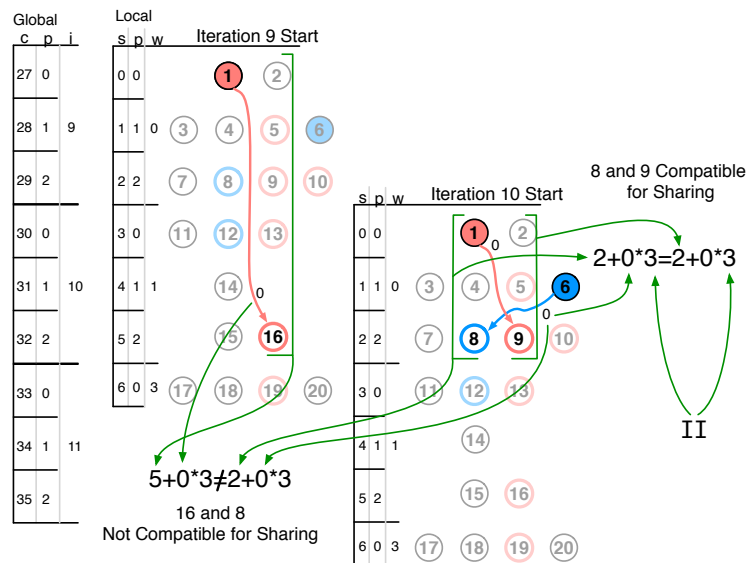


Figure 5.8: Calculations showing where sharing can and cannot happen.

Op_{16_9} , which is Op_{1_9} . The CDT in Figure 5.6 indicates that the predicates generated by operations 1 and 6 will never be true in the same iteration, however it says nothing about their relationship across iterations. This means that $Op_{1_{10}}$ and $Op_{6_{10}}$ cannot both be true, and so operations 8 and 9 can share the same resources. However, Op_{1_9} and $Op_{6_{10}}$ may both be true, which means both Op_{16_9} and $Op_{8_{10}}$ would execute in the same cycle, so operations 16 and 8 cannot share resources.

To accurately determine which operations may or may not share resources in a modulo scheduled system, the compiler must take into account the modulo-expansion of predicates in addition to the relationships of those predicates in the CDT. This calculation must be done carefully, taking into account the relative offsets in local schedules between pipelined iterations. This can be done by considering the local predicate use time relative to the iteration when it was generated. This will be called the predicate's age. For the example in Figure 5.6(b), operation 16 uses the predicate generated by operation 1 at an age of 5, and operation 8 uses the predicate generated by 6 at an age of 2. To put it mathematically, if the start time of the dependent operation relative to that operation's iteration start is s_{op} , the current scheduled initiation interval is II , and the number of

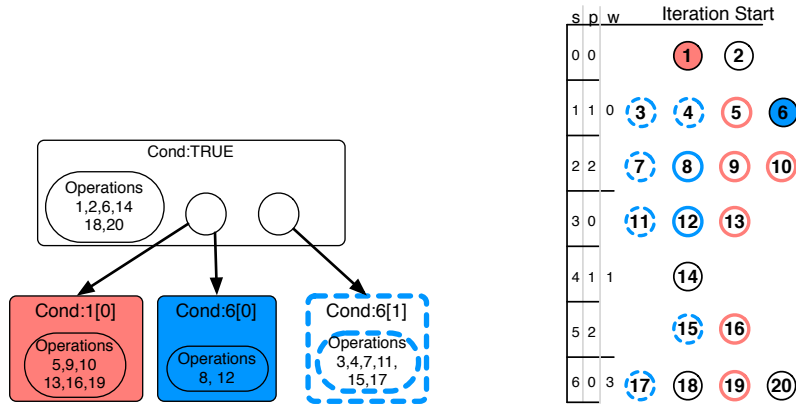


Figure 5.9: Base CDT and example schedule for cross-iteration sharing.

iteration delays between the predicate definition and use is id , then the predicate age, a_p , is given by Equation (5.1).

$$a_p = s_{op} + II * id \quad (5.1)$$

When two operations have the same predicate age, then their conditions can be compared in the CDT for mutual-exclusivity. To simplify the implementation, all of the work here only shares when id and s_{op} are identical between the conditions of the two operations being shared. The results of applying Equation (5.1) to the example in Figure 5.6(b) are shown in Figure 5.8. The predicate ages of Op_{16_9} and $Op_{8_{10}}$ do not match, so they cannot share resources. The ages of $Op_{8_{10}}$ and $Op_{9_{10}}$ match and they are executed under mutually exclusive conditions in the CDT, so they may share resources.

Cross-Iteration Sharing

Due to the overlapped execution of iterations in a pipelined modulo schedule, sharing is also possible with operations from different iterations that will execute in the same cycle with the right condition relationships. This is called cross-iteration sharing. However, the relationships that enable cross-iteration sharing are not immediately obvious from the structure of the CDT, and require the math of Equation 5.1 plus some reasoning about the operations generating the predicates. An extension of the example from Figure 5.6 used to demonstrate cross-iteration sharing is shown in Figure 5.9. Here, another condition has

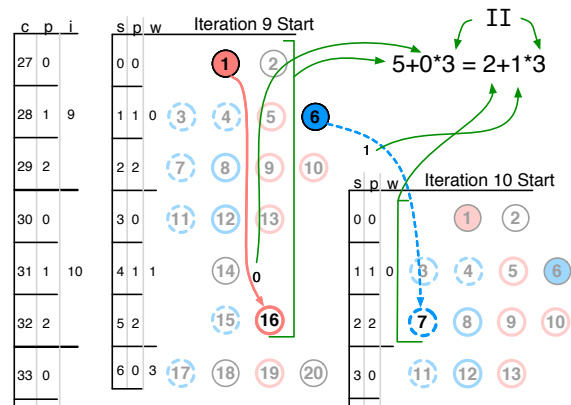


Figure 5.10: Calculations showing where cross-iteration sharing can happen.

been added to the CDT that uses the predicate from condition 6[0] ■, but delayed by one iteration. This new condition, condition 6[1] □ is represented by a blue dashed outline. Some of the operations in the schedule have also been marked with this condition.

The key to enabling this sharing is observing that condition 6[0] ■ and condition 6[1] □ use the same predicate values, simply at different times. Since the predicates generated by operation 6 are never true in the same iteration as those generated by operation 1, it is possible to share anything that depends on condition 6[1] □ from a given iteration with anything that depends on condition 1[0] ■ from the previous generation. An example with the math of Equation 5.1 is shown in Figure 5.10.

For a more intuitive idea of when cross-iteration sharing is possible, it is helpful to understand where iteration delays come from in source code. Essentially, they are a representation of loop-carried dependencies – the use of a variable in an iteration that was defined in a prior iteration of a loop. The number of iterations between the definition and the use is the iteration delay. Example code is given in Figure 5.11, where blocks X, Y and Z generate operations under conditions 6[1] □, 1[0] ■, and 6[0] ■, respectively. The `if` guarding Y comes after the update

```

a = false;
for(i=0;i<100;i++){
  if(a)
    X
  a = (i%2 == 1)
  if(!a)
    Y
  else
    Z
}

```

Figure 5.11: A loop with cross-iteration sharing.

of a , so it uses the value from the same iteration. However, the `if` guarding X will use the value of a that is stored from the previous iteration, leading to a loop-carried dependence and a corresponding iteration delay of 1. In this example, the update of a is operation 6 from the previous example. For a loop executing a digital-signal processing filter, an iteration delay is equivalent to a delay element. For recursive filters, it will be common to have operations dependent on values across different iteration delays.

The final results of combining the CDT and schedule of Figure 5.9 are shown in the top of Figure 5.12. This includes the duplication of the CDT per start time, annotated with `@[start time]`. This full expansion is quite complicated, and aside from this example, the CDT combined with scheduling information will be presented in the simplified form used in Figure 5.5. Below the expanded CDT, enough iterations of the loop are shown to highlight the steady state modulo-schedule. The work in this dissertation will take advantage of the within-iteration sharing – operation 12 with 13 and operation 8 with 9 or 10. It is left for future work to investigate cross iteration sharing – operation 16 with 7 and 11 with 19.

5.2.5 CDT Pruning and Verification

As currently implemented, SPR runs two pre-passes on the CDT to clean it up. First, it removes any nodes that do not provide partitioning. This is common, for example, where there is an `if` structure with no `else`. An example of this is shown in Figure 5.13 where the simple example CDT from Figure 5.2 has been pruned of the partition nodes that do not provide any mutual-exclusion. The `a` and `b` conditions come from sequenced `if` statements without a corresponding `else`, so they are pruned and their operations promoted to their parent, in this case `TRUE`.

Second, because operations with side effects may be promoted, we need to ensure the predicate signals are always valid and are always computed unconditionally. The Macah front end currently ensures this, but the SPR tool will double check that any operation providing a CDT predicate executes unconditionally. In future work, it may be possible to examine the operations that generate the predicates and have SPR automatically promote

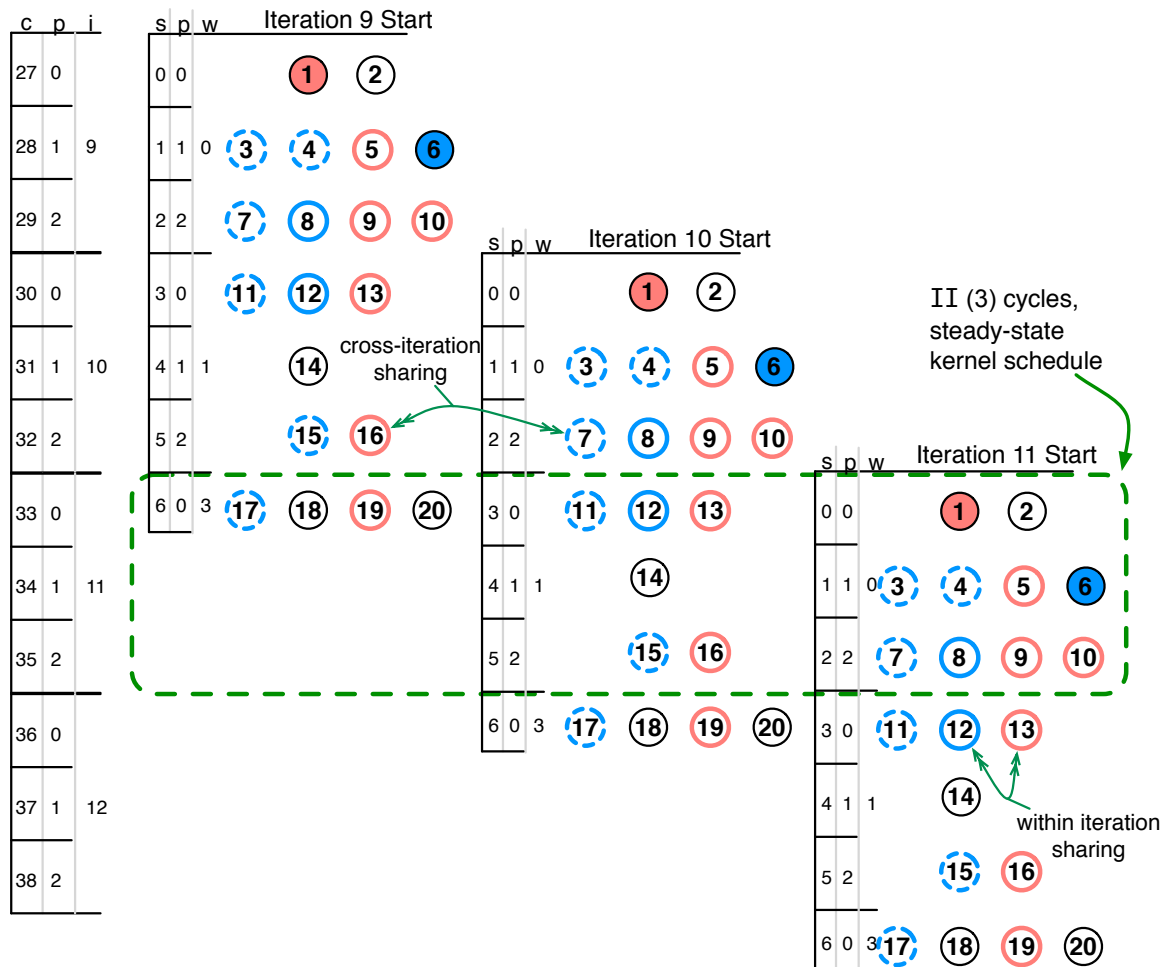
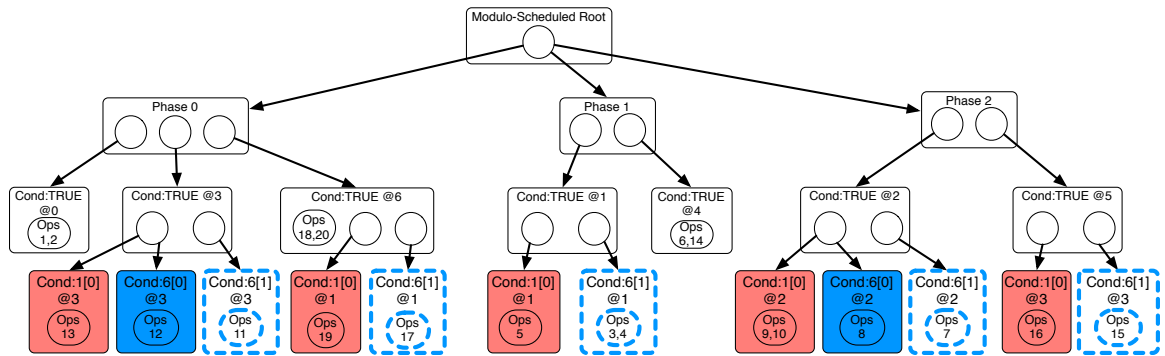


Figure 5.12: The final fully expanded scheduled CDT along with the steady-state kernel schedule.

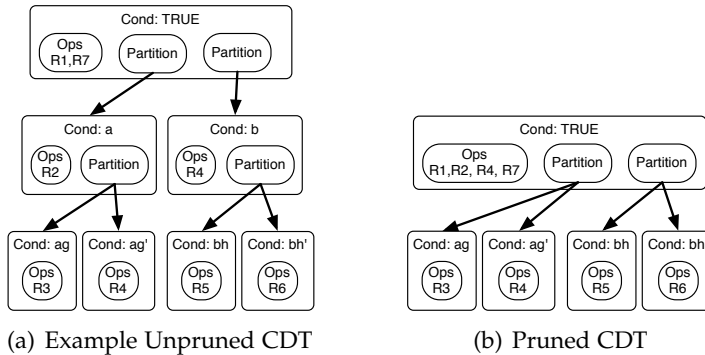


Figure 5.13: Example of pruning partition nodes that do not provide any mutual-exclusion.

the necessary operations to run unconditionally, but for now it simply generates an error message when this situation is encountered.

SPR also assumes that the predicates in the architecture are absolute instead of relative. This means that predicates represent the result of ANDing together all conditions in the full path from the root to the operation, instead of just the last condition. This is needed because we assume operations are mutually exclusive if there is some partition between them, at any depth, and so the actual expression representing that partition must be ANDed in at run-time to ensure the proper control signaling for execution.

5.3 Representing Predicates

The CDT provides a method for reasoning about the conditions under which operations and communication will happen at run-time. However, a quick and efficient method of answering the simpler question of whether two operations can share resources at run-time is valuable for use in the inner loops of scheduling, placement and routing.

If multiple operations are attempting to share a resource, the execution of each operation must be mutually exclusive with all other operations using that resource. Often, the algorithms will be attempting to add a single operation to a group of other operations that are already sharing a resource. The most straightforward approach to this would be to check the operation to be added with each of the existing operations to see if a conflict exists. However, a more efficient approach is to create a union representation of the op-

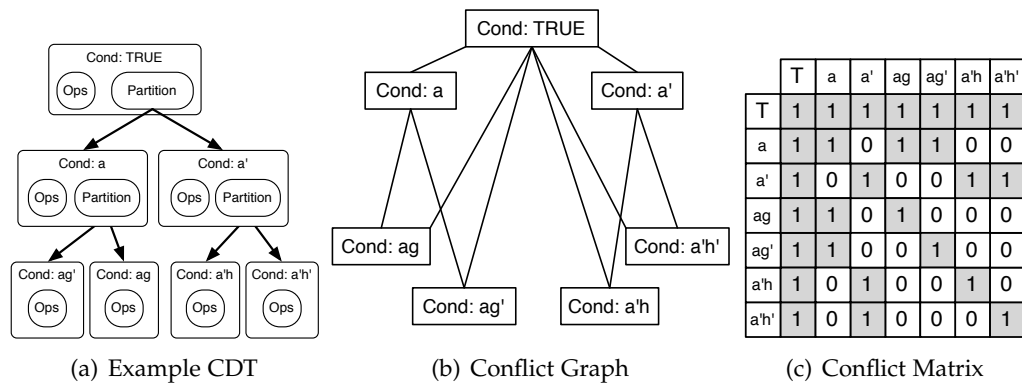


Figure 5.14: Example of a CDT and the conflict relation it embodies.

erations already sharing the resource, check once against the union representation, and then if the new operation can be added, include it in the union for future checks.

The key element in the efficient approach is a conflict representation that has an efficient union operation. The work presented here uses a conflict bit vector, constructed as follows. If any two nodes in the CDT are not mutually exclusive (that is they are not separated by a common ancestor partition node) then we say they conflict. The conflict graph can be generated by creating a vertex for each condition in the CDT and adding edges between all pairs of conflicting conditions. This graph can be represented as a binary matrix, where there is a row and column for each vertex, and a 1 represents an edge between the row vertex and the column vertex. The conflict relation is reflexive, so the edges are undirected, yielding a symmetric matrix. An example of a CDT and the corresponding conflict graph and matrix are shown in Figure 5.14. Each row of this matrix can be considered the conflict bit-vector for the condition corresponding to that row. This representation is similar to the 1dnf form used in [JS96]. This representation is useful because it is easy to test for conflicts and it is easy to represent the conflicts from multiple operations mapped to a single resource. To test for a conflict between two conditions, the compiler simply has to check if the bit of one condition is set in the conflict vector of the other.

When multiple operations are mapped to a single resource, the bit vectors can be ORed together to obtain the union of conflicts. When attempting to add a new operation,

the bit corresponding to that operation's condition can be checked against the union to determine if it conflicts with any of the operations already assigned to that resource. Note that removal of a conflict vector from the union is generally not possible, so when a condition is removed, the union must be regenerated from the bit vectors of the remaining conditions.

Once scheduling has been completed, the modulo variable expansion of all the predicates also needs to be taken into account. This is accomplished by annotating each predicate with the start time of their use relative to the iteration where the predicate is generated. While it is possible for two operations to be compatible if they are scheduled in different waves and they depend on the same predicate with different iteration delays, as described in Section 5.2.4, supporting this is left as future work. For this implementation, the predicates are simply checked for differing delay annotations that would make them incompatible. We will call this logical union of the existing sharing constraints coupled with scheduling delays the *aggregate execution condition*. An example is illustrated in Figure 5.15. The rows from the conflict matrix of Figure 5.14(c) for conditions a and a'h' are annotated with delay information to represent the individual conditions. The conflict bit for a is checked in the vector for a'h', where a 0 represents no conflict. Similarly, the conflict bit for a'h' in the vector for a is 0 by the reflexivity of the conflict relation. Finally, if the delays that come from scheduling match, the conditions are compatible and can be combined into an aggregate execution condition, which is represented as the list of [a,a'h'], the logical OR of the conflict bit-vectors, and the common delay between them. Here we can see that the only condition that could be added later would be a'h with a delay of 3. The conflict relation is reflexive, which means both directions do not need to be checked for conflicts. With single conditions, one condition can be arbitrarily chosen to check against the other. When checking a single condition against an aggregate, the bit position corresponding to the single condition can be checked in the aggregate conflict vector or each condition bit from the list of conditions in the aggregate should be checked against the single condition. Similarly, when checking for conflicts between two aggregate execution conditions, checking the bit positions of the aggregate with the smaller list of conditions will minimize the number of checks. In practice, if the conflict vector can fit

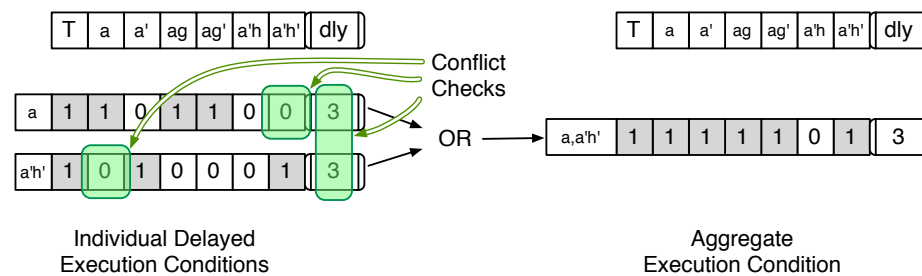


Figure 5.15: Illustration of an aggregate execution condition. Two conflict vectors plus their delay are checked for conflicts then combined into an aggregate execution condition.

in a machine word, the entire check can be made through a single AND operation, so the direction of the check does not matter.

This representation is efficient for testing conflicts and adding to a set of operations sharing a resource. If an operation is removed from the set sharing a resource, removing the associated conflicts from the aggregate execution condition requires more work. The aggregate execution condition will track all of the conditions that are added to it as a set. A condition is removed from the aggregate by removing it from the set and re-creating the conflict bit-vector by ORing together the conflict bit-vectors of the remaining conditions in the set.

This imbalance between checking for conflicts and removing a condition from the aggregate execution condition is an optimization for the common case of the algorithms that will be using this aggregate representation. The Simulated Annealing and Quick-Route algorithms both require checking for conflicts to estimate costs before committing the changes that those checks are done for. In both cases, more checks will be made than actual additions of conditions to the aggregate execution condition. This also holds for removals, since they are bounded by the number of additions made. Thus, having the aggregate execution condition support fast checking at the expense of removals is appropriate for the algorithms that use it.

5.4 Hardware Support for Mutual Exclusion

The final background element needed for predicate-aware resource sharing before moving on to the algorithms is a reasonable hardware model. This section will introduce some

of the design concerns and options, but it is not meant to be a thorough exploration and quantification of architectural choices. Instead, it is meant to plant ideas and provides a reasonable abstraction that predicate-aware algorithms can target.

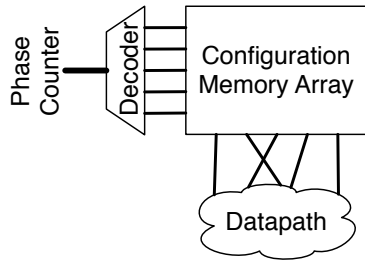


Figure 5.16: A basic configuration hardware design.

A straightforward implementation of the configuration storage for these time-multiplexed architectures is shown in Figure 5.16. In this diagram, the phase counter is provided to a decoder for the memory array. This selects the word-line to be read from a memory array. The memory array stores the actual configuration bits. After being read out of the array, the bits are split up and distributed out to configure the reconfigurable datapath.

5.4.1 Regions of Control

In a large array, the configuration may be divided up across several such memory structures for scalability. The modulo-counter based configuration selection can be accomplished through similar duplication, with each memory structure maintaining its own

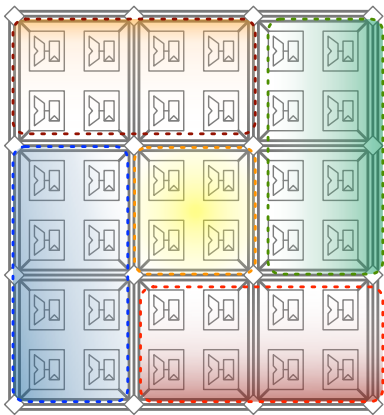


Figure 5.17: CGRA colored by configuration regions.

local copy of the counter. As long as these distributed copies run off synchronized clocks and are reset to the same values at some point before the program starts executing, such as a global reset, the counters will remain synchronized throughout the execution of the program. This enables scalability, since the counter can be replicated as needed and does not depend on a central value that must be broadcast across the chip.

There is no central value that must be broadcast, so the compiler should not assume there is

a central modification point that will alter loaded configuration. Each configuration memory will be in control of the configuration for a certain subset of computational resources. This motivates the first new hardware abstraction needed in a predicate-aware SPR. Each of these areas will be called a *configuration region*, and are illustrated in Figure 5.17.

It is important for the compiler to know about these regions, because they specify resources with configurations that must be constructed together – altering the configuration for one resource may alter the configuration for others in the region. This is the first step in providing the architectural abstraction for predicate-aware mapping; it specifies *where* predicates may alter the configuration. The next step is to specify *how* the predicates may alter the configuration.

5.4.2 Modifying Configuration Retrieval

To support predicate-aware execution, the simple configuration memory design must be modified so that live predicates may be used to alter the run-time configuration selection. There are several candidate modification points in this design where predicates could be used to alter the configuration. These are indicated in Figure 5.18 by the light-blue highlights. In a typical general-purpose processor, the modification for control is made to the program counter. The closest equivalent in these reconfigurable architectures is the modulo phase counter. The scalable nature of the modulo phase counter is important to these spatial architectures. If control was implemented via adjustment of the base counter sequence, such as in a program counter in a CPU, the modification would need to be broadcast to all regions to keep them synchronized, which leads to global signaling that does not scale well. For this reason, modifying the sequence itself is excluded from consideration for this work, and so it is not highlighted in Figure 5.18.

Predicates may alter the behavior in any of these highlighted areas, but with different trade-offs. The highlight in the lower-left of Figure 5.18 represents the distribution

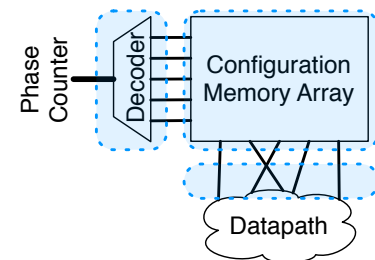


Figure 5.18: Configuration hardware diagram with potential modification points marked in blue

of the configuration values to the reconfigurable logic across the chip. This will be referred to as modifying the configuration at distribution time. It would be possible to pass the output configuration bits through some reconfigurable logic and combine that with predicates from the computation to alter the configuration. This would require that the compiler algorithms synthesize this extra logic to transform the configurations that are being generated with the predicate bits. This technique is one that has been demonstrated in the RaPiD architecture [ECF⁺97a]. To enable completely general predicate control could require extra hardware at the granularity of every bit of the configuration for the reconfigurable hardware. This is on par with the reconfigurable logic itself.

The next area for possible modifications is the memory array that holds the configurations, highlighted in the upper right of Figure 5.18. This array can be viewed as the embodiment of a function mapping the decoded address to a configuration word. Instead of retrieving values from an SRAM array, there could be a configurable logic circuit that computes the configuration as a function of the decoded modulo-counter address and the predicate bits. If this can be an arbitrary function implemented by a full look-up table, then the memory array remains the same and the actual modification simply includes the predicate bits in the address to the configuration memory. If the function is restricted in some other way, representing those restrictions in the mapping algorithms will be architecture dependent and potentially very complicated.

The final area highlighted in the diagram is the decoder that translates between the current phase counter and the configuration that will be retrieved from the memory array. Modifications to this portion of the design have a high amount of leverage. The counter bits are used as address bits for the configuration memory, and modifying a single address bit here will result in retrieval of an entirely different configuration memory line. This can be accomplished by the addition of a few multiplexers on the address lines to choose between a phase counter bit or a control datapath bit.

When compared to the other two options for modification, altering the address decoding stage is appealing for this initial study. It has a low hardware overhead and provides a simple abstraction compared to the other two options. Additionally, having phase and predicate bits interact with the configuration system in similar ways maps well to our

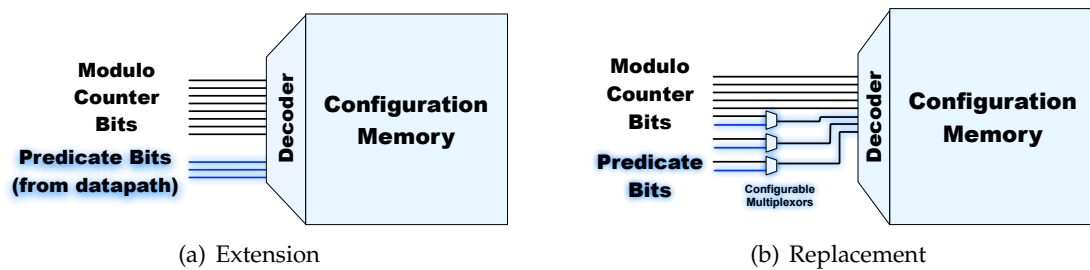


Figure 5.19: Combining counter and predicate bits.

abstraction of treating the phases as top-level conditions in the CDT. For these reasons, this is the method used in this study, with the others left for future exploration.

5.4.3 Exposing Configuration Control

An abstraction of the interaction between predicate bits and the configuration retrieval needs to be provided to the compiler to allow the compiler to properly allocate and route the predicate bits. The decoding stage of configuration retrieval is the focus for modifications, so the bits of the phase counter and the predicate bits that will be decoded can be viewed as address bits for selecting a configuration. There are two straightforward options for combining predicate bits and phase bits in the decoding stage that are illustrated in Figure 5.19. The first, Figure 5.19(a), is to simply extend the address bits generated by the modulo counter with the predicate bits. This modification leads to a large increase in size of the configuration memory array.

The second option, Figure 5.19(b), is to replace the modulo counter bits in a reconfigurable way. For applications requiring a large II , the entire configuration memory is usable. For high-throughput applications that would otherwise waste much of the configuration memory, the unused space can be dedicated to supporting more complex control flow. Note that in both cases, the predicate bits come from signals routed through the interconnect of the array, so a different set of predicate bits can be used in each phase of the schedule. This will allow the compiler to use these extension bits more efficiently, mapping predicates to them only when those predicates are needed. This hardware model allows the compiler to trade-off between applications with large initiation interval re-

quirements and applications with more complex control. This supports the motivating case of using wasted configuration memory in high throughput kernels to allow more complicated control flow.

From the point of view of the compiler, there are limits on the number of predicate bits that can be used in both cases. The predicate bit usage will have to be tracked by the compiler to ensure that they are not oversubscribed. For Figure 5.19(a), this capacity is directly defined by the number of predicate bit paths that run to the decoder. The configurable trade-off shown in Figure 5.19(b) between modulo counter bits and predicate bits means the amount of control flow that can be supported and the maximum Π will be interrelated. With the scheduling preceding placement and routing in the tool-chain, the Π for the kernel will be set during scheduling. Once the Π is set, the number modulo-counter bit lines required to support that Π are reserved. At that point, the number of predicate bits available during placement and routing is determined. For Figure 5.19(a), this is simply the number of predicate bits designed into the architecture. For Figure 5.19(b), this is determined by reserving exactly the number of bits required to support the Π , and allowing any remaining configurable lines to act as predicate bits.

In addition to allocating predicate resources in the configuration hardware, a predicate-aware version of SPR will need to ensure that the predicates are communicated from the spatio-temporal location they are computed to the configuration hardware for regions where they are needed. At some point in this communication, the predicate signals will need to cross from the configurable interconnect resources into the configuration hardware. This location in the architecture will be known as a predicate gateway. A predicate gateway will be used by the router as a target sink for routing predicate signals to a region that needs them. For simplicity, a one-to-one correspondence between regions predicate gateways is assumed, though this could be lifted in future work.

5.5 Supported Abstractions

This chapter has explored the concepts and representations that lay the groundwork for creating predicate-aware scheduling, placement and routing algorithms. This section

summarizes the representations that will be used to develop the compiler algorithms in the subsequent chapters.

5.5.1 Control Dependence Tree

The control dependence tree of Section 5.2 provides a way of communicating the control structure of the program to the compiler. Information about the run-time mutually-exclusive nature of operations is directly provided, avoiding the need to extract this information through inference as in [JS96, SHA00]. Additionally, awareness of the hierarchical nature of conditions allows us to exploit partial promotion.

5.5.2 Aged and Aggregate Conditions

The operations of the program will need to be scheduled and grouped for sharing resources. As part of the scheduling process, the conditions under which operations execute will go through a modulo-expansion process to differentiate conditions from overlapping loop iterations, as described in Section 5.2.4. To support efficient conflict testing, conditions will be represented internally as the conflict bit vector of the condition along with an age representing the condition's modulo-expansion, as shown in the left side of Figure 5.15 in Section 5.3. These are combined into aggregate execution conditions as shown in the right side of Figure 5.15 to represent a group of conditions.

5.5.3 Regions

The configuration for large CGRAs will likely be divided up spatially to support scalability. For a predicate aware compiler, these divisions become visible, and are represented as different configuration regions to the compiler. The abstraction of regions is described in Section 5.4.1, and is used to indicate the resources that all have their configuration retrieved by the same modulo-counter and predicate bits. Up to this point SPR has treated each configurable element independently programmable within each phase of the modulo schedule. If a predicate bit is used to change the configuration of a single element, it will alter the configuration that is retrieved for all elements in that region. Different elements in different areas of the region could require completely unrelated predicates. In

this case, the full cross-product of those predicate values could occur at run-time, each retrieving a different configuration word. It is up to the compiler to ensure that all of those configuration words are constructed in a way that provides the proper configuration to each element in the region.

5.5.4 Predicate Gateway

With CGRAs divided into regions, the predicates representing which conditions are true at run-time will need to be distributed from their origin to the regions that depend on them for choosing a configuration. In order to ensure these signals get to the regions at the right time, they will be routed from their origin to a sink target for the region called a predicate gateway, as described in Section 5.4.3. For simplicity in the initial implementation, a one-to-one relationship is assumed between configuration regions and predicate gateways. These predicate gateways will also represent the number of predicate bits that can be used in the region. It is up to the compiler to be aware of this capacity and ensure it is not over-subscribed.

5.6 Predicate Aware SPR Overview

The next three chapters will build on the foundational concepts laid out here to present a formulation for predicate-aware scheduling, placement, and routing in modulo-scheduled CGRAs. Chapter 6 presents predicate-aware scheduling. The predicate-aware scheduling used for CGRAs builds on the work of [SMDL03] for predicate-aware modulo scheduling of VLIW architectures. Additionally, it takes into account the modulo-expansion of predicate signals by using aggregate execution conditions to represent resource sharing in the resource table used to track allocations.

Chapter 7 covers predicate-aware placement. Starting from the Simulated Annealing based placement originally used in SPR, the placer is extended to represent multiple occupancy of resources using aggregate execution conditions, and will turn any conflicts between the conditions of co-located operations into a cost that can be optimized by annealing. Additionally, costs are created for tracking over-subscription of the predicate bits available for configuration switching in each region of the architecture.

Chapter 8 details the most complex changes made to create the predicate-aware SPR, the predicate-aware routing. The conditions that apply to operations from the CDT are extended to include the routes of data values between operations. These routes take on the condition of the source operation at the beginning and the sink operation at the end. If the source and sink conditions differ, it is up to the router to choose the best condition to route under at intermediate points. Additionally, as a route traverses different configuration regions of the architecture, the source or sink conditions may not be available. To make progress and limit the loss of sharing opportunity, partial promotion is used to assign an alternate condition to route under.

The PathFinder congestion negotiation is made predicate-aware in separate ways for control congestion and signal congestion. Predicate-aware signal congestion is tracked using an aggregate execution condition per routing resource, which allows co-routing of signals routed under mutually exclusive conditions in the same way as co-location is allowed in placement. Predicate-aware control congestion is tracked using a generalization of the control congestion presented in Chapter 4, with conditions that are unavailable within the region taking the place of the phase signals that are unavailable in static muxes.

The QuickRoute pipelined routing algorithm is extended to support predicate-aware sharing through careful separation of signal fan-outs into separate routing queues based on the destination routing condition. Once signal routes are completed, they are only used to seed other fan-outs that are routed under a compatible set of conditions. Along the way, the actual conditions of a route are tracked, along with the transition from source-condition to sink-condition routing and any partial promotion needed to cross regions without certain predicates available.

Finally, in Chapter 9, the benefits of predicate-aware mapping will be evaluated across a set of benchmarks. The performance will be compared against both the standard SPR mapping and upper bounds derived through relaxing constraints on the mapping, demonstrating the benefits of sharing for control heavy, resource constrained applications.

Chapter 6

PREDICATE AWARE SCHEDULING IN CGRAS

In SPR, mapping an application to the CGRA is done in three stages – scheduling, placement and routing. The following chapters explore adding predicate awareness to these stages, beginning here with scheduling. The predicate-aware version of SPR, will be referred to as PA-SPR.

Adding predicate awareness to scheduling means two things. Primarily, it means being able to share resources between operations that are on mutually-exclusive control paths. Secondly, it means managing the proper trade-off between being able to share resources once a predicate signal has been generated, and being able to speculatively execute operations before the predicate signal is available.

The original SPR scheduling algorithm of Iterative Modulo Scheduling comes from the VLIW community. Following the evolution of scheduling algorithms for VLIWs leads to work for predicate-aware scheduling found in [SMDL03, SMD04]. That work will form the basis of SPR's predicate-aware CGRA scheduling stage, as it presents a method for addressing the primary concern of sharing resources between mutually-exclusive operations. Earlier work on optimizations that enable more parallelism for modulo-scheduling [TLS90] introduced promotion for speculative execution as a way to reduce the minimum recurrence II. That idea is integrated with using sharing to reduce the minimum resource II to support the secondary goal of finding the proper trade-off between sharing and speculation.

This chapter will provide an overview of predicate aware scheduling in VLIWs and then describe how it has been applied to CGRAs in PA-SPR. In addition to combining predicate aware scheduling and promotion, the notion of predicate ages is used to deal with modulo-variable expansion of the predicates.

6.1 Predicate Aware VLIW Scheduling

This section will provide a basic overview of predicate-aware VLIW scheduling; for a more detailed discussion see [SMDL03]. For clarity, the specific algorithm presented in [SMDL03] will be referred to by the acronym used by the authors – PAMS (predicate-aware modulo scheduling).

PAMS is based on iterative-modulo scheduling (IMS), the same scheduling used in SPR.

PAMS is based on iterative-modulo scheduling (IMS), the same scheduling used in SPR. Figure

6.1 shows the motivating example for PAMS, which is similar to the SAD example from Chapter 5. PAMS is designed to work on predicated code, which is shown in part (b) of Figure 6.1. Here, predicates $p1$ and $p2$ are mutually exclusive, so the goal is to schedule the operations that depend on those predicates on the same resources. This is shown in Figure 6.2, comparing the longer $\Pi=4$ schedule obtained with IMS to the shorter $\Pi=3$ schedule that is possible when operations can share resources.

```

1: for ( i = 0; i < im_size; i ++ )
2: {
3:     prod = q_im[i] * bin_size;
4:     if (q_im[i] ≥ 1)
5:         res[i] = prod - correction;
6:     else
7:         res[i] = prod + correction;
8: }

```

(a) Source code

```

op1: t1 = load(i1, q_im) if p0;
op2: prod = mult(t1, tbs) if p0;
op3: p1, p2 = cmpp.lt.uu(t1, 1) if p0;
op4: t2 = sub(prod, tcor) if p1;
op5: t2 = add(prod, tcor) if p2;
op6: store(i1+ = 4, res, t2) if p0;
op7: if(i ++ < im_size) goto op1 if p0;

```

(b) Assembly code after if-conversion ($p0=$ True)

Figure 6.1: Fig. 1 of [SMDL03]: Example code segment

Time	A	M	B
0	op5	op1	
1	op4	op6	
2	op2		
3	op3		op7

(a) $\Pi = 4$

Time	A	M	B
0	op3	op1	
1	op4	op5	
2	op2	op6	op7

(b) $\Pi = 3$

Figure 6.2: Fig. 4 of [SMDL03]: IMS kernel (a) versus PAMS kernel (b) schedule

In the PAMS formulation, the Predicate Query System (PQS) [JS96] infers the relationships between predicates in predicated code. A predicate being true is a concrete run-time

representation of the abstract idea that a condition has enabled a control path. To keep the terminology clear, predicates will always refer to a run-time signal, and condition will refer to the abstract execution state that signal represents.

The relationships that are inferred by PQS are stored as Boolean expressions. To determine whether two predicates are mutually-exclusive, or disjoint as they are referred to in [SMDL03], the expressions are ANDed together. If the result is unsatisfiable, then the predicates will never be true at the same time, and operations predicated by them may use the same resource.

These predicate expressions are first used to compute the resource minimum II (resII). In IMS, all the operations are added to a usage count for the resources they use, and the largest count sets the resII. For PAMS, this would over-count, so the predicate expressions are used to find ways of combining usage in the counting process. A list of predicate expressions is kept for each resource in the architecture, with each expression representing a single use of that resource. For each operation, the list of predicate expressions for the resource that operation maps to is examined in order. If the operation is disjoint with any of the existing expressions in the list, it is added to that expression by ORing together the expressions - representing the sharing. If there are no existing expressions that an operation can combine with, its expression is added to the end of the list, representing an increment in the usage count. At the end, the length of the longest list sets the resII.

Time	Res 1^{may}		Res 2^{may}		Res m^{must}	
0	op1	op2 pred_expr01		 op3 true
1				 op4 true
...
n			op5	pred_exprn2

Figure 6.3: Fig. 8 of [SMDL03]: Predicate-aware reservation table

PAMS augments the individual reservations in the IMS resource table with these Boolean expressions to make IMS predicate aware, as shown in Figure 6.3. When attempting to find a slot in the resource table for an operation, that operation's predicate

expression can be tested with the reservation's expression for disjointness. If they are disjoint, the operation can share the slot with the other operations that have already reserved it. It is added to the list of operations in the reservation, and its predicate expression is ORed with the expression in the reservation to represent the union of the operations in that reservation slot.

With this enhanced resource table and the appropriate access and update routines, the rest of the scheduling proceeds in the conventional IMS manner. The authors were able to use PAMS to demonstrate an average performance gain of 18% and 7% on 4-issue and 6-issue VLIW processors, respectively, across the benchmarks they used.

6.2 Applying Predicate Aware Scheduling to CGRAs

The same general approach is taken making the IMS portion of SPR predicate aware, though some of the implementation details differ. In particular, PA-SPR extends the resource table with predicate-aware sharing information. PA-SPR also includes the appropriate access and update routines to maintain the sharing information, as in [SMDL03]. However, PA-SPR uses an improved method for computing a resII, and the predicate expressions are replaced with aggregate execution conditions. Additionally, PA-SPR balances the resII against the recurrence minimum II (recII) to achieve the best possible performance. These differences are detailed in this section.

6.2.1 Computing the Resource Minimum II

IMS is an iterative algorithm, attempting to schedule heuristically at lower II values, and trying incrementally higher II values after a budgeted number of scheduling operations. A minimum II is determined at the beginning of this process to avoid needless iteration at unschedulable II values.

The PAMS scheduler creates the resII by greedily packing operations together when their predicate expressions are disjoint. Consider the compatibility graph operations that use the same resource type, defined as a graph $G = (V, E)$, where V is the set of operations that use a given resource type and an edge $e_{u,v} \in E$ where $u, v \in V$ if and only if the predicate expressions of u and v are disjoint. Operations $v_1, \dots, v_n \in V$ are able to

share a resource if they are all compatible with one another, e.g. they form a clique in G . To determine a lower bound on the resource minimum Π , we want to find the minimum number of resources the operations can be packed into, which is equivalent to finding the minimum clique cover of G .

Given that the CLIQUE PARTITION is problem 13 in Karp's original list of NP-complete problems [Kar72], it is clearly intractable to find the optimal $\text{res}\Pi$ for large kernels. The PAMS greedy approach will either find the correct minimum Π or overestimate it. However, when computing a lower bound for an iterative algorithm that will work upwards, the proper approach is to underestimate the $\text{res}\Pi$.

PA-SPR will compute an under-estimate by assuming perfect sharing between operations that are partitioned into mutually-exclusive subsets in the CDT. A count of operations by type will be created by starting with the counts at the leaves and performing a reduction up the CDT to the root. When combining counts at partition nodes, the per-type maximum value will be passed up the tree. Combining counts at condition nodes is done by per-type addition. The maximum operation reflects the resource count if all of the operations in the sub-trees could be perfectly shared. The addition operation captures the property that ancestors and sub-trees joined at condition nodes cannot share resources. Once the reduction is completed, the resource minimum Π can be computed from the resulting per-type operation counts using the original IMS method.

6.2.2 *Alternative to PQS*

The PAMS algorithm targets scheduling predicated code. As a result, PAMS uses the PQS to extract the relationships between the predicates and store them as predicate expressions. In PA-SPR, the predicate relationships are provided directly by the front-end compiler in the form of the CDT. PA-SPR generates conflict vectors from this representation and uses aggregate execution conditions to replace the union predicate expressions in the predicate-aware resource table.

One issue that is not clearly addressed in the PAMS work is the potential problem of predicates that are live for more than Π cycles. As was discussed in Section 5.2.4, when predicates are live for multiple waves, operations that depend on them are only compara-

ble if they are using predicates generated in the same iteration. The execution conditions and aggregate execution conditions used by PA-SPR include the extra scheduling information necessary to deal with this issue. This ensures that operations from different iterations are not accidentally shared on the same resources, even though the predicate expressions (or conflict bit-vector) alone may indicate this is possible.

6.2.3 Promotion for Dependence Trimming

In a spatial architecture, the `then` and `else` branches of an `if` statement are often executed simultaneously before the condition of the `if` is resolved, with the correct result selected once the condition has been computed. This is called eager execution and predicate lifting in [TLS90]. They found it to be very useful for extracting enough parallelism for speedups on their benchmarks.

Predicate-aware sharing of resources re-introduces the dependence between the predicate computation and the operations that are sharing; the predicate is needed to choose which operation to execute. These extra dependencies may create larger recurrence loops. In a highly resource-constrained situation, this may not be an issue because the resII constraint will be larger than the recII . However, in cases that are marginally resource constrained, the resII may be reduced at the expense of a larger recII . The larger of the two sets a lower bound on the II , so a large increase in recII will squander any gains from resource sharing.

PA-SPR handles this situation by using promotion to balance the recII against the resII prior to scheduling, while the minimum II is still being resolved. Initially, all operation conditions are set by the CDT, and an implicit dependence is created between an operation and the operation that computes the predicate for its condition. The scheduler first computes the resII and recII bounds. If the recII is greater than the resII and there are implicit dependencies from predicate computations on the largest recurrence loops, the scheduler trims those dependencies using promotion. The scheduler removes the dependence between the predicate computation and the predicated operation by promoting the operation to unconditionally execute under the `TRUE` condition. It repeats this process until the recII is calculated to be less than or equal to the resII , or there are no more of

these predicate dependencies on the critical recurrence loops. The benefits of this balancing process are evaluated in Chapter 9.

6.2.4 The Region Abstraction and Scheduling

One aspect of predicate-aware CGRA mapping that is notably not handled during scheduling is the region and predicate capacity limits discussed in Chapter 5. PA-SPR assumes perfect resource sharing when computing the resII. Operations are greedily packed together in the resource table during scheduling, but with the eviction and iteration inherent in the IMS algorithm. Both of these ignore the limitations on the number of predicates that can be used to switch between operations and where those predicates are distributed.

Properly optimizing for these limitations requires deciding where the predicates are computed and where they are distributed to – a decision that requires a spatial awareness of the architecture. To maintain the separation of concerns of the original SPR between scheduling, placement and routing, this spatial optimization is left to the placement and routing stages in PA-SPR. If the resulting schedule cannot be realized, more constraints on the scheduling will be provided to the scheduler and the process will iterate. The next chapter begins the discussion of the spatially aware stages of PA-SPR, covering predicate aware placement.

Chapter 7

PREDICATE AWARE PLACEMENT

This chapter details the changes needed to add predicate awareness to SPR in the placement stage of PA-SPR. The placement of SPR is based on the Simulated Annealing optimization framework [KGV83]. The modifications focus on providing the proper costs and constraints to include resource sharing in the optimization process.

Predicate-aware placement requires changes that can be divided into two main conceptual categories. The first category is made up of changes that allow for more than one operation per device, which can be thought of as the local sharing concerns. The second category contains changes that are needed to optimize routing predicates to region gateways and the associated limits on the number of predicates available in the region. Many devices fall within the same region, so these will be considered region costs.

The adaptations to the Simulated Annealing cost model for these two categories will be discussed in Section 7.2 and Section 7.3, after the goals of predicate aware placement are covered in Section 7.1. This chapter concludes with a description of the process of moving on after either a successful or failed placement.

7.1 The Problem

The goal of predicate aware placement in PA-SPR is to take an initial schedule and spatially place the operations across the architecture in a way that will be routable by the routing stage. The initial schedule already contains some operations that are assigned to the same resources. This schedule will be turned into an initial placement by randomly assigning reservation slots to resources in the architecture and then annealing that random placement. The schedule includes operations that are packed together, but the packing was performed only considering compatibility and dependence constraints. Once this schedule is converted to an initial placement, the gateways of the regions may be over-

subscribed, requiring more predicates to accomplish the sharing than there are predicate ports in the gateway. The predicates may also not be routable from the places they are computed to all of the gateways where they are needed. During the annealing process, these issues must be resolved by moving operations or else the routing stage will not be able to successfully complete.

Simulated annealing works by repeatedly generating random changes to the system, in this case operation moves or swaps, and probabilistically accepting that change based on the difference in system energy, or cost. In SPR, the optimization is formulated as a cost reduction, so moves that reduce cost are always accepted, and moves that increase the cost are accepted based on a cooling schedule. Early in the process, the simulated temperature is high, which means the probability of accepting a move that increases the cost is high. The temperature is reduced according to the adaptive process used by VPR [BR97a], and at lower temperatures, the probability of accepting a move is reduced. Since generating a routable placement is the primary goal, the costs of moves are formulated in terms of routability.

The first obstacle to predicate-aware resource sharing in placement is the assumption of a one-to-one mapping of operations to devices in SPR. The move generation in SPR assumes that there is never more than one operation mapped to a device, and never generates moves that cause multiple operations to be mapped to a device. Instead, a swap happens when any operation is moved to a device with an existing operation.

The initial schedule will have already grouped compatible operations on devices. However, that sharing may over-subscribe the predicate resources for the associated region. In order to alleviate this over-subscription, PA-SPR must be able to generate moves that can change the sharing initially set up by the scheduler. This implies that PA-SPR must track the individual operations and the compatibility between them, maintaining a many-to-one mapping of operations to devices. The approach used to accomplish this is covered in Section 7.2.

The second obstacle to predicate-aware mapping is the limitation in SPR of only considering signals that route to or from an operation in that operation's cost. In order to support predicate-aware mapping and new costs need to be created that reflect the

routability of the predicates that enable sharing. These signals are routed from their producer to the gateway of the region an operation is in, not the device where an operation is placed. Additionally, the predicates only need to be routed once per region, but multiple operations in the same region may all require the predicate for sharing. Finally, operations that execute under a particular condition may not be sharing a device with other operations. In this case, the predicate is not required by that operation any more, so it should not be included in the estimate of the routing cost.

The local nature of cost computations in SPR enables an important optimization. As moves are generated, the effect that move will have on the total cost can be computed incrementally by only considering the delta in the cost of operations and the signals attached to them. This keeps the complexity of computing the cost of a move limited to the average degree of nodes in the computation graph. If the total cost needed to be recomputed at each move, the complexity would grow to the order of the computation graph. When incorporating the non-local costs, especially those that are affected by all operations in a region, it is important to find a way of computing the cost incrementally to preserve the efficiency of adjusting the cost at each move. The method used in PA-SPR to accomplish this is detailed in Section 7.3.

7.2 Local Sharing

The first step in making resource sharing possible in PA-SPR placement is to allow many-to-one mapping of operations to devices. The placer is provided with an initial schedule that contains groups of compatible operations sharing the reservation entries of the resource table. There are three possible choices for the amount of regrouping the placer may perform to adjust this sharing:

- None – each set of operations in a reservation entry are treated as a single operation by the placer, maintaining the view of a one-to-one mapping.

- Valid sharing – operations may be moved between sharing groups, as long as the operations mapped to a single resource execute under conditions that will make them mutually exclusive at run-time.
- Conflicted sharing – operations may be moved between sharing groups, and if operations that are not mutually exclusive are mapped to the same resource, a large cost is incurred to make this an unlikely state at the end of the annealing process.

The first option of None is not feasible, because the placer must reconcile any over-subscription in the initial placement. The scheduler allowed sharing in reservation entries without any notion of regions or the limitations they put on predicate availability. PA-SPR sets up the initial placement by assigning the sets of operations that share a reservation in the schedule to a random resource of the appropriate type. The predicates required for the sharing may not be routable to the region within the latency constraints of the schedule, or the amount of sharing within a region may require more predicates than the capacity of the region gateway. Therefore, the placer must reconcile these limitations using routability estimates and a description of the architecture which includes the regions. To do so, it must move operations to remove sharing where the predicate gateways are over-subscribed, and it may need to move sharing based on certain predicates closer to where those predicates are generated.

Before deciding between maintaining valid sharing and allowing conflicted sharing, it is enlightening to consider how SPR handles other constraints within the simulated annealing framework. In SPR, the placer is allowed to alter the scheduled time of operations, but only within the limits of the schedule slack. This means that the validity of the schedule is an invariant throughout placement. An alternative would be to allow the placer to change the scheduled time in ways that produce an invalid schedule, but at very high cost. Then, it is up to the annealing process to eliminate all of the scheduling violations through optimization of the cost. However, simulated annealing is an optimization framework, not a constraint satisfaction framework. It is possible that violating the schedule for one operation could allow a lower cost placement for enough other operations to

offset the chosen cost for a single schedule violation. In this case, it is more optimal to retain the violation from the point of view of the optimization problem. Unfortunately, this is not a viable mapping solution because any violation of scheduling constraints means that an operation will either happen before its inputs are ready or after consumers of its output have started execution.

This pathological case can be avoided by either choosing a high enough cost such that no combination of other benefits can outweigh it, or by simply maintaining the invariant of a valid schedule over all moves. SPR is designed to enable architectural exploration by making few assumptions about the architecture, so choosing an appropriate cost may not be possible, and the latter approach is used instead. The scheduler provides a valid schedule to the placer, and the placer only generates moves that respect the scheduling constraints. This way, the schedule remains valid throughout placement.

The same reasoning applies to the options of maintaining valid sharing or allowing conflicted sharing in PA-SPR. This initial exploration of predicate aware sharing will use the method that has proven successful for scheduling and maintain valid sharing throughout placement. Initially, groups of operations that share the same slot in the resource table of the scheduler will be mapped to a random device of the appropriate type in the architecture. The placer maintains a list of operations mapped to a single device, along with an aggregate execution condition that can be used for tracking operation compatibility, allowing the placer to ensure operations will be compatible when generating moves.

There is also a small change that needs to be made to the procedure of calculating the change in an individual operation cost when it is moved. The original SPR counts the change in routing cost for all of the operation's input and output signals. For PA-SPR, the change in the routing cost of sending a predicate to the region gateway is included as well.

A predicate only needs to be routed to the region once per phase to be usable by operations on all devices in the region. As a result, counting the cost of routing the predicate for each operation may seem like over-counting. However, this gives the simulated annealing algorithm a toe-hold for moving a set of operations that use the same predicate to a region that is closer to the predicate generation. The placer does not generate moves for

all operations using a given predicate at once. The entire set must be moved by individual moves. In order for this to happen with reasonable likelihood, individual moves that may eventually lead to a better placement must show some benefit in the cost function. In general, this will tend to pull operations that execute under a particular condition towards the operation that generates the predicate for that condition, or the operation that generates the predicate towards the regions which have the most operations using it.

These changes give the placer stage in PA-SPR the ability to manage the sharing of a device by multiple compatible operations. The added cost calculations can still be executed incrementally per operation move. However, none of these changes will help with the problem of an over-committed region gateway if too many predicates are required in the region. The next section covers the changes that are needed to cope with this aspect of placement.

7.3 Region Costs

The region and predicate gateway of the region were introduced in Section 5.5.3 and Section 5.5.4 as a way of providing a scalable abstraction of hardware modifications to support predicate aware sharing. The difference in cardinality between regions and devices is the key aspect that support scalability and abstraction across architectures. The architect is free to set the region of influence of a predicate signal and describe where and how many can connect to the data interconnect.

The difference in cardinality also means that tracking region over-subscription cannot be easily done locally on a per-device basis. Instead, PA-SPR includes per-region placement information in addition to the per-device information. The union of required predicates is tracked per-region. Each region has a list of predicates that are needed within the region, along with a count of how many operations in the region require the predicate. The counts are used to manage the incremental updates. As the last operation that requires a predicate is moved out of the region, the count for the predicate will reach zero and it is clear it is no longer needed. If an operation using a particular predicate is moved out of the region, but there are still some left using it in the region, the count will reflect

this and the predicate will still be tracked in the region. Without the counts, all of the operations in the region would need to be checked to ensure a complete and up-to-date set of predicates were being accounted for in the region after each modification.

Once the per-region accounting is in place, it is easy to detect when a region is over its predicate capacity by checking the length of the list of required predicates. In PA-SPR, predicate bits are assumed to replace modulo counter bits, so the capacity of the regions is dependent on the current II of the schedule. PA-SPR is provided with a maximum II for the architecture, and it is assumed that there are $\log_2(maxII)$ counter bits available for indexing the configuration memory. In each region, there is a predicate gateway device with several input ports. Each input port represents a bit that may be used by a predicate, and each port can be used to replace one of the high order modulo counter bits. For a given scheduled II , the capacity in the region for predicate bits, p is given by Equation (7.1):

$$p = \min (\log_2(maxII) - \log_2(schedII), n_p) \quad (7.1)$$

The $\log_2(schedII)$ term sets aside the bits that are needed to index through the phases, providing a global heartbeat to keep the architecture in sync, while the remaining bits are available for use as predicates. The number of ports in the gateway is represented by n_p . The capacity can never be higher than the number of ports or the number of remaining bits once the phase bits have been reserved.

PA-SPR can track predicate usage and determine that a region is over its capacity, but resolving the issue requires a cost model that will bias the annealing towards placements that are at or below capacity. Resolving an over-capacity conflict requires operations to be moved in a way that reduces the number of predicates needed for sharing. To accomplish this, an extra cost will be assigned for any region that is over capacity, but regions that are at or below capacity will incur no cost. Additionally, because the capacity is a constraint and a valid mapping is not possible with any over-capacity regions, each predicate required over the capacity will be tracked as a broken constraint the same way

unroutable connections are tracked. If there are any over-capacity regions at the end of the annealing process, it will be considered a failed placement.

A single predicate may be required by a single operation or many operations in the region. If given the choice of removing a predicate that supports sharing for few operations versus removing one that supports many, it is clearly better to remove the predicate that supports fewer. Any operations that are supported by the predicate will need to be moved to new locations, either no longer sharing resources or sharing resources in another location.

To reflect this, there should be a greater benefit to moving an operation, and potentially removing its condition, that has the fewest other operations under the same condition – the ones that get the least benefit from having the condition available in the region. This is akin to the principle of diminishing returns, in this case the principle of diminishing costs. As more operations are added to an existing condition in an over capacity region, they should increase the cost, but by a decreasing amount because they are increasing the virtualization that particular condition provides. Conversely, if there are only a couple of operations in a region, removing them will greatly decrease the placement cost, until the point of removing the overcapacity problem, where the cost will drop a great deal. The cost, c , for an over-capacity region is given in Equation (7.2):

$$c = \sum_{pred} \sum_{i=1}^{n_o} \max \left((base * rate^i), c_{min} \right) \quad (7.2)$$

Where $base$ represents the initial base cost for over-provisioning, n_o is the number of operations using the predicate $pred$ in the region, and $rate$ is the rate of diminishing costs. The cost is clamped at c_{min} so that there is a minimum cost to any over-capacity predicates. In this work, $base = 1023$, $c_{min} = 1$ and $rate = .95$.

The difference in cardinality between regions and operations also means that incremental cost updates during placer moves must be done carefully. If each move was limited to one operation or a swap of two operations, it would be a simple matter of checking whether the cost needed adjustment based on one or two regions. However, the clustering covered in Section 3.3 can increase a move to include many operations across a

cluster. If regions are defined at a smaller size than clusters, this could lead to a move involving many regions. It may also involve many operations per region. To ensure that the costs are properly maintained in an incremental fashion, the set of operations for a move is iterated through to build a set of regions requiring updates. The over-capacity cost is calculated (or retrieved) once per region before the operations move, and re-calculated once per region after the operations move, with the individual operation moves updating the predicate usage counts in the interim. Once the before and after costs are calculated, the difference is taken and added to the total cost to complete the incremental update.

7.4 Finalizing the Placement

The annealing process uses an adaptive temperature schedule, and once the cost improvement rate has slowed down enough, the process terminates. For a normal optimization problem, this would be the final result. However, there are additional constraints involved for placement in PA-SPR.

- All routes must be routable according to the cost function, including the routes of predicates to the regions where sharing depends upon those predicates.
- The number of predicates required in a region must not exceed the capacity.

If any of these constraints are not satisfied, it is clear that routing will not succeed. The iterative nature of SPR and PA-SPR handles this by relaxing the problem and re-starting from the scheduling stage. The latency padding technique from Section 3.1 was shown to handle any failures in the first constraint, and is applied to the predicate routes as well in PA-SPR. To deal with violation of the second constraint, the minimum II is set to one more than the currently scheduled II before iterating. This effectively adds more virtual resource slots, but at a performance cost. A finer grained approach may be possible, for instance promoting all operations from the condition that has the fewest, but incrementing the II provides an easily implemented alternative for this initial study. By incrementing the II, more virtual resources are provided, which reduces the amount of sharing needed to fit within the allotted resources.

If all constraints are satisfied, then PA-SPR can prepare to move on to the routing stage. The routes for predicates must be finalized before moving on. Whereas all data connections between operations must be routed, the predicate connections between the operations that generate them and the region gateways should be routed only if they are needed. Predicate routes are generated for the predicates of any operations that are sharing resources. For the current implementation of PA-SPR, the ports of the gateway are assumed to have equivalent connectivity, so predicates that require routes to a gateway are chosen at random and assigned to the ports of the gateway in increasing port number. This means the lowest port number should correspond to the most significant counter bit to ensure a proper trade-off between bits used for the phase and bits used for the predicates.

Once these dynamic routes have been generated, PA-SPR can start the routing process. The modifications that enable predicate-aware routing are described in the next chapter, and are far more extensive than those of either scheduling or placement.

Chapter 8

PREDICATE AWARE ROUTING

CGRAs use compiler controlled routing interconnects that must be configured to enable communication between operations. Once the placer has determined the locations in the CGRA where operations will execute, the router must create the configurations that will allow that communication. The routing problem consists of finding an assignment of settings for configurable switch points in the architecture that ensures signals will travel from the generating operation's spatio-temporal location, or source, to the appropriate consuming operations' spatio-temporal locations, or sinks. The problem of predicate aware routing adds the extra dimension of sharing routing resources based on the condition compatibility of the sources and sinks. The approach taken in PA-SPR limits the hardware modifications to those already described and extends the current routing algorithms used by SPR to be predicate-aware.

This chapter presents the modifications of the PathFinder and QuickRoute algorithms used by SPR to support predicate-aware routing. For PathFinder, the notion of congestion will need to be refined to take into account sharing opportunities between signals that are mutually exclusive at run-time. Mutually exclusive signals should be allowed to share resources – instead of assuming co-location in the unrolled architecture graph means there is congestion. The costs for both control and signal congestion will be examined separately, each for both present sharing and historical sharing, as they were in Chapter 4. These modifications are presented in Section 8.2.

PA-SPR must route predicate signals to any regions where operations are sharing compute resources based on the mutual-exclusivity of those conditions. This enables switching between the appropriate configurations at run-time. Operation sharing within a region defines the minimum set of predicates that must be routed to the region. These are in addition to signals that must be routed between operations, as required by the

program data flow. During the routing process, the router may need other predicate signals in certain regions to enable mutually-exclusive signal sharing of routing resources. This means that the process of signal routing may create more sinks for predicate signals that must also be routed, implying that the routing problem itself is dependent on the current partial routing solution. To handle this dependence, a method that updates the predicate routes at the granularity of PathFinder iterations is proposed in Section 8.3.

The single-sink routing portion of QuickRoute can remain largely unchanged. QuickRoute uses the PathFinder costs during routing, so properly adjusting those costs will result in predicate-aware routing decisions. However, the source and the sink of a particular signal may execute under different conditions. This means that a predicate aware signal router must choose the appropriate condition under which to reserve resources along the route, ensuring the signal will have a complete path at run-time. This choice will be covered in Section 8.4.5.

The multi-sink portion of PathFinder that combines single-sink QuickRoute results requires more modification to become predicate-aware. In the SPR router, QuickRoute routes each sink individually. Subsequent sinks for a common source are able to start from the routing resources of prior sinks with zero initial cost, using those resources to provide seed paths. This allows and encourages re-use of routing resources for multi-terminal networks, helping reduce routing requirements. Extending this re-use to multiple sinks under multiple conditions, multiple sources under multiple conditions, and even multiple sinks in the same physical locations is not an easy task. It requires carefully filtering the seed routes so they are only re-used under the proper conditions. There are even cases where seeds must be used to avoid false congestion between different sinks of the same net. The modifications needed to effectively handle these cases are explored in detail in Section 8.4.

The next section will explain the predicate-aware routing problem in detail. The problem formulation assumes that the predicate-aware scheduling and placement have been completed as described in the prior chapters.

8.1 The Problem

This section will describe the assumptions, important requirements and observations necessary to develop the predicate-aware routing algorithms. In general, the problem presented to the predicate-aware router is the same pipelined-routing problem as before, with the addition of predicate annotations on the signals that allow the router to share resources between compatible signals. There are three important considerations to take into account when designing a predicate-aware routing algorithm:

- *Each source and sink of a signal is marked with the corresponding operation's condition.* – The router must be aware of the predicates that control any given communication in order to share resources between signals. Whenever two communicating operations both execute, the corresponding signal between them must have a route to ensure the communication happens properly. If only one or neither executes, then the communication is not necessary. This implies that the route need only be complete in iterations where both the source and sink conditions are true.
- *The conditions may differ between each source and sink pair.* – The router will be responsible for managing what conditions under which each portion of a route is reserved, which in turn determines which configurations use a resource. An alternative would be to define routing resources that represent transition points between conditions and have their location defined by the placer. However, this does not allow the router to be as flexible in routing around congested areas because there are more fixed locations that the router must reach. PA-SPR leaves the complexity of managing the routing condition to the signal level router to maintain flexibility. Note that with operations sharing resources, there may be multiple sources and multiple sinks at the same physical resources, so the conditions may differ between sources, the conditions may differ between sinks, and the conditions may differ between any source and any sink.

- ***Multiple sources/sinks may be at the same spatio-temporal location under mutually exclusive conditions.*** – The placement done prior to routing will be predicate-aware, and will allow mutually exclusive operations to share what appears as a single time-slice of a physical device in a modulo schedule. All operation conflicts must be resolved by placement before transitioning to routing, because the router will not be adjusting the placement of operations. This means all co-located sources and sinks must occur under mutually exclusive conditions in order for the co-location to be valid.

In addition to the proceeding considerations, there are some important assumptions required by the solution presented in this chapter. The following is a list of the assumptions that are a result of the hardware execution model or are used to simplify the problem for this initial investigation into providing predicate aware execution in CGRAS:

- ***Limited predicates are available.*** – Only predicate signals routed to the controller for a region are available for switching configurations. At a minimum, the predicates available to a region will be those necessary to support the predicate-aware sharing of devices that the placement requires.
- ***The available predicates will change along the route.*** – A route can cross an arbitrary number of regions, even leaving and returning to the same region. The predicate availability at any point along the route will be dictated by the predicates that are routed to the region controller for the current routing resource. Some regions may have no predicates available.
- ***Predicate availability is fixed.*** – The set of regions acting as sinks for predicate signals is assumed to be fixed within an iteration of the PathFinder algorithm. This assumption eliminates the need for creation and destruction of routes to send predicate signals to region controllers in the inner loop of the QuickRoute algorithm. Between PathFinder iterations, the router may change the set of predicates available to a region, but it may be costly to do so. The router calculates costs incrementally

to reduce routing time. Routing decisions are made using topology and costs attributed to the routing resources. In order to allow predicate-aware sharing, these costs will become dependent upon predicate availability for sharing, and changing the availability of predicates in a region could require recalculation of the costs for all routes throughout the region, or potentially a complete re-route.

- *Routing resources have no side effects other than passing values from input to output, with possible delay.* – The routing resources of an architecture will be signal-transport devices, such as wires, multiplexers, bus connections, registers, and in some cases, ALUs or other compute units set for a pass-through operation. Assuming that use of these routing resources will have no side effects allows the router to speculatively route values, whether they need to be communicated or not.
- *Operations that cannot be speculatively executed will have a distinguished predicate input.* – Operations that have side effects cannot be speculatively executed. The input data flow graph was meant to be spatially executed, so it is assumed that any such operations should have a predicate signal to prevent unintended execution. This predicate input must be distinguished so that the router may ensure that the corresponding signal is unconditionally available to the device.

In addition to these assumptions, there are several implications that are important to be aware of when attempting to solve this problem. The first is that the source and sink condition of any given route cannot be mutually exclusive at the latency of the communication. For example, if A communicates with B with an iteration delay of 2, the compiler can assume that at run-time, there may be a situation where A executes and then B executes two iterations later. A source-sink pair represents communication between operations, and if the source operation never produced a value when the sink operation needed to consume a value, they would never be able to communicate. This fact will be used to trim down the number of source-sink condition relations that need to be accounted for during routing in Section 8.4.3.

The next important observation is that partial promotion is a way to generalize the predicates that govern a route's source and sink to those that are available in the region without resorting to unconditional speculation. Partial promotion was introduced in Chapter 5 as it related to an operation's position in the CDT. Ancestors of condition nodes represent the super-set of executions, so promoting a portion of a route means that the routing resource will be configured to allow the communication more often than is necessary. If the predicate for the current route is not available in the region of a routing resource, then the route can be promoted to execute locally under an available predicate. In the worst case, it may be promoted all the way up to run unconditionally, where no predicate is needed. This worst case unconditional routing corresponds to routing without predicates, so it conveniently reduces to the original SPR routing problem.

Promoting portions of routes to more general conditions does carry an opportunity cost with it. Conditions that are more general are not mutually exclusive with as many other conditions, so there is an impact on the potential for sharing resources. However, this situation is not as bad as it first appears. The limited predicate availability only requires promotion for configuration switching purposes. Promotion is not required for sharing signal level resources such as wiring and registers. Here, the notion of sharing configuration space and physical hardware are split just as they were in Chapter 4 for supporting time-multiplexed routes on resources limited to a single configuration.

Figure 8.1 shows an example where local promotion does not limit sharing. There are two mutually exclusive routes shown, one represented by a red dotted line, and the other represented by a blue solid line, each traveling between a pair of operations represented by red and blue squares, respectively. One could view this as routes for an image processing application, where the red and blue components of an image are processed in separate passes, leading to the opportunity to share the processing resources for the red and blue image components. For this example, assume the data values are being routed from the top left to the bottom right. The CGRA is divided up into regions represented by the light colored gradients outlined by the finely dotted line. The diagonal marks indicate the regions where the red and blue predicates are available. Any areas without the diagonal marks have no predicates available, so everything must execute unconditionally.

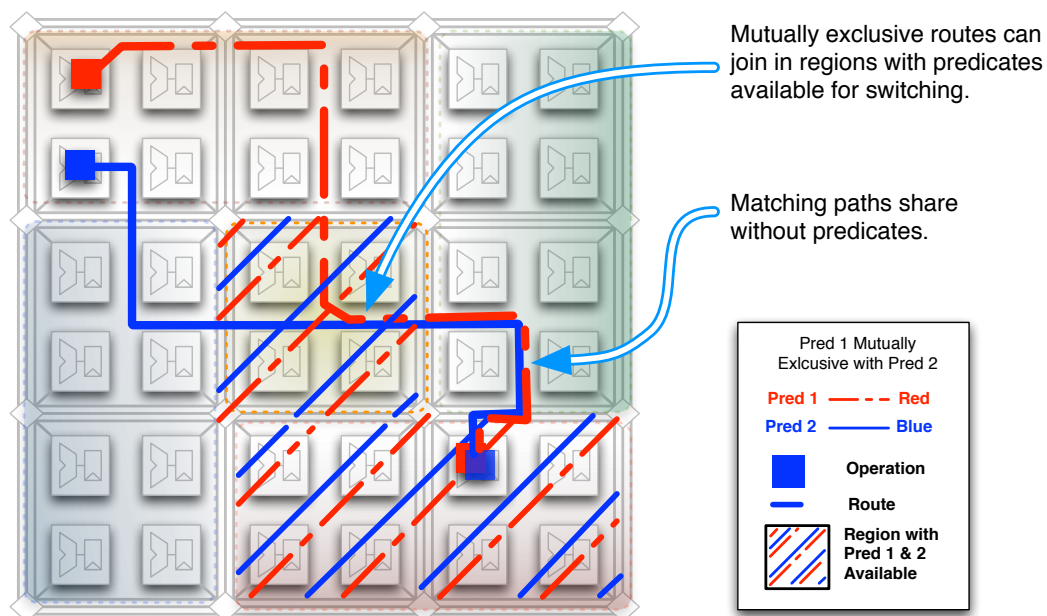


Figure 8.1: Joining and tunneling for mutually exclusive routes.

The time-dimension of the time multiplexing has been left out of the diagram to provide a simplified presentation.

In the upper-left region, the predicates for the two routes are unavailable, so they must both be considered to be routing under the unconditional predicate for control generation purposes. They cannot join routes in this region, because doing so would require a routing mux to switch between two possible routes at run-time based on an available predicate. Once both signals are in a region with the predicates available to switch, they can be joined and route along a common path to the destination. The signals can continue to share routing resources even when the routed path strays into a region without the red and blue predicates, as long as the paths go the same direction, requiring the same configuration. The two routes seem to “tunnel” through the unconditionally configured region. Because they will never occur at run-time together, they will never collide in that tunnel. This example makes it clear that for switching control purposes, signals will need to be promoted to conditions available in the region. For signal level

resource sharing, the promotion is not necessary, and signals can always share resources based on the original source or sink conditions.

Since a route may be promoted to use an ancestor condition, it is logical to consider *demoting* the route locally where an entire set of child conditions are available. Instead of choosing a single condition higher in the tree, is it possible to move down the CDT, distributing a route to execute under all of the predicates at a lower level? Unfortunately, this will not ensure correct execution. While the partition nodes of the tree divide the run-time execution into disjoint subsets, there is nothing to guarantee that they are a complete covering of their ancestor's executions. As an example, pseudo-code and the associated Euler diagram for the space of possible conditions for one iteration is shown in Figure 8.2. Assume that the case statements are fully separated by break statements, and so there is no fall-through. Note that the space for a_m and a_n are completely disjoint and lie entirely within the space for a , but they do not completely cover the space for a . This is because there may be a third case that is not handled, so a_m and a_n would both be false with a true. While at most one child condition of a partition may be true for a given iteration, it is possible that none are true at run-time, even when an ancestor condition is true. If a portion of a route was demoted, the routing resource may not be configured correctly when all the children conditions are false, leading to an incomplete route and failed communication at run-time. This is because the union of the children is not equivalent to the parent.

Signals with source and sink under different conditions present more opportunity for sharing, because the entire route only needs to be complete when both the source and sink operation would execute. Therefore, any portion of the route may be valid under the source condition, the sink condition, or any generalization thereof, to ensure the entire route will be valid when the source and sink actually need to communicate.

8.2 Predicate-Aware PathFinder Costs

The goal of routing is to find a configuration of the interconnect that enables all operation-to-operation communication at run-time. Without predicate aware routing, this task is

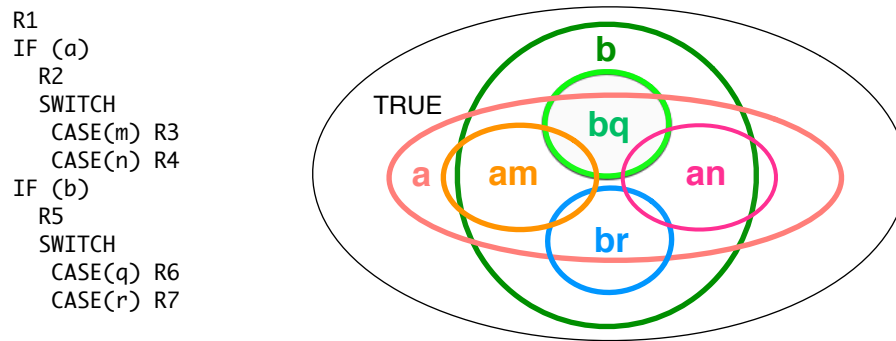


Figure 8.2: Code and Euler diagram of the space of possible conditions.

limited to finding a valid setting for each phase of the II, effectively time-multiplexing the routing resources across phases. As presented in Chapter 4, routing multiple signals through static resources is possible when they all use a common configuration. By viewing resources requiring different settings in different phases as congested, applying a congestion negotiation algorithm will resolve the conflicts and allow routes to share if they use the same configuration on the same device in different phases. By re-defining congestion again, a negotiation-based algorithm can also be used for mutually-exclusive predicate based sharing.

The new formulations of congestion for predicate-aware routing are divided into two categories: signal-based and control based. The signal based notion of congestion is simpler and presented first, in Section 8.2.1. The new notion of control based congestion builds on the formulations used in Chapter 4, and is presented in Section 8.2.2.

8.2.1 Predicate-Aware Signal Congestion

The aggregate execution condition representation used during placement is also used for tracking signal level congestion in the routing. Each unrolled instance of a routing resource maintains an aggregate execution condition for all signals that are routed through it. The aggregate execution condition indicates when there are multiple signals that will require a resource at the same time during execution, and this is used to create a present

signal congestion cost. As each signal is added to the aggregate condition, it incurs a signal congestion cost proportional to the number of existing signals with which it conflicts.

This cost allows for differentiation between resources where there are conflicting signals that can be partitioned into compatible sub-sets and resources where all signals conflict with each other. In terms of negotiated routing, this is done so that signals are routed through resources that have better potential for becoming uncongested by re-routing the fewest signals to alternate paths.

8.2.2 Predicate-Aware Control Congestion

Tracking control congestion in a predicate-aware manner is not as easy as tracking signal congestion. In Chapter 4, control congestion was used to represent two signals conflicting over the required settings for the single available configuration word – specifically, signals in different phases using different routing multiplexer inputs. A similar conflict arises from the limited availability of predicates throughout the architecture. Predicates routed to regions of the architecture are able to select between different configuration words, allowing signals routed under those predicates to use different settings. If predicates that distinguish between two mutually-exclusive signals are not available to the region at run-time, then the signals can still share the routing resource, but only by using the same configuration setting.

In Chapter 4, a table was introduced that tracks control congestion and encourages the same settings for signals in different phases. It is possible to extend that idea to sharing settings across conditions and use the same table structure to track predicate-aware control congestion. In the case of single configuration sharing, the phases are a set of mutually-exclusive conditions under which the operations will execute at run-time. If the condition signals (phase counter bits) are not able to switch the configuration – in this case because there is only one configuration – the configuration must be shared across them. The rest of this section will show how to use the same techniques that allowed sharing of static resources in Chapter 4 to aid in predicate-aware sharing. The difference is that with static sharing, the conditions are defined by scheduling and in predicate-aware sharing the conditions are defined by program control flow.

This sharing is presented in Figure 8.3 with the added context of the CDT. Without control conditions, the CDT consists of the single unconditional node. This is duplicated across the mutually exclusive phases created through time multiplexing. Figure 8.3(a) presents some example routes through a 4-input mux across three phases of time-multiplexing, while Figure 8.3(b) and Figure 8.3(c) represent the dynamic and static control congestion tracking presented in Chapter 4.

Figure 8.3(b) presents control congestion in the original case of fully time-multiplexed interconnect. In the top half of Figure 8.3(b), we see the CDT for this case. There is a new root for the CDT which splits out to the unconditional node for each of the phases of time multiplexing, which will be indicated as Condition 0 \square .¹ In this figure, each of these nodes is fully visible to represent that the phase signal is available to switch between configurations for the routing mux. In the bottom half of Figure 8.3(b), we see the tables that would be used for tracking input usage of the mux. Control congestion happens when multiple inputs are required by signals in the same table. The phase signals are available to switch between configurations, so each phase gets a separate table. In this case, only one signal can be in each table; any more would cause signal congestion (Section 8.2.1) on the mux. Therefore, there will never be any control congestion without signal congestion, so these tables are not needed.

Figure 8.3(c) presents control congestion when there is a single configuration available. This is analogous to not having the phase signals available to switch between different configurations, and so the nodes for the unconditional CDT node in each phase are dimmed. In this case, there is a single table where signals from all phases are tracked for control sharing, and each signal from Figure 8.3(a) is counted in the table. Even though the signals are mutually exclusive in time, they all must share the single configuration available for the mux, and so there is a conflict if they require different mux inputs. The signals in phases 0 and 1 are compatible with each other, but the signal from phase 3 uses a different input. Doing an OR reduction across the phases yields a vector indicating the inputs required. If the sum reduction across this vector is greater than 1, it indi-

¹The box symbol is intentional, and will be use in more complicated examples to provide a visual cue for the condition being discussed. The unconditional case will always be a white box.

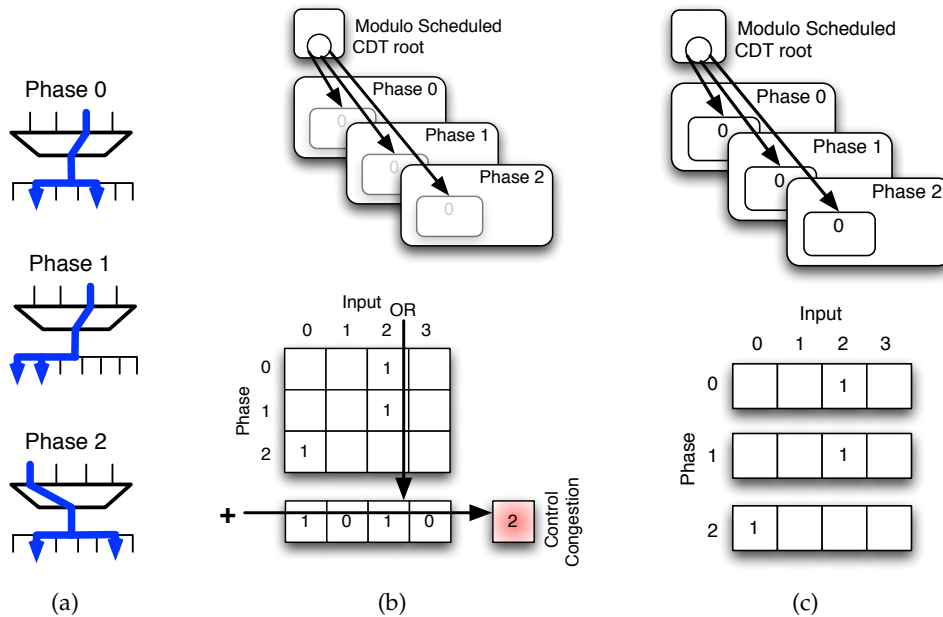


Figure 8.3: Revisiting the control congestion from Chapter 4 with the context of the CDT

icates the need for more than one setting for the single configuration, and there is control congestion.

Now consider the case of signals under mutually-exclusive conditions sharing a single phase of a time-multiplexed mux. If the predicate signals representing those conditions are not available in the region, this is analogous to the phase signals being unavailable when sharing a single configuration slot. It logically follows that to share the single phase configuration, the same table structure can be used as before, with mux inputs forming the columns, but replacing the phases with the conditions that the signals are routed under for the rows.

This observation is powerful because it re-uses existing mechanisms to enable predicate-aware sharing *when the predicates are not available*. This broadens the amount of sharing available in a scalable, distributed design like CGRAs because it means that signals can even share resources in areas of the chip that are beyond the distribution range for the predicate. The local ability to share compatible signals with identical settings will lead

to the global behavior of routes sharing paths in regions without predicates, as shown in Figure 8.1.

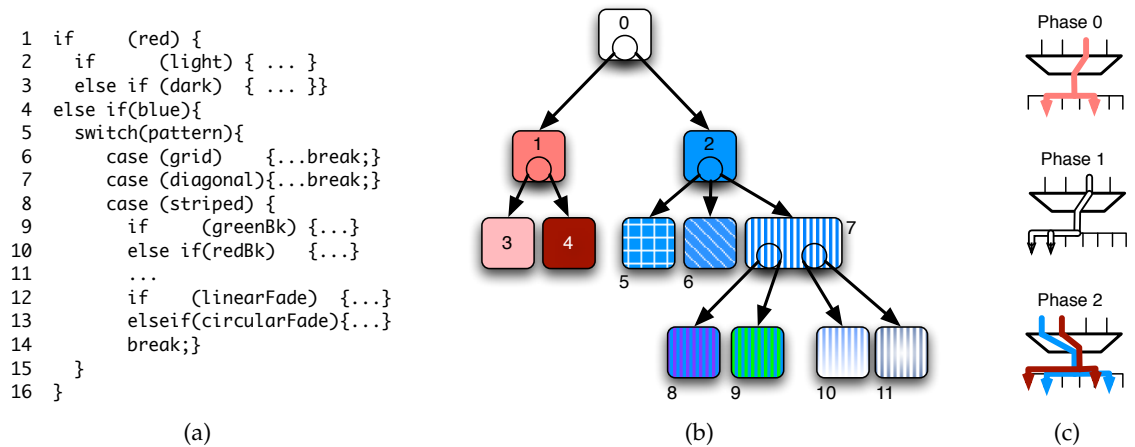









Figure 8.4: An example CDT and routing using texture and color.

A small example that will be useful in exploring this idea is presented in Figure 8.4. A nested `if/case` structure is given in example code in Figure 8.4(a). Each condition represents a color or texture choice. The corresponding CDT can be found in Figure 8.4(b), presented in a compact form. The operations at each level are left out, each condition is represented as a colored/textured rectangle, and partition nodes are represented as small circles within the appropriate rectangle. The conditions are numbered, and will be referred to in the text by both the number and an icon with the corresponding color/texture – for example conditions 1 , 2  and 5 . Finally, Figure 8.4(c) presents a sample set of signals routed through the different phases of a 4-input mux, where the color of the heavy line represents the condition that the signal is routed under. This is similar to the example routes presented in Figure 8.3(a). The signal in phase 0 is routed under condition 1 , the signal in phase 2 uses condition 0 , and in phase 3 the mux is shared by mutually exclusive signals routed under conditions 2  and 4 .

The CDT of Figure 8.4(b) is the raw version generated from Figure 8.4(a) before scheduling. After scheduling, there will be a new root node, and the CDT will be duplicated for each phase. In this case, each phase will be shown as a different layer, as

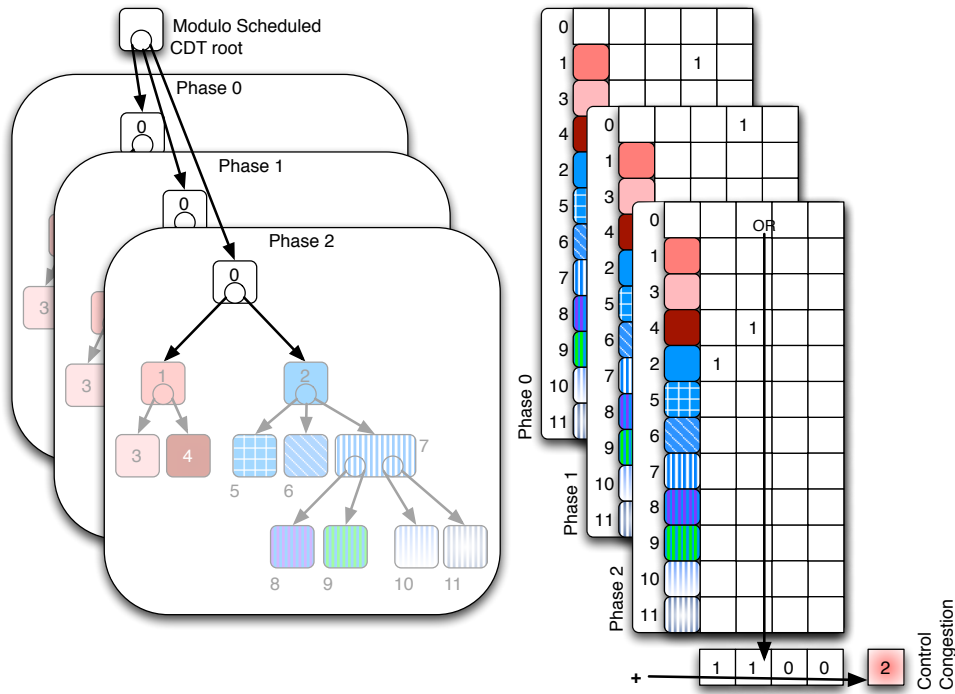


Figure 8.5: Predicate-aware control congestion with only the phase available.

in Figure 8.3. Figure 8.5 shows this for a predicate aware version of Figure 8.3(b). The CDT from Figure 8.4(b) is used, and the phase signals are available in the region, but no predicate signals are available, leaving all but the 0 □ node dimmed. What was only a single entry in Figure 8.3(b) becomes a full table with a separate row for each condition without a predicate available. This is because the tables are used to negotiate between signals that must share a configuration at run-time. The phase signals allow for switching the configuration, so any signals in different phases are accounted for in separate tables, yielding one table per phase. Each signal routing in a condition from a specific phase will have to negotiate with other signals from conditions *in the same phase*. Thus, each phase gets a separate set of rows for each phase-duplicate of the conditions in the CDT. In Figure 8.5, the example routes from Figure 8.4(c) are accounted for in the appropriate tables, and it is clear that even though the two routes in phase 2 exist under mutually exclusive conditions, there is no predicate to switch the input at run-time. This is indicated

after running the reductions across the tables, leading to control congestion value of 2 for phase 2.

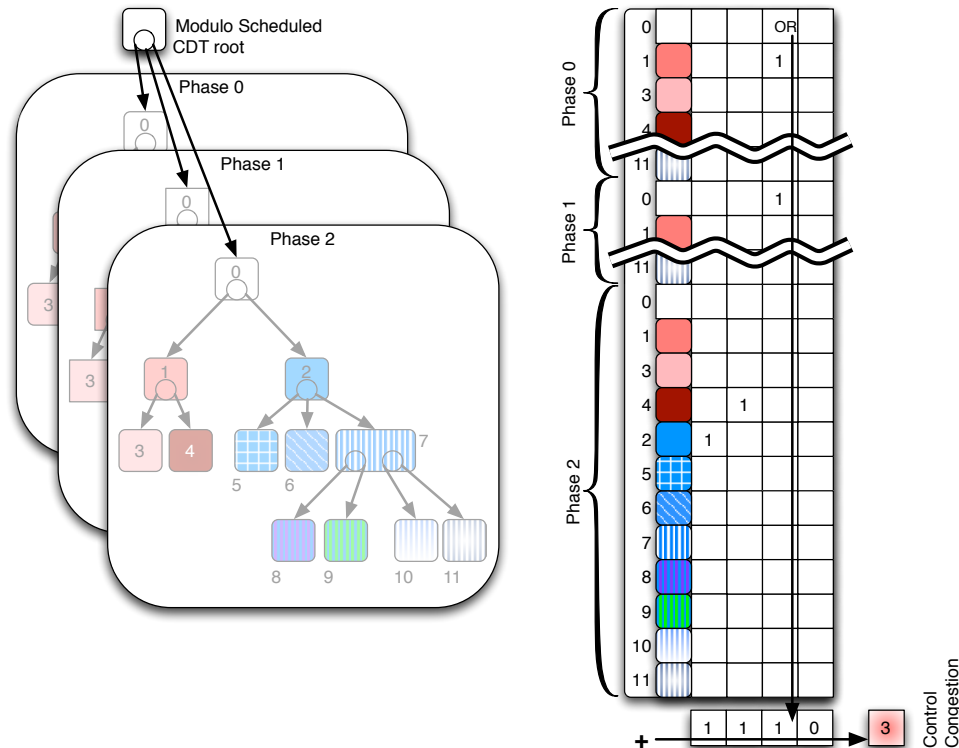


Figure 8.6: Predicate-aware control congestion without the phase signals available.

Similarly, if the phase signals are also unavailable, the tables can be joined into a single table representing the single configuration word, as was done in Figure 8.3(c). This is illustrated in Figure 8.6 where the condition 0 \square nodes for each phase have been dimmed, and the three tables have been concatenated into a single long table. Now all four signals from the example routes in Figure 8.4(c) will be negotiating for the single configuration, and the reductions across the table lead to the larger control congestion measure of 3.

Tracking control congestion in this manner will work when there are no predicates available in the region; however, the routing problem to be solved allows regions where a limited set of predicates are available. If a particular predicate needed for sharing is

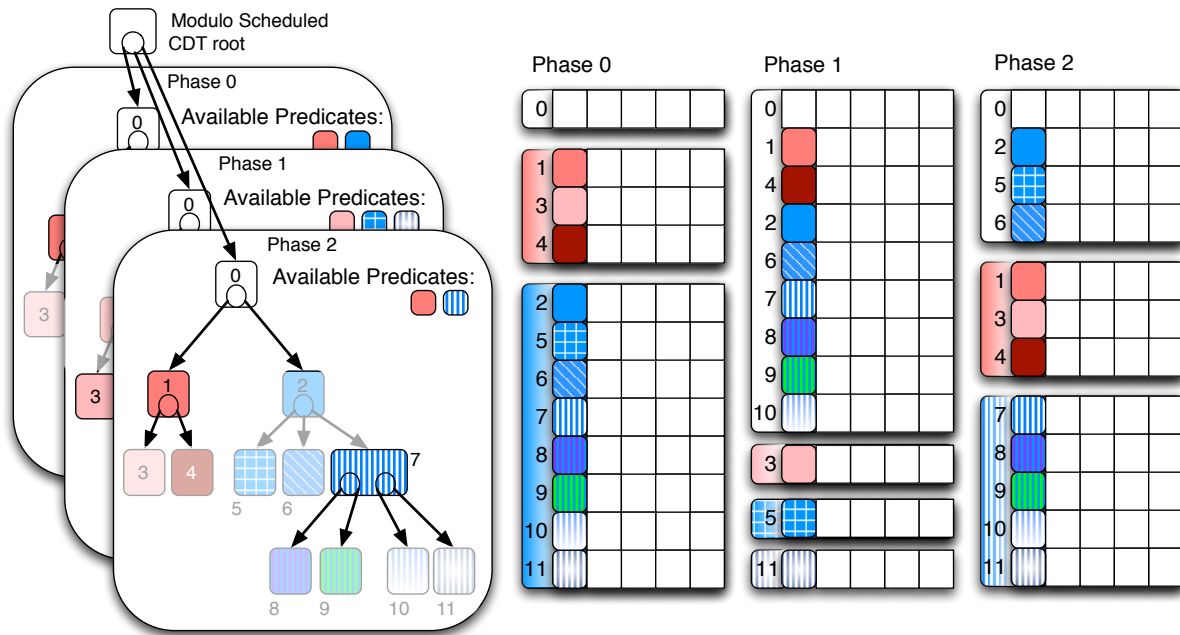


Figure 8.7: Predicate-aware control congestion with tables for partial promotion with limited predicate availability.

not available in the region, the notion of partial promotion can be used to find a valid predicate in the region to use in place of the original. This promotion is only done for control congestion purposes, as the signal congestion calculation can use a condition whose predicate is not available within the region. That is, if two signals are routed under mutually exclusive conditions, when the signal congestion is calculated, it always uses the original routing condition. However, when calculating control congestion, the router will promote the condition to one that is available in the region before computing congestion.

For control purposes, the signal is promoted to the closest (most specific) ancestor condition that has a predicate available in the region. This promotion moves a signal up past partition nodes that do not have the predicate signals available to enable the corresponding partitioning of configurations at run-time. Signals that must negotiate to use the same configurations can easily be identified in this way because they are promoted up to the same conditions.

The use of the CDT guarantees a hierarchical structure to the run-time conditions, which means that ancestor conditions will be true whenever the descendant conditions are true, and possibly more. Thus, if a signal route is valid under the more general condition of the ancestor, it will always be valid when the condition it is actually being routed under is true. This form of partial promotion will be referred to as finding the most precise ancestor. If there are no ancestor predicates available in the region, the router will simply promote all the way to the unconditional case. Each available predicate in the region, plus the unconditional case, represents a valid partial promotion target for signals routed through the region.

In the first two examples, each table had a one-to-one correspondence with a configuration available in the architecture. However, in this more complicated case with partial promotion, each predicate available in the region corresponds to a table. Each phase can have a separate set of predicates available, and so the partitioning of conditions into tables will be different per phase. This is because the available predicates represent a way to switch the configuration at run-time, and this switching allows us to partition the control negotiation into separate tables. An example with limited predicates available and the resulting tables per phase is shown in Figure 8.7. In this example, the predicates that are available in each phase are shown in the upper-right hand corner of each phase in the CDT. The corresponding tables are shown on the right-hand side of the diagram. The large tab on the left of the table is colored according to the available predicate it represents, and each condition that will be promoted to that table has a row.

Instead of a single configuration word, each promotion target corresponds to a set of configuration words, one of which will be selected when the predicate is true, and correspondingly to a set of configuration words that must be programmed for any signal routed under the condition for that predicate. This predicate available in a region plus its expansion to a set of configuration words will be called a *configuration slot*. Examples to illustrate this are shown in Figure 8.8. On the left of the diagrams is a table of the possible addresses generated by the predicate signals. The phase portion of the address is left out for simplicity. Because certain sets of predicates will be mutually exclusive with each other, there are certain combinations of predicates that will never occur, and

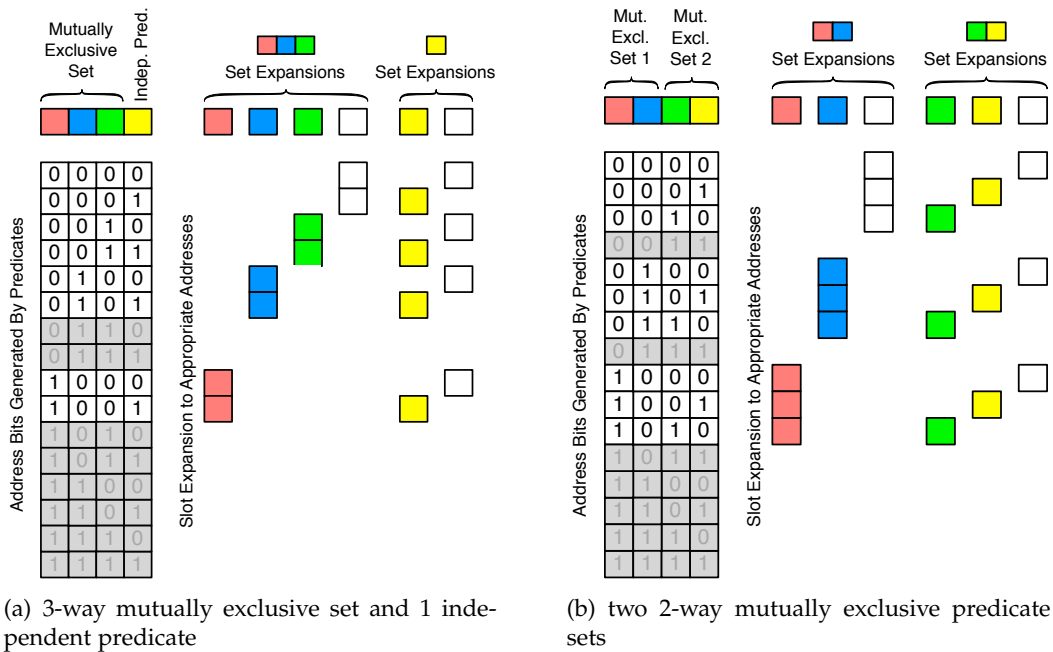


Figure 8.8: Configuration slot expansion to configuration words.

their corresponding addresses will never be accessed at run-time. These addresses are grayed out in the diagram. The grayed out addresses illustrate one of the drawbacks of our simple hardware model. Some ideas for reducing this waste are discussed in Chapter 10. Each predicate represents a configuration slot that was negotiated for with a table, and once that table is congestion free, it represents the value that should be stored. The right side of the diagram shows the addresses where each condition is true, marked by a colored box. This corresponds to where the configuration value should be stored in the configuration memory.

Before moving on, I will show that tracking congestion at the slot level – that is, a table per predicate that is available in the region – will lead to correct routes. Conceptually, the control congestion term allows PathFinder to negotiate for sharing configuration words. However, having a separate table for each possible configuration word allocated to a routing device will lead to a memory footprint that scales linearly with the maximum

supported initiation interval of the hardware, instead of the scheduled initiation interval of the application. Instead, congestion is tracked per configuration slot and the final configuration settings are generated by expanding the slot configuration to the configuration words that it represents. This reduces the memory footprint from one table per configuration word (or maximum Π of the architecture) to one table per predicate. Predicates are used to choose configuration words, so this can provide a logarithmic decrease in memory usage assuming predicates are directly used as address bits.

To show that the router will still either produce a legal mapping or fail due to unresolvable congestion even when using only one table per predicate, I need to exploit the separation of control and signal congestion. Signal congestion is tracked by the aggregate execution condition, just as it tracks compatibility for operations in placement. To ensure a valid final mapping, the tables created need to indicate there is no control congestion only if a valid mapping can be made or there is other congestion to indicate an invalid mapping. Extra aliasing of the signal congestion may happen, but this will not affect the validity of the final mapping, and so it is allowed. Because the aggregate execution condition will track the signal congestion, we can assume without loss of generality that all signal congestion has been resolved and the control congestion tables are for negotiating between a set of mutually-exclusive signals.

There are three possibilities for where each of these mutually-exclusive signals are counted for control-congestion purposes on a given resource:

1. If the predicate for the condition the signal is routed under is available in the region, there will be no partial promotion of that signal. The signal will be accounted for in the table of the slot corresponding to the signal the condition is routed under, and it will be the only signal using that slot's configuration. Any other signal in the table would have to be routed under the same condition, or have been originally routed under a descendant condition and promoted to the slot's condition. For both of these cases, this hypothetical second signal is not mutually-exclusive with the first, by construction of the CDT, and would cause signal congestion. This violates the

assumption that there is no signal congestion, so there cannot be a second signal, and so the single signal may choose any configuration it requires.

2. If the predicate is not available, and no ancestor predicates are available, the signal will be counted in the unconditional table, corresponding to the top node in the original CDT. This may have happened for several signals, but they must be mutually exclusive once there is no signal congestion. This table will track the control congestion between them until they are using the same routing setting, at which point they can safely share the resource.
3. If the predicate that a signal is routed under is not available, but it has been promoted to the predicate of an existing slot, it will be counted in that slot's table. This case is similar to that of the unconditional table, where multiple mutually-exclusive signals are promoted to the same slot. Again, the slot's table will track the control congestion between them until they are using the same routing setting.




















It is clear for all three cases that any signals in a single slot will be able to share the resource once there is no congestion. Finally, I need to show that once there is no congestion, signals using different slots will never require conflicting configuration settings, and at run-time the appropriate configuration setting will be used. This requires a method to map from the configuration setting that is used in a slot to the configuration words that will be programmed with that setting.

There are three possible run-time relationships between slots that need to be considered to decide if conflicts will ever happen. These relationships are based on the relative positions of the conditions of each slot in the CDT. The first possibility is that the conditions of two slots are mutually exclusive, and so the setting of both slots will never be needed at the same time. The second possibility is that the condition of one slot is the ancestor of the condition of the other. Here, the condition of the ancestor is true in a super-set of the executions where the descendant is true, leading to possible times when both settings will be needed at run-time. The third possibility is a catch all-for everything else, so both settings may or may not be needed simultaneously at run-time.

Case 1 Non-intersecting: Assuming that there was no partial-promotion needed and that there is no signal congestion, the signals mapped to a given resource will be mutually exclusive and fall into mutually-exclusive slots. Even with partial-promotion, if none of the signals were promoted past the partition node separating them from the others, then all of the slots with signals will still be for mutually exclusive conditions. Each slot will be expanded to the configuration words where the slot's predicate is true, each of the mutually-exclusive predicates are false, and all possible settings of the remaining predicates. The setting for a slot will be written to all of the configuration words of a slot's expansion. A different predicate bit must be true for each slot's set of configuration words, so each set will be disjoint and there will be no conflict for setting any given configuration word. At run-time, whenever the routed signal's condition is true, the corresponding predicate bit will be set and the appropriate setting will be selected from the set of configuration words.

Case 2 Super-set: If there was some partial promotion, a signal may have been promoted up past the partition that separates it from other mutually exclusive routes, leading to an ancestor-descendant relationship between two slots with signals in them. A simple example of this is when only the `else` predicate of an `if-then-else` construct is available in the region, and there are routes under both the `then` and `else` conditions going through the same multiplexer. The route under the `then` condition will be promoted past the partition up to the next ancestor with an available predicate, or the unconditional case if none is available. Since the `then` route was promoted past the partition, there is now an ancestor-descendant relationship between the slots that have routes in them, even though the signals themselves are mutually exclusive. The ancestor predicate is not mutually exclusive with the descendant, leading to an overlap of the configuration word expansions of the two slots.

The descendant slot is more specific, so the configuration settings for the descendant should override the configurations for the ancestor. Partial promotion for routing stops at the most precise ancestor, so any routes in the ancestor slot are guaranteed not to be from a condition below the descendant slot. The promoted routes cannot be from a condition

above the descendant slot, otherwise they would not be mutually exclusive with routes that were promoted to that descendant slot. Any promoted routes must be from branches that are mutually exclusive with the descendant slot. Therefore, whenever the descendant slot condition holds true at run-time, it is guaranteed that the original conditions for routes promoted to the ancestor slot must be false, and the configuration can be safely set by the descendant. Any remaining configuration words in the expansion of the ancestor should be set according to the configuration of the ancestor slot. An example is shown in Figure 8.9. Consider a set of signals routing under Conditions 5 , 6 , 10  and 11 . They are all mutually exclusive, but the predicates available for switching them are 2 , 10  and 11 . The routes for 5  and 6  will be promoted up to 2 , who's slot expansion conflicts with that of 10  and 11 . Because 10  and 11  are more specific, their configuration will be written to any locations where a conflict with 2  exists. This will ensure that the appropriate setting is used at run-time for these routes. Note that 5  and 6  are both promoted to 2 , and there is no predicate available to distinguish them. This means they will negotiate to share the same configuration, and this is what will be written in the non-overridden 2  configuration words.

Case 3 Partial-overlap: The final case is one that cannot happen once all of the signals on a resource are from mutually-exclusive conditions. Any pair of signals being routed through a single resource will be mutually exclusive if there is no signal congestion. This means they started in mutually exclusive conditions and upon promotion, one of three things may happen:

- They will be promoted to slots that are below the original partitions and thus are mutually exclusive.
- They can both be promoted past all partitions to the same slot.
- One may be promoted past all partitions to an ancestor slot of the other.

Each of these three possibilities is covered by the prior two cases – non-intersecting slot expansion and super-set expansion – and so once all signal congestion has been

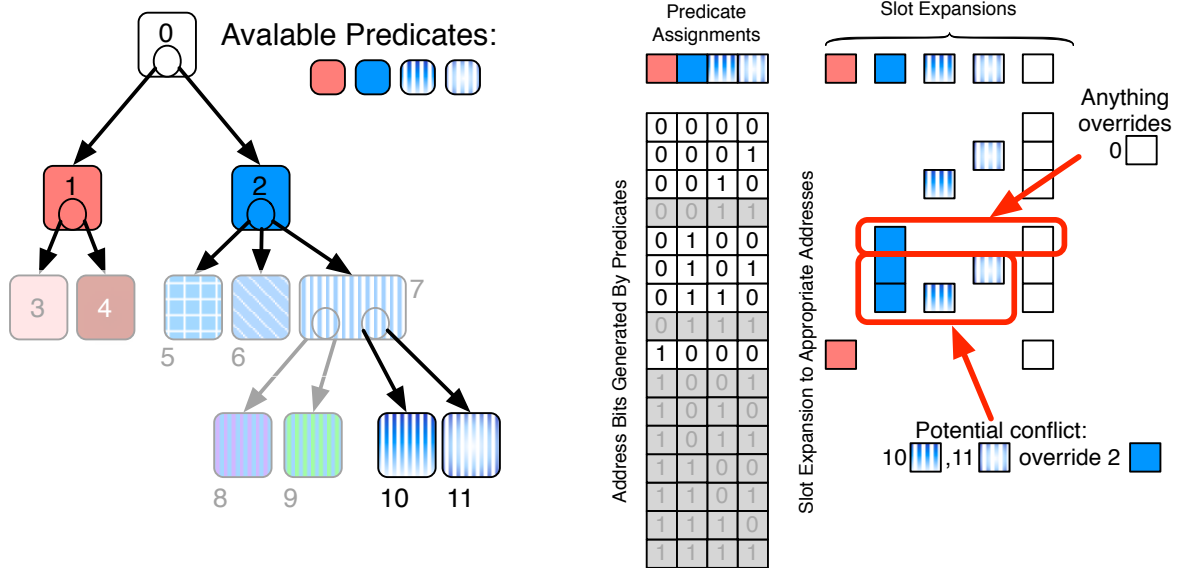


Figure 8.9: Child slot override when there are partial promotion conflicts.

resolved there is simply no way for there to be a partial-overlap in the configurations that need to be written for different signals on a device. In summary, to generate a valid configuration from congestion free slots on a device, each slot is expanded to the set of configurations where that slot’s predicate is true, any mutually exclusive predicates are false, and all other predicates are set to all possible values – for ancestor predicates this may be reduced to only the true case. For any pairs of slots that form an ancestor-descendant relationship, the setting for the descendant slot is used. This will ensure that at run-time, the appropriate configuration setting is chosen for any signals that need to be routed under the current run-time conditions. Notice that the same configuration is duplicated for all possible values of the predicates unrelated to the slot used by signals on the device. This duplication is used to ensure that the configuration for the current device is constant while the predicates used for sharing on other devices can change their configuration.

As a final note, it is important to remember the examples presented here have been simplified for presentation. Each phase only shows one copy of the CDT, where there

is actually a full copy per wave of the schedule, each of which is not mutually exclusive with the other copies in the same phase. This is the same simplification that was made when the time-based duplicates of the CDT were introduced in Figure 5.5. All of the same reasoning still applies; there is just a larger tree in each phase based on the length of the schedule.

8.2.3 Negotiated Routing Costs

The last two sections covered predicate-aware signal and control congestion in PA-SPR. These represent how problematic situations are tracked during the routing. This congestion must be turned into present sharing and history costs to be used by PathFinder to accomplish the negotiation for routing resources. The control congestion uses the same representation as presented in Chapter 4, and so the same costs will be used as well. The signal congestion now is based on the compatibility of multiple signals routing through the same mux at the same time, and so the costs need to be reconsidered.

In the regular SPR implementation, no signals can share a routing resource in the unrolled graph, so the amount of congestion is directly used as the present sharing cost. This is the number of extra signals that are trying to use the resource.

In PA-SPR, the amount of congestion is not as simple to calculate. If two compatible signals are on the same resource, that should not incur any present sharing. If two incompatible signals are on a resource, it should incur a present sharing cost. If the router is trying to decide between sending a signal through a resource that has one compatible signal and two incompatible signals, or a resource that has three incompatible signals, the cost should steer the signal toward the resource with only two incompatible signals. To gain this level of discrimination, the signal present sharing cost in PA-SPR is calculated as the number of edges in the conflict graph between signals using the resource. The history cost update remains a single increment for any resources that are congested at the end of an iteration.

8.2.4 *Simple Optimizations*

There are a couple of simple optimizations that have been added into the implementation of PA-SPR over the course of development and are briefly described here. The first is a partial-promotion cache for accelerating the lookup of the appropriate slot given a signal's condition. The second is a method of lazy construction for the slot tables to avoid excessive memory usage.

Once the router knows the set of predicates that are available in a region, it can determine the partial promotion used for control congestion of any given condition. This partial promotion is used to find the appropriate table for tracking congestion and computing routing costs every time a node is reached in the routing process. Instead of tracing through the CDT every time this happens, PA-SPR retains a hash based map from all conditions to those available in a given phase and region. This map is used to provide amortized constant time translation between the condition a signal is currently routed under and the condition it should be promoted to due to predicate availability. This optimization simplifies and accelerates the innermost loop of the routing process.

The control congestion tables used for each slot require a separate row for every condition that maps to that slot via partial promotion. There can be far more conditions than were in the original CDT once scheduling information is taken into account. There is a virtual duplicate of the CDT for every single start time in the schedule. This leads to many conditions. Instead of pre-allocating the rows for all of those conditions, they are only added to the slot tables of routing resources as a signal is routed through the resource using that condition. As implemented, there is no "garbage collection" to remove rows once they are empty, since that may result in a loss of the history values for that row, and thus it should be avoided if possible. However, when a new row is added, the router must create new values for the history costs of that row. For each input of the new row, the minimum value across all of the other rows is used for initialization. This will capture the relative contour of the existing history, so that signals under the new condition will be guided toward agreement with the signals that have previously used the resource.

8.3 Routing Predicates

The predicates are datapath signals themselves, and must be routed as well. There are two approaches to routing predicate signals that may have drastic effects on the applicability of mutually exclusive sharing.

The first is to only route predicates to locations they are needed by the placement before routing starts. This allows the router to include the routes of the predicates with all of the rest of the signals by setting up their routing requirements between placement and routing. However, this does not allow the router to change the predicates available in a region based on routing need. This means that regions without any sharing based on placed devices will not have predicates available for sharing routes.

The second is to allow the router to request predicates that are not available initially in a region. This means that the routes for predicates will be determined dynamically during routing based on other routes, complicating the routing algorithms.

Limiting the predicates to those required by the placement is more problematic than it may seem at first. The placement tries to avoid over-subscribing region gateways by not requiring predicates when they do not enable sharing of a device in the region. This can greatly reduce the required predicates, but it also means that they may not be available in the region in the cycles preceding and following the cycle where operations are shared. These are important clock cycles for sharing routes under the same predicates. Initial investigations indicated that when the router could only use the predicates directly needed by shared placements, routing would congest and fail due to a lack of predicates in the cycle just before or after the sharing operations that needed the results. This motivated including a predicate request mechanism in the router to provide the sharing that would alleviate such congestion.

8.3.1 When to Request Predicates

There are several levels of granularity that predicates could be requested at, corresponding to the depth of the algorithmic loop that the request happens in. At the finest granularity, the router could attempt to route different predicates to regions at each exploration

step in QuickRoute. As a signal goes through a mux, if a needed predicate is not there, we could try to route it and roll this into the cost of that resource on the route. This would require significant work in the innermost loop of the router, in addition to creating very complicated interactions with the QuickRoute algorithm itself. This is likely impractical in both coding and execution complexity.

Another approach would be to attempt to route required predicates for a signal with conditional dependence right after the signal itself is routed. This way, the signal itself has tried to find the cheapest route without requiring any new predicates. Routing the predicate signals after a PathFinder iteration to only the places they are actually needed should lead to significantly less routing. The savings would be highly architecture dependent, as the router may need to explore more in an architecture with broad, uniform cost than an architecture with highly variable costs for different routes.

Coarsening the granularity further, the router can attempt to re-allocate predicates for routing after some number n iterations of the PathFinder algorithm. If $n = 1$, the requirements for predicates are unlikely to change unless a conflict was created by moving one signal and later relaxed by moving a second within the same iteration. For $n > 1$, n may either be a fixed value or an adaptive choice made during routing. One possible extreme choice for adaptive re-routing is to only attempt to change predicate availability once other options are exhausted. This happens when the remaining congestion is only control congestion due to not having predicates available for sharing. In this case, the predicates needed and the regions that need them are clearly indicated by the congestion. If it is possible to route the needed predicates, then congestion will be immediately resolved. Even in cases where the congestion is not entirely due to lack of predicate availability, sending predicates to areas with high history costs in slots that could be split by providing that predicate could generally ease routing and allow other areas of congestion to be relieved.

Depending upon the relative costs of different types of congestion, the routing may never reach a point where all congestion is limited to control congestion. Additionally, routing predicates to a region does not always simply reduce the control congestion. It may also increase both the control and signal congestion, because now there is more signal-routing overall that needs to be accomplished. Immediately following the intro-

duction of new predicate routes, the history costs will not reflect the congestion due to these new routes. It may take several iterations for appropriate history costs to build to adapt to the change in the routing problem. For these reasons, the implementation of routing in PA-SPR requests new routes after i initial PathFinder iterations, and then again every n PathFinder iterations after that, where i and n can be set in the configuration file. The values used here are $i = 5$ and $n = 10$. PA-SPR only requests predicates based on routing resources that only have control congestion. This allows the router to use new predicates to alleviate some of the congestion in areas that will clearly benefit from it. For resources that include signal congestion as well, some subset of the signals will need to be re-routed through other resources to clear the congestion. This re-routing may clear the control congestion as well, so the router should not prematurely waste region-wide predicate resources attempting to resolve control congestion where there is also signal congestion.

The modified predicate-aware PathFinder iterations used in PA-SPR is given in Algorithm 8.1. Within the `while` loop, the first `for` loop represents the usual PathFinder execution. The second `for` loop represents the new procedure for adding new predicate routes as they are needed to each region for every phase. This is guarded to only execute every n th iteration, with some offset number of initial iterations i . This describes *when* new predicate routes will be chosen, the next section will cover the inner portion of the new `for` loops, which handles *choosing* the new predicates.

8.3.2 Finding Needed Predicates

There are two interesting and related choices to make when creating new routes that will provide predicates to regions. They both center around the question “What is the most *needed* predicate?” At the most abstract level, the most needed predicate is the one that will lend the globally optimal amount of flexibility in the routing – yielding the best route possible. This is handled by PA-SPR by separating the process into two separate problems. The first is deciding which predicates are *needed*, and the second is finding a way to decide which subset will provide the most flexibility. This section covers the first, followed by the second in the next section.

Algorithm 8.1: Predicate Aware PathFinder

```

1 begin
2   Clear sink and visit marks
3   Setup routes for predicates to regions as required by the placement
4   while iterate do
5     for routeable signal do
6       if congested then
7         rip-up signal
8         re-route signal
9     if after i iterations, on every nth iteration then
10      for region do
11        for phase do
12          for routing resource do
13            Find control only congestion
14            Determine needed predicates and relative ranking
15            Choose predicates based on ranking and region capacity
16            Setup new predicate routes as capacity permits
17      iterate  $\leftarrow$  isCongestion  $\wedge$  (iteration < limit)  $\wedge$  progress
18 end

```

To make tackling this problem more tractable, the router begins by using a local view. Each routing resource will have a set of congestion tables, as described in Section 8.2.2, that is responsible for tracking sharing. Each table will be tracking sharing for a different slot representing each predicate available in the region. As mentioned previously, if the router is attempting to choose new predicates to route to the region, the only congestion on the resource will be control congestion.

In each slot, there will be entries for signals under multiple conditions, one row per condition. The congestion on that resource would be fully resolved if the predicate for each occupied row were successfully routed to that region. Each of those rows would then be given a slot of their own. There could be a way to choose predicates that would partition the rows to eliminate the congestion using fewer predicates, but for the initial implementation, all predicates for occupied rows are used. When this is done across all routing resources in a region that only have control congestion, it provides a list of

predicates that could be used to resolve all of that congestion. However, this may require more signals in a region than there is capacity to support, so choosing a subset is the topic of the next section.

8.3.3 Selecting Predicates with Capacity Limitations

The previous section demonstrates a way of obtaining a set of predicates that could reduce congestion in the region if successfully routed. However, each region gateway has a finite, likely small capacity for predicates. Given only a set of predicates that will reduce control congestion and a capacity limit for the region, it is difficult to decide on the subset that should be chosen for routing.

Instead of simply collecting the set of predicates, the router can count the number of resources that need each predicate. The predicates with the highest counts will resolve the most congestion if they are successfully routed, so the predicates are ranked by this count. Also note that a predicate can only resolve the congestion if it can be successfully routed to the region. The router already has access to an architecture-specific plug-in that estimates routability and routing cost. Applying this estimator to the potential predicate routes allows the router to eliminate predicates that cannot be routed to the region due to either latency or connectivity constraints.

After this ranking and filtering process, the router now can choose the top predicates from the list, up to the capacity of the predicate gateway. The router assigns these new predicates to un-used ports in the gateway and makes the appropriate updates so they will be routed on the next iteration. Additionally, all of the congestion tables in the region need to be re-built to reflect the new predicates. Predicates are only added, creating new slots with their own congestion tables. This process requires creating the new tables and splitting off the row entries of the existing tables and re-assigning them to the appropriate newly created tables. Additionally, the mappings in the promotion cache need to be updated. This process can be expensive, so it is much better that it is done once every few PathFinder iterations, rather than once for every visit of a congested node.

8.4 Predicate-Aware Signal Tree Routing

With the topics of predicate-aware PathFinder and routing predicates covered by the previous sections, this section will detail the final major change – making the signal tree routing predicate aware. Fortunately, the predicate-aware nature of the congestion costs provided by PathFinder allows the core QuickRoute algorithm to remain largely unchanged. The minor changes needed are covered in Section 8.4.1 using the context of a single route where the source and sink both execute under the same condition. However, the problem becomes more complex when multi-fanout signals are coupled with the case of source and sink executing under different conditions. The methods used to deal with this complexity are described in Section 8.4.2.

8.4.1 QuickRoute

The single fan-out pipelined routing problem is stated in [LE04] as: “Given a source node and a sink node in a directed circuit graph and $N > 0$, find the shortest path from source to sink that includes exactly N registers.” The predicate-aware variant is: Given a source node, source condition, sink node and sink condition in a directed circuit graph and $N > 0$, find the shortest path from source to sink, using conditions that will be true at least whenever both source and sink condition are true, that includes exactly N registers.

The changes to QuickRoute to create the predicate-aware version are outlined in Figure 8.10(b), with the original algorithm listed in Figure 8.10(a). The first change is a choice of condition to route under as each routing resource is explored, as seen on line 12 of Figure 8.10(b). The choice requires knowledge of the source and sink operation conditions in order to ensure the route is done under a condition that will ensure the proper configuration whenever the communication between the two operations should happen. When the same condition is given for both, that is the condition that is chosen. That is the most specific valid condition for the route, so it will afford the most sharing opportunities. When a different condition is used at the source and sink of a route, one of the two will be chosen along the way, but making the correct choice is a difficult decision. An alternate condition could be chosen using partial promotion, but that would only increase the possibility of

QuickRoute (source, sink, N)

```

1 for all nodes n,
  for all  $i \leq N$ ,  $n.visited[i] = 0$  do
2 initialize the priority queue PQ
3 insert the path [source] into PQ
4 while PQ is not empty do
5 remove the shortest path P from PQ
6 if P.end == sink and P.latency == N then return P
8 else
9 for every neighbor n of P.end at latency < N do
10 if  $n.visited[P.latency] < k$  and
11 n is not in P then
13 add new path  $P' = [P, n]$  to PQ
    with cost = P.cost + n.cost
14 increment  $n.visited[P.latency]$ 
15 endif
16 endFor
17 endif
18 endwhile
19 endForAlls
20 return failed

```

(a) QuickRoute of [LE04]

PAQuickRoute (source, srcCond, sink, snkCond, N)

```

1 for all nodes n,
  for all  $i \leq N$ ,  $n.visited[i] = 0$  do
2 initialize the priority queue PQ
3 insert the path [source] into PQ
4 while PQ is not empty do
5 remove the shortest path P from PQ
6 if P.end == sink and P.latency == N then return P
8 else
9 for every neighbor n of P.end at latency < N do
10 if  $n.visited[P.latency] < k$  and
11 n is not in P then
12 n.cond = chooseCond(srcCond, snkCond)
13 n.cost = cost(n, n.cond)
14 add new path  $P' = [P, n]$  to PQ
    with cost = P.cost + n.cost
15 increment  $n.visited[P.latency]$ 
16 endif
17 endFor
18 endif
19 endwhile
20 endForAlls
21 return failed

```

(b) Predicate-Aware QuickRoute

Figure 8.10: Comparison of original and predicate-aware versions of QuickRoute.

conflicts, as it pushes up past partition nodes in the CDT. The choice between routing under source or sink condition will be explored further in the next section.

The second change is shown on line 13 of Figure 8.10(b). This change computes the cost of using the routing resource based on the chosen condition. This represents retrieving the cost according to the current predicate-aware PathFinder costs. The chosen condition will be used with signal congestion tracking to indicate whether there will be any conflicts with other signals routed through that resource. The chosen condition will also indicate the configuration slot that should be used for computing the control congestion cost. The translation process from the chosen condition to the appropriate slot represents a partial promotion to a condition that is available in the region. This means routing can proceed through different regions with varying predicate availability without needing to take that into account in the chooseCond procedure of PAQuickRoute.

With these two changes in place, the costs used by QuickRoute will be predicate aware. This allows QuickRoute to take advantage of the sharing allowed by predicate-based con-

figuration switching through the control congestion costs. The run-time mutual exclusion that supports the tunneling illustrated in Figure 8.1 is supported through the aggregate execution condition based signal congestion costs. However, QuickRoute is only formulated to handle a single source to sink route. In SPR, this limitation is overcome in PathFinder by stitching together multiple single-source routes into a tree for signals with multiple destinations. In addition to doing this stitching in a predicate-aware manner, PA-SPR must allow multiple signal trees both starting and ending at the same location due to operations sharing a device (e.g. Figures 8.11-8.13). The next section covers the methods for handling these complications.

8.4.2 Predicate Aware Multi-source/-sink Routing

In the description of predicate-aware PathFinder given in Algorithm 8.1, a significant portion of the PathFinder and QuickRoute integration is covered by the simple statement *re-route signal*. The PathFinder/QuickRoute integration used in original SPR is shown in Algorithm 8.2, where QuickRoute replaces the Dijkstra's shortest path/A* search from the original PathFinder formulation. The *re-route signal* step has been expanded to show the per-sink routing loop, with s_i representing the source of the route, t_{ij} representing the sinks that are ordered by their estimated distance from the source, and c_n representing the PathFinder cost for node n .

Note that an initialized priority queue is provided to QuickRoute instead of simply specifying a source and sink pair. For brevity, the portion of a signal that needs to be routed between a particular source-sink pair will be called a fanout. The priority queue initialization is important for routing multi-sink signals. If each sink were treated as a separate fanout, the first sink to be routed would occupy the resources immediately following the source device. The next sink would need to use these resources as well, and the costs would reflect false congestion with the first fanout route.

An important consideration now that resources are being shared is whether or not signals from different operations that are sharing the same device will interfere with each other with similar false congestion costs. For the simple case of a pair of single-condition routes from a pair of operations sharing a resource destined for the same sink resource,

Algorithm 8.2: PathFinder (Negotiated Congestion)/QuickRoute

```

1 begin
2   while iterate do
3     for routeable signal do
4       if congested then
5         Rip up routing tree  $RT_i$ 
6          $RT_i \leftarrow s_i$ 
7         while  $\exists$  unrouted sink  $t_{ij}$  do
8           Initialize priority queue  $PQ$  to  $RT_i$  at cost 0
9           Run QuickRoute search for  $t_{ij}$  using  $PQ$ 
10          for all nodes  $n$  in path  $t_{ij}$  to  $s_i \notin RT_i$  do
11            Update  $c_n$ 
12            Add  $n$  to  $RT_i$ 
13        iterate  $\leftarrow$  isCongestion  $\wedge$  iteration  $<$  limit  $\wedge$  progress
14 end

```

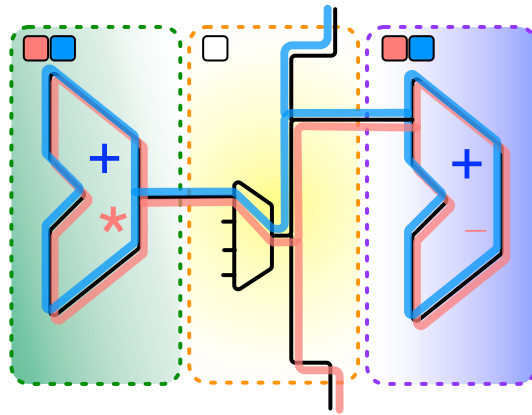


Figure 8.11: Compatible signal tunneling.

the conditions of the fanouts will be set to match the operations. Since the operations are mutually exclusive, the signals will be as well. If the signals follow the same path or diverge, they will not interfere through congestion. An example of this is illustrated in Figure 8.11. The green, gold and purple dotted areas represent separate regions, while the red and blue tint represents operations and the associated signals routed under mutually exclusive conditions 1 ■ and 2 ■ from our example CDT in Figure 8.4(b). The red and

blue predicates must be available in the regions with the operations in them, but the center gold region might not have any predicates available. This is indicated by the small color patches in the upper corner, similar to the way available predicates were indicated per phase in Section 8.2.2. The illustration is simplified by omitting the time dimension. The signals can tunnel through the gold region without interfering with one another like the previous illustration in Figure 8.1. Routing in different directions on wires requires no control switching, so the blue signal heading up and the red signal heading down also incur no congestion costs.

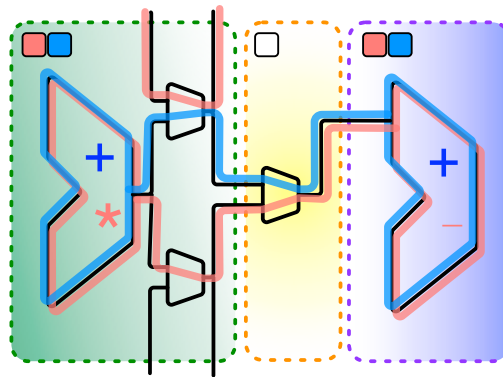


Figure 8.12: Single signal divergence and re-convergence with control congestion.

In this example, the signals clearly do not interfere with each other through false congestion costs. Generalizing, mutually exclusive signals that either follow the same path or diverge will not interfere with one another, even in regions without the predicates available for sharing configurations between them. What this example does not demonstrate is re-convergence after divergence. The reconvergence will require the use of two different inputs of a mux. In this case, there will be no signal congestion, however, there could still be control congestion in nearby regions. An example is illustrated in Figure 8.12. Here, two muxes have been added in the green region where signals routed under 1 ■ and 2 ■ may share.

The predicates needed for switching are available in the green and purple dotted regions, but not the gold region. Additionally, there is another signal routing under 1 ■

that appears at the top of the diagram. The top mux may be shared by both the signal routed under 1 ■ coming from above and the signal routed under 2 ■ coming from the operation in the green region because the predicates are available for making the configuration switch. However, the input mux in the gold region requires a configuration change to route from the different inputs. Without the predicates being available, these will have control congestion. Assuming the bottom mux has a base cost that is higher than the top mux plus the cost incurred from control congestion, this is a plausible set of initial routes.

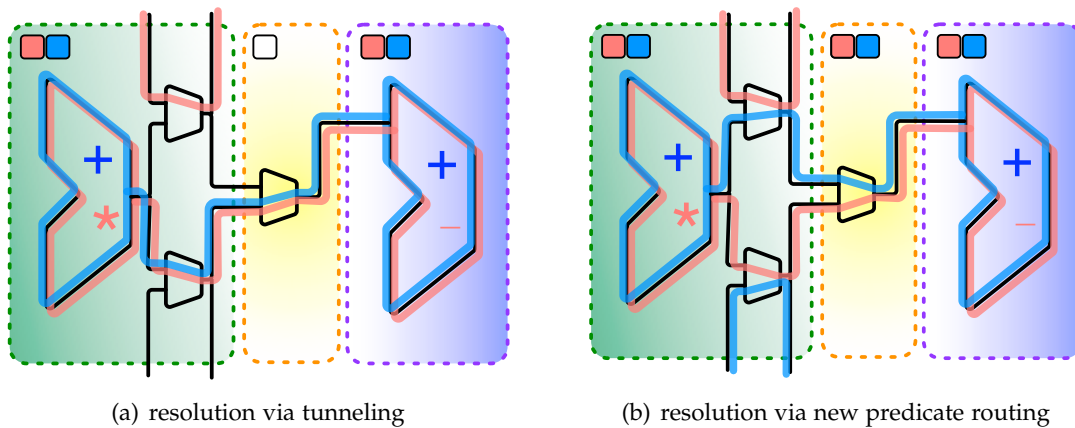


Figure 8.13: Example conflict resolution for routes.

If the routes under 1 ■ and 2 ■ from the operations shown on the left of Figure 8.12 are treated as independent signal trees, then the mechanisms that have already been introduced will serve to resolve the congestion. The control congestion will increase the history cost associated with the mux in the gold region, eventually causing the route under 2 ■ to take the lower path as shown in Figure 8.13(a). Alternatively, this may be prevented by another signal under 2 ■ routing through the lower mux, in which case the congestion could be resolved by routing the predicates for 1 ■ and 2 ■ to the gold region, illustrated in Figure 8.13(b).

Even though there is control congestion, and the signals are routing from the same physical resource, it is not false congestion. It is not valid for the router to simply start

the route under 2 ■ with the route under 1 ■ like in the multi-sink regular PathFinder/QuickRoute. Any portions routed using 1 ■ will only be guaranteed to exist during executions where the condition 1 ■ is true. Since the portion routed under 2 ■ is needed only when condition 1 ■ is false, if the route under 2 ■ were to assume the use of part of the route without reserving it, that portion of the route may not be properly configured at run-time. In this simple case of source and sink conditions matching, signal trees from co-located operations may be routed independently without concern for unresolvable false sharing (discussed below).

This extends to the case where there are multiple sinks for the routing trees of each shared source, where the sink conditions match the corresponding source condition. Each tree may be routed independently using the priority queue initialization of Algorithm 8.2 for each branch.



8.4.3 Predicate Aware Multi-condition/-source/-sink Routing

The source and sink conditions will be the same for any signals between operations within the same control flow block. However, the signals representing live values being transferred into and out of a control flow block generally have different conditions for their source and sink operations. In fact, with multi-sink signals, each sink could be executed under a different condition. How those conditions relate to each other dictates which portions of the routes may be re-used in the priority queue initialization across fanouts.

There are several possibilities for how different conditions relate to one another at run-time based on their relationship in the CDT. Here they are given in terms of conditions A and B , and the \Rightarrow symbol will be used to denote the relationship. Here are the possible relationships of $A \Rightarrow B$:

- Ancestor – When A is on the path from B to the root, A is an ancestor of B and will be true in a superset of the iterations where B is true.
- Descendant – When B is on the path from A to the root, A is a descendant of B and will be true in a subset of the iterations where B is true.

- Mutually Exclusive – The least common ancestor of A and B is a partition node, and so A and B will never be true in the same iteration.
- Unrelated – The least common ancestor of A and B is a condition node, and so there is no known correlation between the iterations where A is true and the iterations where B is true.

Note that this relation is not reflexive, because if $A \Rightarrow B$ is Ancestor, then $B \Rightarrow A$ is Descendant. Of those four relationships, only three are possible for the conditions of a given fanout. Let C_s denote the source condition and C_t denote the sink condition. C_s will be the ancestor of C_t when the sink operation is in a block nested within the block where the source operation occurs. C_s will be a descendant of C_t when this nesting is reversed. C_s and C_t will be unrelated when each operation is nested in a separate control construct, each of which occur within the same block, for example conditions 9  and 11  from the sequenced if-then-else constructs in Figure 8.4. C_s and C_t will not be mutually exclusive, relative to the latency of the communication. This means that if the communication between the source and sink is within the same iteration, C_s and C_t will not have a partition node as their least common ancestor in the original CDT. If the communication is across iterations, the router does not know about any relationship between C_s and C_t , as currently implemented (though this could change, as per Section 5.2.4). This is because the CDT is constructed for a loop body, representing a single iteration. If the source and sink operations were to execute under mutually exclusive conditions, whenever the source executes, the sink would not, and vice versa, meaning no communication between the two could take place. A corollary to the fact that a source is never mutually exclusive with the sink is that for a given source condition, there are no sinks from co-located (mutually exclusive) sources with the same condition. That is to say, for two source operations sharing a device, s_1 and s_2 , under conditions C_{s_1} and C_{s_2} , respectively, there is no sink of s_1 under condition C_{s_2} , and there is no sink of s_2 under condition C_{s_1} .

Now consider the possible relationships between the conditions of source operations of multiple fanouts emanating from same physical source. The fanouts may be from the

same operation, in which case the source conditions are the same. They may instead be from co-located operations that are sharing the physical resource, in which case the conditions will be mutually exclusive. These are the only possible cases, because operations may not share a physical resource when their conditions have any other relationship.

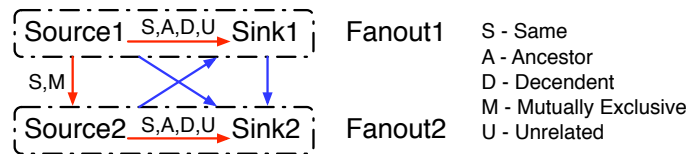


Figure 8.14: Possible condition relationships between fanouts with the same source device.

These source-source and source-sink condition relationships are illustrated in Figure 8.14. Two fanouts are shown with red arrows representing pairs of conditions whose relationships are important to routing. The limitations described in the previous two paragraphs are given as annotations on the appropriate arrows, where each letter represents a valid relationship for that pair out of the five possibilities on the right. These limitations are independent of each other. The limitations for the remaining relationships may be derived given the within-fanout source-sink and cross-fanout source-source relationships. If Fanout1 is routed first, these relationships will determine which portions of the route may be re-used by Fanout2. The limitations lead to 32 possible combinations of relationships between the four conditions.

A particular fanout may be routed using either its source or sink condition at any point in time, because the route only needs to be complete when both operations will execute. Once the route is completed, each location along the route will be reserved under either the source condition or the sink condition. These will be referred to as *s*-routed or *t*-routed, respectively, reflecting the common use of *s* and *t* to denote source and sink in routing algorithms. A fanout can re-use a portion of an existing route starting at the source as long as that portion was contiguously routed under a condition that can serve as a surrogate for the current fanout. Acceptable surrogate conditions are the same as or ancestors of either the source or sink condition of a fanout. This definition of surrogate

ensures that the proper configuration will be retrieved at run-time for the current fanout. It must be specified as a contiguous portion because the route may be switched back and forth between being *s*-routed and *t*-routed. If only the source condition is a valid surrogate for the current fanout, there are *s*-routed portions that cannot be used as a starting point for the current fanout because there are *t*-routed portions of the partial-route that will not be valid for the current fanout at run-time.

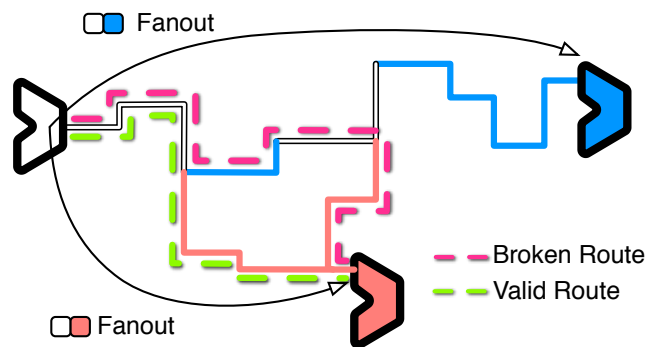


Figure 8.15: Illustration of broken and valid route re-use.

An example of a valid route and a broken route that can occur from re-using a prior route is illustrated in Figure 8.15. There are two fanouts, the first routed under conditions 0 □ and 2 ■, the second routed under 0 □ and 1 ■ from the example CDT in Figure 8.4(b). Condition 0 □ is an ancestor for condition 1 ■, so it is a valid surrogate for the entire second fanout (source to sink). However, condition 2 ■ is not. This means the initial contiguous *s*-routed portion of the 0 □ 2 ■ fanout can be used as a starting point for the 0 □ 1 ■ fanout, given by the marked valid route. The intervening *t*-routed section under condition 2 ■ is not guaranteed to be valid when condition 1 ■ is true, so the route marked as broken may not be complete at run-time. The process that generates alternating *s*- and *t*-routed partial paths will be covered in Section 8.4.5; for now it is just important to recognize that it is a possibility.

Consider the relationships between two fanouts from the same source. The 16 possibilities are listed in Table 8.1. In the table, conditions for sources 1 and 2 are denoted s_1 and s_2 , respectively, and conditions for sinks 1 and 2 are denoted t_1 and t_2 , respectively.

The letters S, A, D, M, and U stand for the possible relationships between conditions as given in Figure 8.14. The heading indicates which relationship in Figure 8.14 that column represents, and there is a separate row for each possible combination of the independent relationships of $s_1 \rightleftharpoons s_2$, $s_1 \rightleftharpoons t_1$ and $s_2 \rightleftharpoons t_2$. For example, the A entry under column $s_1 \rightleftharpoons t_1$ in the second row indicates that the condition for source 1 is an ancestor of the condition for sink 1. Remember that the CDT only tells us about the relationships between things that are in the same iteration. Therefore, the relative iteration delay between two uses of a condition needs to be determined to be able to correctly determine the relationship in cases like $t_1 \rightleftharpoons t_2$. If the relative iteration delay between t_1 and t_2 is zero, then they can be compared using the CDT, otherwise they receive the default relationship of U. These relative iteration delays must be determined in some common frame of reference, and since everything here shares a common source, that can serve as a default frame of reference. Therefore, determining the relative delays between t_1 and t_2 is simply a matter of finding the iteration delays from the common source to each sink operation and taking the difference.

	Given			Derived		
	$s_1 \rightleftharpoons s_2$	$s_1 \rightleftharpoons t_1$	$s_2 \rightleftharpoons t_2$	$s_1 \rightleftharpoons t_2$	$s_2 \rightleftharpoons t_1$	$t_1 \rightleftharpoons t_2$
1	S	S	S	S	S	S
2	S	A	S	S	A	D
3	S	D	S	S	D	A
4	S	U	S	S	U	U
5	S	S	A	A	S	A
6	S	A	A	A	A	A,S,D,M,U
7	S	D	A	A	D	A
8	S	U	A	A	U	U
9	S	S	D	D	S	D
10	S	A	D	D	A	D
11	S	D	D	D	D	A,S,D
12	S	U	D	D	U	D,U
13	S	S	U	U	S	U
14	S	A	U	U	A	U
15	S	D	U	U	D	A,U
16	S	U	U	U	U	A,D,M,U

Table 8.1: Table of possible condition relationships for fanouts from the same source.

For any pair of fanouts with the same source condition, the second routed fanout may always re-use the initial s -routed contiguous portion from the first. The condition relationships in Table 8.1 need to be examined to determine when t -routed portions may be re-used. The $s_2 \Rightarrow t_1$ and $t_1 \Rightarrow t_2$ are the columns relevant to applying t -routed portions to the second fanout because they represent the relationship between when the t -routed section is valid and when the second fanout will need them.

A value of S or D in the $s_2 \Rightarrow t_1$ column indicates that the t -routed portion was routed under a condition that is either the same as or an ancestor of the second source condition. This means that any t -routed portions were routed under a condition that serves as a surrogate of the second source. Since both the s - and t -routed portions are fit for re-use, the entire route may be added to the initial priority queue for the second fanout.

A value of S or A in the $t_1 \Rightarrow t_2$ column indicates the t -routed portion was routed under the same/ancestor condition of the second sink condition. Thus, it was routed under a surrogate of the second sink. The entire route may be re-used in these cases as well.

Observe that an D in the $t_1 \Rightarrow t_2$ column indicates that the t_1 condition is the decedent of the t_2 condition, so swapping the routing order of the fanouts turns this relationship into an A. If the fanouts are sorted by the depth of their conditions for routing, then the router is guaranteed to never see any D relationships between t_1 and t_2 . All fanouts of a particular condition will have been routed before any fanouts with a descendant condition. This sorting can be accomplished using either a breadth-first or pre-order depth-first ordering. Using a breadth-first ordering will also guarantee that all fanouts with a sink condition that is an ancestor of the source condition are routed before any others. This will ensure the most re-use due to D entries in the $s_2 \Rightarrow t_1$ column, and this is what is implemented in PA-SPR.

Using these relationships, Table 8.1 can be reorganized. Assuming sink-condition-depth sorted routing, the entries with a D in the $t_1 \Rightarrow t_2$ column may be removed entirely. The remaining entries are separated into two tables. The first is given in Table 8.2, and represents all of the situations where the entire route of the first fanout can be used to initialize the priority queue for the second fanout. The entries from Table 8.1 that were

	Given			Derived		
	$s_1 \Rightarrow s_2$	$s_1 \Rightarrow t_1$	$s_2 \Rightarrow t_2$	$s_1 \Rightarrow t_2$	$s_2 \Rightarrow t_1$	$t_1 \Rightarrow t_2$
1	S	S	S	S	S	S
2	S	S	A	A	S	A
3	S	S	U	U	S	U
4	S	D	S	S	D	A
5	S	D	A	A	D	A
6	S	D	D	D	D	A,S
7	S	D	U	U	D	A,U
8	S	A	A	A	A	A,S
9	S	U	U	U	U	A

Table 8.2: Table of condition relationships that support sink- and source-routed re-use.

	Given			Derived		
	$s_1 \Rightarrow s_2$	$s_1 \Rightarrow t_1$	$s_2 \Rightarrow t_2$	$s_1 \Rightarrow t_2$	$s_2 \Rightarrow t_1$	$t_1 \Rightarrow t_2$
1	S	U	S	S	U	U
2	S	U	D	D	U	U
3	S	U	A	A	U	U
4	S	A	U	U	A	U
5	S	U	U	U	U	M,U
6	S	A	A	A	A	M,U

Table 8.3: Table of condition relationships that support only source-routed re-use.

not either placed in Table 8.2 or removed entirely enumerate the situations where only the initial contiguous s -routed portion of the first fanout may be used to initialize the priority queue for the second fanout. These entries are gathered in Table 8.3.

The entries in Table 8.2 that indicate the full route may be re-used are emphasized in bold. Note that in line 3 and 7, there is a plain U entry in the $t_1 \Rightarrow t_2$ column. For these situations, the t -routed portion is only re-usable because it was routed under an ancestor of or the same condition as s_2 . The opportunity for sharing under these cases can be maximized by routing any sinks under the common source condition or any ancestor before any other conditions at the same depth.

Having covered fanouts that share the same source condition, it is now time to consider the other 16 cases, where the source conditions are mutually exclusive between

fanouts. When the sources are mutually exclusive, the s -routed portion of the route cannot be re-used. Note that re-using a portion of a route through priority queue initialization requires that portion of the route to start at the source. Since the fanouts are from two mutually exclusive operations on the same resource, at least the first wire will need to be shared because the routes need some space to diverge; thus, they will likely be routed under the source conditions. This means that there is no opportunity to share between fanouts with mutually exclusive sources if they are routed independently.

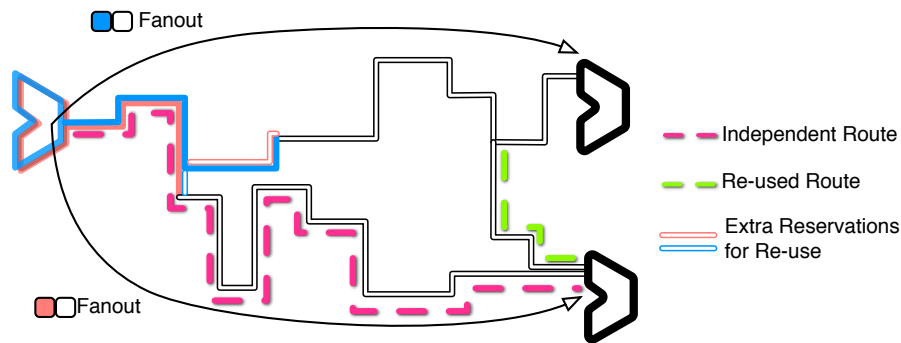


Figure 8.16: Illustration of re-used versus independently routed mutually exclusive source fanouts.

The s -routed portions of the fanouts may be routed jointly under a union of the mutually exclusive predicates. This simply reserves the routing resources for the source conditions of all fanouts that are routed together for the s -routed portions. An example of the difference is illustrated in Figure 8.16. Any portion that is not re-used is effectively over-reserved, preventing other signals from routing through later. If the fanouts in Figure 8.16 are routed independently, with the 2 0 0 fanout routed first, the 1 0 0 fanout will need to reserve an entirely separate set of resources under 0 0, marked by the independent route path in the figure. However, if the s -routed portion is routed jointly under the combination of 1 2, then the entire path will be provided in the priority queue initialization for the second fanout. The second fanout can then branch off of that route much later, requiring fewer resources under condition 0 0.

This method of jointly reserving resources for a route under the union of source conditions effectively turns the mutually exclusive source conditions into the same merged

source condition for routing purposes. This means the same reasoning that applied to the same-source-condition routing applies here, including Table 8.2 and Table 8.3 for re-using routes across fanouts. It is possible to release the reservations for the over-reserved portions that are not actually re-used by the time all the fanouts for a physical device are completed. If this is done before moving on to the next set of fanouts, the over-reservation will not interfere with subsequent routes. However, it will have affected the costs that those over-reserved portions have seen. In the version implemented in PA-SPR for evaluation s -routed sections are jointly reserved, but over-reserved sections are not released.

8.4.4 Predicate Aware Route Re-use

In light of these relationships between the conditions of multiple sinks, the priority queue initialization used to re-use routes across fanouts on Lines 10-12 in Algorithm 8.2 can now be re-formulated to allow predicate-aware route re-use across fanouts. Initially, the router in PA-SPR takes all of the fanouts for mutually-exclusive operations mapped to a single source and groups them by their sink condition. Each of these groups also receives a separate priority queue initialization set. This set will be used to initialize the priority queue as each fanout from the associated group is routed, and any routes that should be re-used will be added to the appropriate sets as they are created.

Next, the router collects the union of source operation conditions, which are jointly set as the source condition for all of the fanouts. The router then processes the groups in breadth-first order of the sink conditions as they appear in the CDT. The initial router implementation in PA-SPR does not include the optimization to route the ancestors of the source conditions before other conditions at the same depth. This may reduce the amount of route re-use, because all fanouts can re-use routes under ancestors of the source condition, but those routed early will not get the chance at this re-use.

The router will go through the fanouts within a group in arbitrary order, using the predicate aware version of QuickRoute to obtain a route for each fanout. After a route for a fanout is completed, the resources are reserved under the appropriate conditions – the set of source conditions for s -routed portions and the sink condition for t -routed portions.

Finally, portions of the route are added to the appropriate priority queue initialization sets. The first contiguous s -routed portion is added to all of the initialization sets. The entire route is added to any initialization set for any sink conditions where the current sink condition can serve as a surrogate, as defined by Table 8.2. This process is shown in Algorithm 7. This algorithm can be sped up by trading execution time for memory space by pre-evaluating and storing the results of Lines 8-12 before the outermost loop begins; however, it is implemented in PA-SPR as shown for simplicity.

Algorithm 8.3: Predicate Aware PathFinder/QuickRoute integration

```

1 begin
2   while iterate do
3     for co-sourced sets of signals  $i$  do
4       if  $\text{congested}(RT_i) \vee \neg \text{routed}(RT_i)$  then
5         Rip-up routing tree  $RT_i$ 
6          $RT_i \leftarrow$  sources of signals  $s_i$ 
7          $c_u \leftarrow \cup$  source conditions of  $s_i$ 
8         forall fanouts  $st_{ij} \in i$  do
9           add  $st_{ij}$  to  $g_{\text{cond}(t)}$ 
10          set  $s$ -routed condition to  $c_u$ 
11         foreach fanout group  $g_c$  do
12           init  $PQ_{g_c}$  with  $s_i$ 
13         foreach group  $g_c$  in BFS CDT order of  $c$  do
14           while  $\exists$  unrouted fanout  $st_{ij} \in g_c$  do
15             Initialize priority queue  $PQ$  to  $PQ_{g_c}$  at cost 0
16             Run QuickRoute search for  $st_{ij}$  using  $PQ$ 
17             forall nodes  $n$  in path  $st_{ij} \notin RT_i$  do
18               Update PathFinder  $\text{cost}_n$ 
19               Add  $n$  to  $RT_i$ 
20               foreach Unfinished group  $g_f$  do
21                 if  $n$  is  $s$ -routed or  $\text{cond}(t)$  is surrogate for  $f$  then
22                   Add  $n$  to  $PQ_{g_f}$ 
23           if on every  $n$ th iteration then
24             setup new predicate routes
25            $\text{iterate} \leftarrow \text{isCongestion} \wedge (\text{iteration} < \text{limit}) \wedge \text{progress}$ 
26 end

```

This section detailed the handling of routes that change the condition they are routed under as they are routed. An algorithm for combining multiple source-sink routes into a routing tree was crafted through an understanding of the relationships between those conditions when they change along the route. Examining the relationships led to two features of the algorithm that increase the opportunity to share resources within the routing tree – routing in breadth-first CDT order of the sink conditions and routing all fanouts with a source condition that is the union of source operation conditions. With the ability to re-use partial routes in place, the final incomplete portion of the predicate-aware router will be explained in the next section.

8.4.5 Choosing the Routing Condition

In Figure 8.10(b), the first modification to QuickRoute is to choose between either the source or sink condition for each step of the route. This section will cover how that choice is made. Near the beginning of the route, the most natural choice is the source condition. The predicates needed to share routing resources are guaranteed to be available as long as the route is in the source region and phase because they are being used by the source operations for sharing. However, as soon as the route leaves that region or passes through a register, the available predicates may change. Similarly, once the route gets to the sink region at the appropriate latency from the source, the predicates for the sink operations will be available.

There are many possible strategies to choosing which condition to route under. For example:

- As Soon As Possible – The route can start under the source condition and switch to the sink condition as soon as the predicate for it is available.
- As Late As Possible – The router can try to route under the source condition for as long as possible before it must switch to the sink condition to avoid conflicting with other routes sharing the same sink via mutually-exclusive operations.

- Spatial Midpoint – The router may try to balance the distance routed under the source and sink conditions by changing to the sink condition when the A* distance estimate to the sink becomes smaller than the currently routed distance when an architecture-specific distance estimate is provided.
- Temporal Midpoint – The router may try to make a similar temporal balance by waiting to changing to the sink condition until after half of the required latency for a route has been passed.
- Locally Cheapest – The router retrieves the estimated cost of routing through a resource under each of the conditions and chooses the cheapest at each stage of the route.

It is not clear which of these options will achieve the closest to globally optimal routing results. In this initial investigation, the one chosen for implementation in PA-SPR is the as-soon-as-possible option. Taking coding considerations into account and the structure of the existing SPR code-base, this was one of the simplest options to implement. Additionally, it will skew the routes towards more *t*-routing, which reduces the penalty from over-reserving resources in the *s*-routed portions. This also affords the router the most opportunity to find a location for the route to begin sharing with any other signals that will end up at the same sink for mutually exclusive operations. Note that the router does not switch to routing under the sink condition until after the predicate for it has been generated. It is possible that the source operation was scheduled before the operation that generates the predicate, so there will be a portion of the time where the only predicate available to support resource sharing is the source predicate. Investigating other methods of choosing the condition to route under is left for future work.

One final note is that the choice of conditions must be latency aware. As the route accumulates latency, the source condition must be adjusted to represent the appropriate scheduled time relative to the source operation scheduled time. Once the router is executing, it must use the appropriate conditions from the CDT that has been expanded with time information, as presented in Section 5.2.4. When the route is switched to using the

sink condition, it must do the reverse calculation, using the latency left to go on the route and subtracting it from the sink schedule time.

Chapter 9

EVALUATION OF PREDICATE AWARE MAPPING

This chapter will cover a brief empirical evaluation of the new approaches presented in Chapters 5-8. The evaluation will be done using PA-SPR, an extension of the original back-end for the Mosaic tool-chain, SPR. The goal of the evaluation is to show that is possible to improve the performance of kernels through sharing, and attempt to identify how much of this potential PA-SPR can realize.

The evaluation uses a suite of 7 benchmarks that are algorithms with loop-level parallelism typically seen in embedded signal processing and scientific computing applications. The benchmarks are written in the Macah language, and the kernels were compiled to an intermediate form suitable for PA-SPR using the enhanced loop-flattening techniques described in [Ylv10]. Those kernels were then mapped to simulation CGRA architectures. I modified the architecture simulation to support predicate based configuration switching suitable for the abstractions in Chapter 5. The architectures are defined as structural Verilog, generated by the Mosaic Architecture Generator plug-in to the Electric [SS] VLSI Design System. The mapped kernels were validated in Verilog simulation to verify the correctness of the mappings.

The next section will describe the architecture model. Section 9.2 will cover the set of benchmarks used in the experiments. Section 9.3 will use the scheduling results across the benchmarks to quantify the potential benefit of predicate-aware mapping. Finally, Section 9.4 will present the results of running a full predicate-aware mapping for all of the benchmarks across a range of architectural sizes.

9.1 Evaluation Architecture

The architecture used for the evaluation of PA-SPR is generated using the Mosaic ArchGen plug-in for the Electric VLSI system. The initial Mosaic architecture explorations done

in [VE10] developed a clustered CGRA that were suitable for targeting with SPR. This evaluation uses a simplified version of those clustered architectures, augmented with support for predicate-based resource sharing. The simplifications were made either to overcome technical limitations of the current implementation of PA-SPR or to eliminate complications that may interfere with PA-SPR's ability to share resources. The reasoning behind individual simplifications will be provided as they are described below.

The general clustered architecture is maintained with clusters of compute elements connected together with a grid style interconnect. The switch-boxes of the interconnect are the same Wilton style switch-boxes described in [VE10]. The structure within the cluster has been simplified relative to [VE10]. The interconnect and clusters are divided into word-wide and bit-wide elements. Each cluster contains a set of compute elements plus some delay elements all connected directly to either the word-wide or bit-wide crossbar.

The word-wide elements in the cluster are the ALU, stream-in, stream-out, constant generator, live-out, and a shiftable memory with a read, a write, and a shift port. The main compute element is an ALU, which can handle all word-wide computation operations in a single cycle for simplicity. The constant generator produces constants and live-in values to the kernel. Macah streams are mapped to the stream-in and stream-out ports to handle the main communication of data with the kernel. The values of live variables are transferred back to the main computation thread through the live-out devices upon kernel completion. In addition to these compute elements, retiming-chains are included as delay elements connected to the crossbar. They provide the storage and pipelining of live values during kernel execution.

The bit-wide elements are the look-up table (3-LUTs), bit-wide constant generator, bit-wide live-out, kernel-completion indicator, and predicate gateway. The LUTs carry out the bit-wise calculations required by the kernel. The run-time signal that indicates a kernel has finished running is routed to kernel-completion indicator so that control can be properly passed back to the main sequential thread at the appropriate time. Finally, the predicate gateway is the new device used by PA-SPR that acts as the sink for any predicate signals being routed to the configuration hardware of the region. The constant generator, live-outs, and retiming-chains operate like their word-wide counterparts.

The simulation infrastructure for these architectures was modified to support predicate-based configuration retrieval. The ports of the predicate gateway can be configurably connected to the most significant bits of the configuration array address lines, using the model presented in Figure 5.19(b). This configurable connection can change on a per-phase basis. The configuration for connecting either the modulo-counter lines or predicate lines to the configuration memory cannot itself be modified by predicates. The region that a predicate gateway connects to covers a whole cluster and the associated switch-box. The architectures are configured to support an II of up to 256, with a 4 predicate-port gateway. Since the predicates are inherently 1-hot encoded, and no re-encoding is supported currently in PA-SPR, that means a maximum 4-way sharing of each device up to an II of 16, then 3-way sharing at an II of 32, and so on.

Aside from the support for predicate-based configuration retrieval, the main differences between the architecture described in [VE10] and the one used here are the lack of rotating register files and the simplification of the processing elements (PEs). The reasoning behind these changes follows in the next two sections.

9.1.1 Register File Limitations

The use of structures with a strict demultiplexor is not currently supported in SPR or PA-SPR. Prior work has shown support could be added [HD08], but it was not done for this evaluation. In particular, this limits the use of SRAM based register files, as their write ports are usually strict demultiplexors, i.e. the register file may only store a new value to a single register at a time. In the original SPR, this limitation is worked-around when the final configuration is generated. All signals written to multiple registers in a register file can be collapsed to the register that will go the longest before being overwritten. Any reads of that signal can all be serviced by that register. After this post-processing, all the stores to register files only need to write to a single register and standard register files or rotating register files may be used.

In PA-SPR, the added complication of mutually-exclusive signals sharing paths through a register file, this post-processing is no longer a viable way to legalize a mapping. An example set of routes that cannot be turned into an equivalent set of single register writes

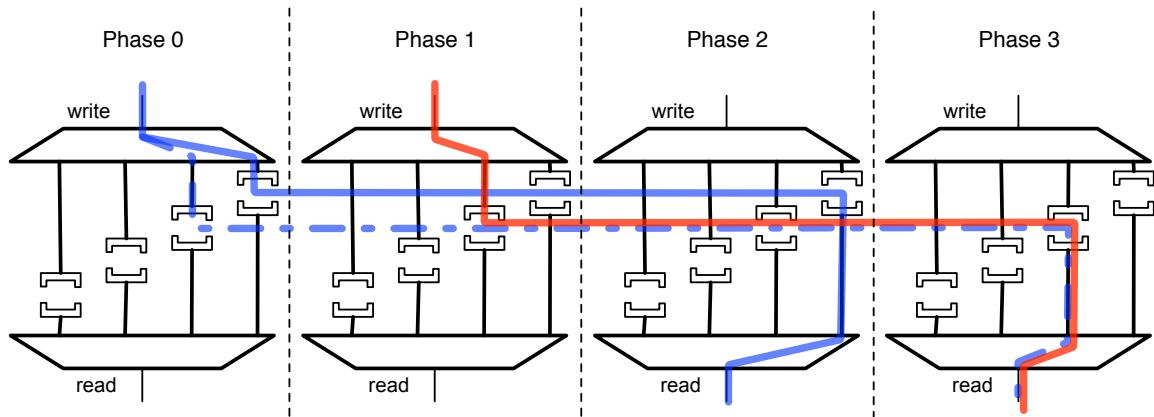


Figure 9.1: Diagram illustrating conditional routing through a register file that cannot be fixed through post processing.

is illustrated in Figure 9.1. The blue signal enters in the first phase and is broadcast to two registers. The solid blue and the dotted blue represent the signal traveling to two destinations under different conditions. Dotted blue is mutually exclusive with solid red, which allows the signals to share resources. However, solid blue and solid red are not compatible. Once these routes are chosen, there is no way to resolve the broadcast of the blue signal into a single register write. If the solid blue signal is transferred to the dotted signal's original register, it will be overwritten by the red signal at times. If the dotted signal is transferred to the solid signal's original register, the read port will need to be changed at execution time based on whether the dotted blue or solid red condition is active. If the predicate needed to do this is unavailable in the current predicate control region, the port will not be able to switch properly. This will render the post-processing invalid, and the program will not execute as intended.

To work around this limitation, the rotating register files and other signal delay resources are replaced with retiming chains. A simple retiming chain is illustrated in Figure 9.2. It is a cascaded set of delay registers with a mux that acts as an output to read the value from any point along the chain. The retiming chains used in this evaluation have two output ports and are 24 registers deep. The retiming chains are able to delay a value

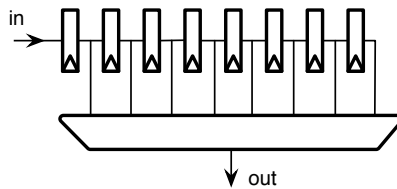


Figure 9.2: Diagram illustrating a simple retiming chain.

up to the maximum depth of the chain no matter what the II of the executing application is.

9.1.2 Simplified Processing Elements

The processing elements (PEs) in the Mosaic architecture can roughly be defined as the computation units plus any support structure that does not include the cluster crossbar. Sending signals across a large crossbar can be costly in terms of power, and as a result the architectural exploration in [VE10] added a large amount of local interconnect resources to the PEs to keep values locally and avoid sending them through the crossbar. An example of one of these complex PE's is shown in Figure 9.3(a). Additionally, the compute units were heterogeneous, with some able to perform the full complement of word-width arithmetic and logical computations available in Macah, and others where the multiplication or the multiplication and shifting had been removed.

In order to maximize the opportunity for sharing, the architectures used in this evaluation forgo compute unit heterogeneity and resort to the original model of an ALU that can process all arithmetic and logical computations. This will allow an addition and a multiplication to share anywhere there is an ALU, instead of only at the locations of the Universal Functional Units as in [VE10].

Initial development of PA-SPR used the complex PE interconnect in testing and debugging. The rotating register files were replaced with deep retiming chains due to the problem from the previous section, and the clock-gated registers on the in puts were replaced with short retiming chains. Often, mappings in PA-SPR would fail due to congestion at the bottlenecks where the PE interconnect is attached to the crossbar. The

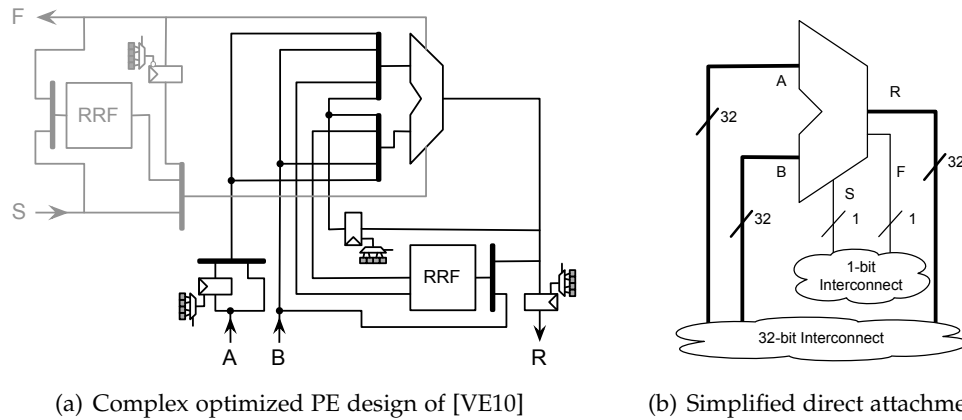


Figure 9.3: Complex and simplified Mosaic architecture processing element designs.

complex PE interconnect was removed to avoid these bottlenecks, connecting the ports of the ALU directly to the bit-wide or word-wide crossbar, as shown in Figure 9.3(b). The number of long retiming chains attached to the crossbar was also increased to compensate for the reduced resources for holding live signals. These changes immediately resolved the failures due to long-standing congestion.

Directly connecting the computation units to the crossbar appeared to increase the success rate of PA-SPR, and so this is the structure that was used for the evaluations presented here. It would be interesting to do a detailed investigation of how bottlenecks such as those introduced with the complex PE structure affect routing success in PA-SPR and SPR, but that is left for future work.

9.1.3 Intra-connect Capacity

Directly connecting all of the compute units to the crossbar separately from all of the storage elements also has another benefit: it provides a simple way to control the capacity of the within-cluster intra-connect. The capacity of the intra-connect, measured either by the number of live values that can be stored or the number of alternate routing paths between units, can be scaled up or down simply by adjusting the number of retiming chains attached to the crossbar. While this may not be the most area- or energy-optimal

architecture design, it does provide a useful control for increasing or decreasing the difficulty of the local routing problem that the router is trying to solve, which is important to evaluating the new mapping methods in PA-SPR.

9.1.4 Test Architecture Configurations

The experiments in this chapter use a sweep of architecture sizes. There are 4 important parameters that are changed in this sweep. A specific architecture is described in the form $C:aXsYdZ$. The number of clusters is given by the leading value C . The cluster composition follows, with X representing the number of computation units, Y representing the number of stateful units, and Z representing the number of storage or delay units attached to the crossbar. For a given value of X , there are X ALUs and word-wide constant generators. Half of that number is used for the bit-wide computation elements, yielding $\lceil X/2 \rceil$ LUTs and bit-wide constant generators in the cluster. For a stateful value of Y , there are $Y * 3$ memories, each with a single read, single write, and single shift port. There are $Y * 2$ stream-in/out devices. The final value, Z , indicates the number of retiming chains included in the cluster – Z word-wide and Z bit-wide per cluster.

The primary values of interest are C and X . The number of clusters and compute units are varied to test PA-SPR across a range of resource constraints. Predicate-aware sharing is meant to improve performance in resource-constrained situations and allow for more complex control flow while maintaining high performance. This evaluation is aimed at measuring that performance improvement. The other values are chosen to avoid interfering with measuring PA-SPR's effectiveness at addressing this main goal.

The amount of intra-cluster and inter-cluster communication and storage is set at a large value relative to the computation resources to provide a mapping success rate that allows comparison between PA-SPR and SPR. The grid inter-connect is set at a width of 24 fully time-multiplexed channels, both in bit- and word-wide sizes. The channel length is set at 2, which means that a signal can be routed through up to 2 switch-boxes before being registered. The number of retiming chains in the clusters is set at a ratio of 5 chains per 2 ALUs.

For each cluster size, enough stateful devices are provided to support all of the benchmarks. Only one stateful element can be mapped to each stateful device in the architecture, such as a single Macah stream per stream-in or a single Macah array per local memory. In the future, the hardware model and SPR algorithms may be extended to support stateful packing, such as combining multiple small Macah arrays into a single local memory, but for this evaluation insufficient stateful elements results in immediate mapping failure. As a result, as the number of clusters shrinks, the number of stateful elements is increased to keep an approximately constant number available in the architecture.

9.2 Evaluation Benchmarks

The benchmarks and Macah have both undergone additional development from what was used in Chapter 4. Enhanced loop flattening was developed in Macah [Ylv10] to take arbitrary control flow within the kernel and flatten it to a set of guarded blocks within a single loop. For the experiments in Chapter 4, this was done in a manual process in the kernel source code itself. The manual flattening process hides a lot of the relationships between the blocks that were inherent in the original control flow. The enhanced loop flattening process in Macah tracks these relationships and represents them in the CDT. As a result, the CDTs for the manually flattened kernels may not offer as much sharing potential as those that were written with more natural control flow. Not all of the benchmarks were updated to use natural control flow, but where it is possible both the manually flattened and the Macah flattened versions are tested.

The evaluation of PA-SPR uses a suite of 7 benchmarks. These benchmarks were developed by members of the Mosaic research group in the Macah language. Of the 7, 6 are the same as those used in Chapter 4 for evaluating static interconnect sharing – motion estimation, matrix multiply, Smith-Waterman sequence alignment, FIR filter, k-means clustering, and 2D convolution. These 6 are the benchmarks that were updated with more natural control flow that Macah transforms into a single loop using enhanced loop flattening. However, the Macah compiler generates an invalid CDT for Matched

Filter for an unresolved reason. It is excluded from the tests here since the CDT is a central component that affects how resources may be shared in PA-SPR. A PET scanner event detection benchmark has been added to the suite, bringing the total used here to 7 different applications. The PET benchmark was developed after the experiments in Chapter 4 were completed.

Each of these benchmarks has a set of tuning knobs programmed into it that can adjust the sizing of various aspects of the algorithm. These allow tuning the application's implementations to the available architectural resources. The values used in this evaluation are given in Table 9.1, along with a few other interesting statistics. In addition to the tuning knob settings (found in the last column), there are columns to quantify the size and maximum theoretical performance of each benchmark. The Nodes column indicates the number of data flow operation nodes that are in the kernel. The Nets column indicates the number of source-to-sink data connections that occur between the data flow operations. The Dependencies column is the sum of source-to-sink data connections and all possible predicate connections that could be required for sharing. The Recurrence II is the minimum recurrence II possible if all operations are executed speculatively.

The old manually flattened versions have little opportunity to benefit from predicate aware sharing. Most of them have all operations execute unconditionally in the loop. Those with a tiny bit of sharing were CORDIC with the potential for sharing 16 out of 222 operations, K-Means with a potential 8 out of 398 operations, and Smith-Waterman with a potential of sharing 4 out of 381 operations. Initial tests on a highly constrained architecture showed only K-means had potential improvement after running the minimum II computation, so it was included in the benchmark set. The manually flattened version of K-means is marked with an (m). Using the tuning knobs, Matrix Multiply and Smith-Waterman were used in two different sizes, small (s) and large (l).

The mappings generated by PA-SPR were validated by running them through full simulation. Each benchmark was simulated on one of the 2- or 4-cluster architectures after mapping to test in situations with high sharing. Simulation was successful for all benchmarks except for the large Smith-Waterman, the large Matrix Multiply, and the 2-D convolution. In those applications, the Verilog simulator would process indefinitely

for an unknown reason. The simulation infrastructure is setup to identify erroneous computation or communication and fail with an appropriate error message, which was utilized extensively along with checks internal to PA-SPR during the development process to identify erroneous mappings. Given that no such error message was produced, it is possible that the failure is due to some other problem in the toolchain or simulator. The results are included here for completeness, with the caveat that the mappings were not fully verified for functional correctness.

Table 9.1: Summary of Benchmarks

Kernel	Nodes	Nets	Deps.	RecII	Knob Settings
Motion Estimation	664	979	1519	4	Block=8x8,Parallelism=4
Smith-Waterman (s)	367	634	878	5	Stripe Width=4
Smith-Waterman (l)	643	1138	1566	5	Stripe Width=8
2-D Convolution	493	772	1164	5	Kernel Radius=2, Stripe Width=4, Parallelism=2
Matrix Multiply (s)	258	425	652	3	Block=4x4
Matrix Multiply (l)	778	1337	24076	3	Block=8x8
K-Means Clust.	249	422	610	3	K=8, Channels=8
K-Means Clust. (m)	347	592	608	7	K=8, Channels=8
Blocked FIR	194	291	474	3	Coeffs=64, Banks=8
PET Event Detection	613	948	1300	4	Datasets=4

9.3 Scheduling Speculation versus Sharing

The portion of the evaluation presented in this section focuses on scheduling. The experimental results presented here serve two main purposes. First, they serve as a scouting measurement, identifying the situations where predicate-aware mapping has the potential to improve final performance. The scheduling algorithms run very quickly, and so covering a broad sweep of applications and architectures by only running the scheduling stage is much easier and faster than running a full schedule, place and route cycle. The initial scheduling sets an upper bound on the throughput (lower bound on the II), so running the scheduling provides an indicator of the best performance that a fully mapped design could achieve.

Second, the results here will test the $\text{resII}/\text{recII}$ balancing algorithm described in Section 6.2.3. When predicate signals are used to share resources and thereby reduce the resII , they may introduce a large recurrence loop, increasing the recII . The scheduler can identify the predicate signal dependencies that are on the large recurrence loops and eliminate (trim) them by promoting the dependent operations to run unconditionally, trading-off resource usage for shorter recurrence loops. This balancing algorithm is easily disabled by disallowing any dependence trimming, thus allowing the maximum amount of sharing possible.

The results in this section are collected from running the scheduler in three configurations so the resulting IIs can be compared:

- SPR - In this configuration, the standard SPR scheduling provides the baseline II measurements.
- PA-SPR no-trim - In this configuration, predicate aware sharing is enabled, but no dependence trimming is done in the face of inflated recII values.
- PA-SPR - In this configuration, the scheduling phase of PA-SPR is run using the algorithm described in Chapter 6 to enable predicate aware sharing and dependence trimming to maintain the $\text{resII}/\text{recII}$ balance.

The application configurations in Table 9.1 were run across a large sweep of architecture sizes, ranging from 2 to 42 clusters, using the three scheduler configurations. Predicate-aware sharing is most useful in resource-constrained situations, so the architecture sweep is intended to cover a range of architectures that tends towards smaller sizes.

9.3.1 *The Potential of PA-SPR*

The results in Figure 9.4 provide a theoretical bound on the performance improvement that can be achieved using predicate-aware mapping. This figure compares the minimum II values calculated using both standard SPR scheduling and the full PA-SPR scheduling.

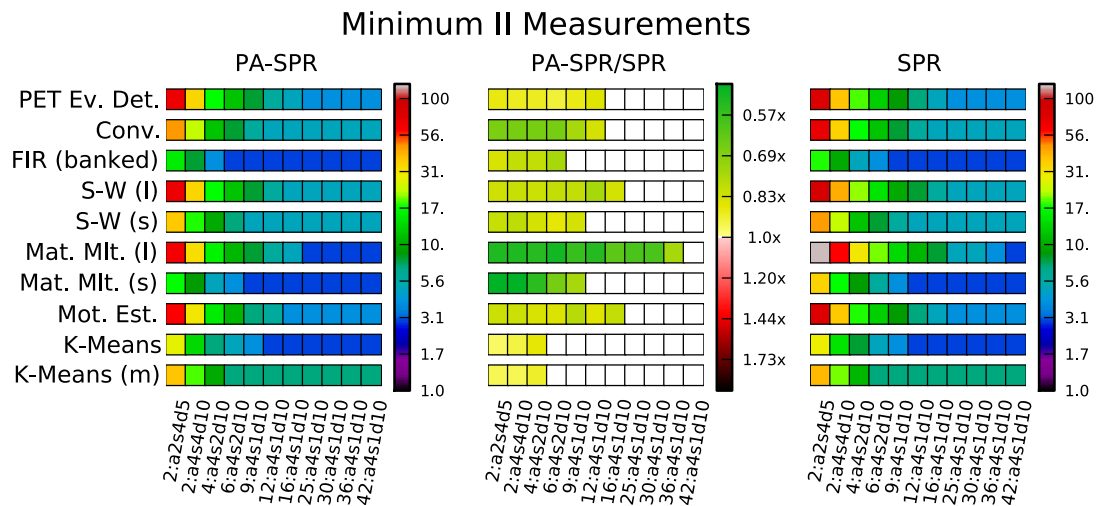


Figure 9.4: Plot of the minimum II estimate across the tests for both the full PA-SPR with dependence trimming enabled and the standard SPR, with the potential advantage of PA-SPR highlighted in the middle plot.

Applications are indicated along the Y-axis, and architecture sizes along the X-axis. In this figure, there are three sub-graphs. The left and right sub-graphs represent the minimum II measurements for PA-SPR with trimming enabled and SPR, respectively. The value of the minimum II is represented by the color of the box at a given application/architecture coordinate. The scale is provided in the colorbar on the right side of each sub-graph; note that for the left and right graphs this scale is logarithmic. The middle sub-graph presents the comparison of these two values, demonstrating the performance difference. The color of each box in the middle sub-graph is generated by taking the minimum II from the corresponding coordinate in the PA-SPR graph and dividing it by the minimum II from the SPR graph, and using that to choose the color from the middle scale. For example, the large matrix multiply kernel run on the 2:a2s4d5 architecture yields a minimum II of 66 and 122 from PA-SPR and SPR, respectively. This represents a .54x reduction in minimum II, or a potential throughput improvement of almost 2x. The II reduction of .54x appears at the top of the middle sub-graph scale, so the resulting box is colored green. The color

scale is constructed so that any reductions in II will be in the yellow to green spectrum, any increases in II will be in the red to black spectrum, and a match in II is colored white.

The results in Figure 9.4 show similar trends between PA-SPR and SPR in the left and right sub-graphs. These trends reflect the expected space/time trade-off across different sizes of applications and architectures. The values of minimum II increase towards the left of the graphs as the architectural resources are reduced and the II must increase to compensate. The larger kernels have higher minimum IIs relative to the smaller kernels. Even with their similarities in trends, the corresponding values can be very different between the two scheduler configurations, as highlighted by the middle sub-graph. There is potential improvement for all of the kernels on architectures with 4 or fewer clusters, and the large matrix multiply shows potential improvement all the way up to 36 clusters. The white beyond these points reflect that the minimum II calculated by both algorithms will be the same once the application is no longer resource constrained. The variability in the inflection points makes it difficult to summarize these results in a single II versus architecture size graph.

Smith-Waterman and matrix multiply were both run in small and large versions because the amount of computation they do scales well with tuning knob settings. Here we see that scaling up the amount of compute done changes the point in architecture sizes where the PA-SPR begins to have potential benefit. However, looking between the two benchmarks, it appears that matrix multiply has inherently more potential for improvement than Smith-Waterman, as evidenced by the darker green of the matrix multiply rows. The difference in minimum II between Smith-Waterman and matrix multiply is much more pronounced than the difference within applications between the two sizes. The variety of light and dark green maximum values indicates that the CDTs across applications vary in the maximum amount of sharing they enable.

It will be interesting to see how well the final improvement follows this variability across applications, so the results will continue to be presented broken out across application. The variability in the architecture size inflection points (the white-green transitions within an application row) makes it difficult to summarize these results in a single II versus architecture graph, so the full grid will be presented in the remaining results.

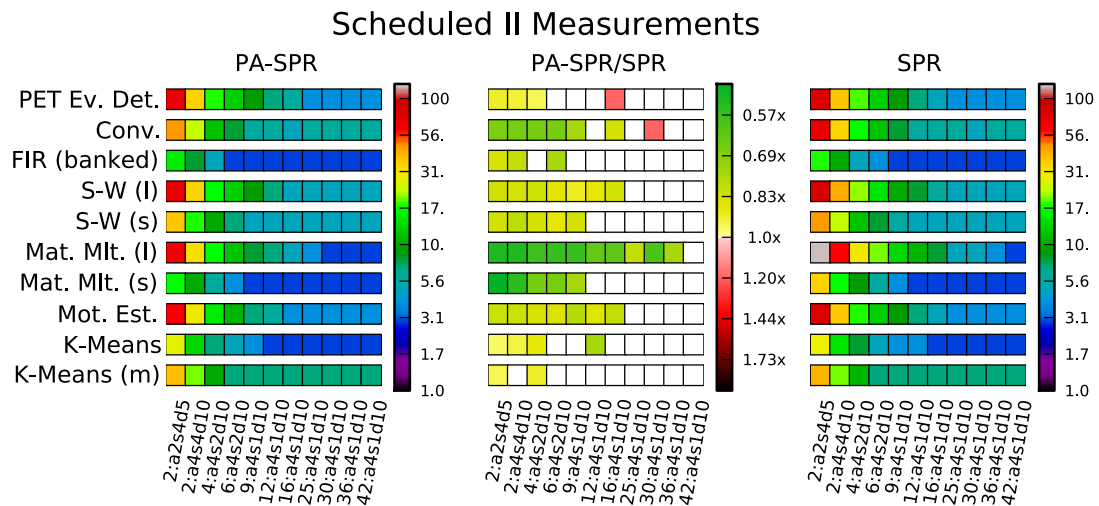


Figure 9.5: Plot of the II advantage of PA-SPR after the scheduling has been completed.

The next set of results presents a similar II comparison after running the full scheduling stage for both PA-SPR and SPR. These appear in Figure 9.5. The full scheduling starts at the minimum II estimate, which ignores all dependency information, and creates a full modulo-schedule that respects all dependencies in the data-flow graph. It is a heuristic scheduling process, so there is a bit of noise which can be seen in Figure 9.5, but overall the scheduler is able to realize most of the potential seen in Figure 9.4.

The actual losses are presented for each of the two algorithms in Figure 9.6. Given that the minimum II estimate in the PA-SPR algorithm is optimistic in two ways – ignoring dependencies and assuming anything that can share will share – while SPR is only optimistic in ignoring the dependencies, it is not surprising to see more II increase in PA-SPR than in SPR. However, the extra increases do not eliminate the advantage that sharing provides to PA-SPR.

9.3.2 Balancing Sharing and Speculation

This section will examine how the dependence trimming discussed in Section 6.2.3 benefits scheduling. The first test simply compares PA-SPR with the trimming turned off to stock SPR. The results of scheduling are shown in Figure 9.7, with the difference be-

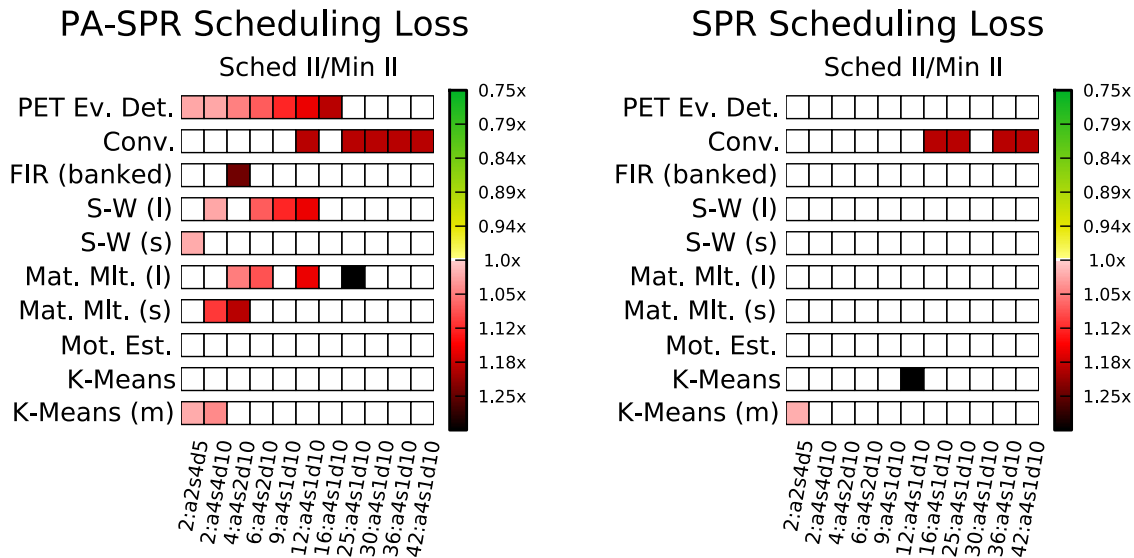


Figure 9.6: Comparison of the schedule II to the minimum II for both predicate aware and regular scheduling.

tween the two shown in the middle sub-graph. In the plots in this section, PA-SPR with trimming is denoted PA-SPR and PA-SPR without trimming is denoted PA-SPR no-trim. Without trimming, PA-SPR is still able to achieve reduced IIs on smaller architectures. However, the large amounts of red and black to the right of the graphs indicate that the final schedule of SPR is much better on larger architectures. This happens because as the architectures grow, more resources are available and the $resII$ shrinks to be smaller than the $recII$. The predicate dependencies that allow sharing increase this $recII$, so on larger architectures, PA-SPR has a larger lower II limit when these dependencies can't be trimmed.

A simple way to overcome this limitation would be to run scheduling using both PA-SPR without trimming and SPR and choose the result with the better scheduled II to continue on to place and route. This would turn any of the red and black in Figure 9.7 to white, as the result would default to that of SPR, and for the remainder it would maintain the improvements.

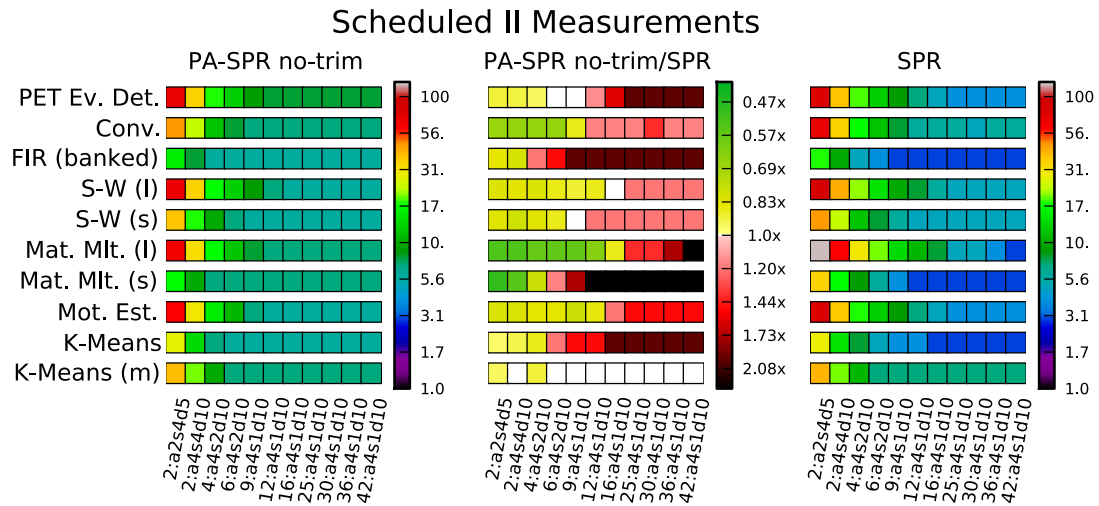


Figure 9.7: Comparison of the scheduled II between PA-SPR with dependence trimming disabled and SPR.

Figure 9.8 uses this hybrid of PA-SPR without trimming and SPR results as a baseline and compares it against the results of PA-SPR with trimming enabled. The results indicate that there is a benefit to using trimming over the simple hybrid method. From Figure 9.8(a), these benefits appear at the larger sizes of architectures where the advantages of PA-SPR begin to taper off. The effect is the most pronounced in the large matrix multiply benchmark, and a graph for this benchmark of the scheduled IIs across the three configurations of scheduling is shown in Figure 9.8(b). From this graph it is apparent that PA-SPR without trimming bottoms-out before regular PA-SPR or SPR. This is caused by the higher recII that comes from the extra dependencies that allow sharing. SPR never shares and so never encounters the higher recII. On larger architectures, PA-SPR is able to recognize that the recII dependencies are inflating the recII beyond what is necessary, and they can be trimmed. This trimming can continue until PA-SPR reaches the same natural recII that SPR uses throughout.

The work on predicate aware scheduling for VLIW processors [SMDL03] didn't appear to use trimming to balance the recII against the resII, so it is natural to consider applying it there. However, the width of VLIW processors tends to be small compared to the number

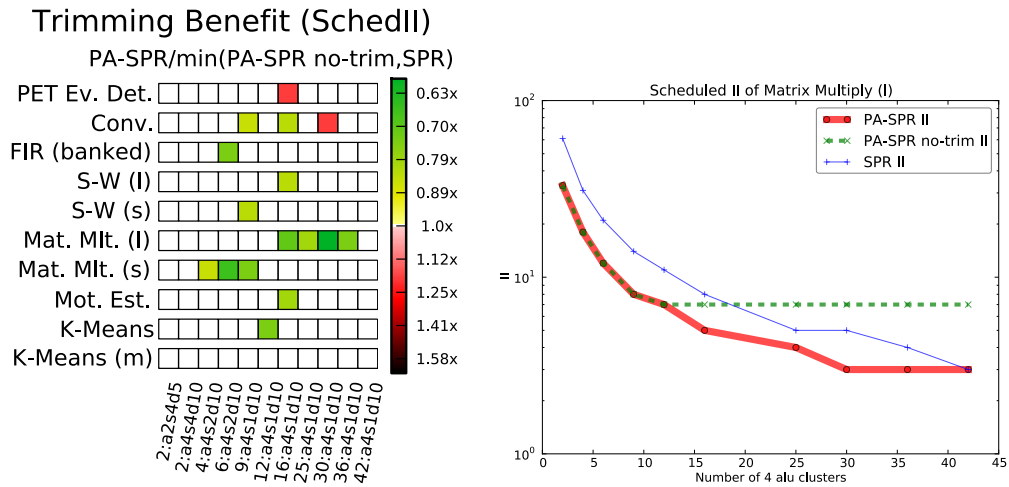


Figure 9.8: Comparison of PA-SPR with dependence trimming to choosing the best of either PA-SPR with no trimming or SPR.

of compute elements in CGRAs. In Figure 9.8(a), the smallest architecture that benefits from the trimming is a 4 cluster, 4 ALU system, for a total of 16 ALUs, but the work of [SMDL03] was looking at 4 and 6-way VLIWs. Larger VLIWs may benefit from trimming in any predicate-aware scheduling, but it appears that it is most helpful in systems that can scale to large numbers of compute units.

This section has presented an optimistic theoretical potential for predicate-aware performance improvement. The results show that the adaptation and improvement of the predicate-aware scheduling technique used for VLIWs effectively produces schedules that maintain that potential. However, these schedules do not take into account the spatial region restrictions and the explicit routing needed for scalable CGRAs. Now that a clear opportunity for performance improvement has been established, the next section will evaluate the placement and routing process by presenting end-to-end results.

9.4 Performance Improvement in PA-SPR

Once all is said and done, the main measurement that reflects the performance of a kernel mapping is the II. The II is the inverse of the throughput, so reducing the final II will

increase the final throughput by the corresponding amount. The previous section already demonstrated that there is potential for improving the II on smaller architectures. The real question now becomes whether or not the adaptations made to the placement and routing algorithms detailed in Chapters 7-8 can carry that potential all the way through to the final mapping.

The results in Figure 9.9 show the final improvements in II using PA-SPR versus SPR. There was no potential speedup shown for architectures of 42 clusters, so they were left out of the final runs. In these graphs, there are *, x and + markers to indicate mappings that did not complete. The * marker indicates that the mapping stopped due to hitting the maximum number of iterations in the outermost schedule/place/route loop, set at 40 for these tests. The x markers indicate that the mapping was killed after 8 hours of running. The + marker indicates the compile ran out of memory, with the limit set at 2.8GB. The boxes in the middle sub-graph are not drawn when a comparison between two runs cannot be made. There is high variability in the results of placement and routing tools using the algorithms that have been adapted in SPR and PA-SPR, so three runs of

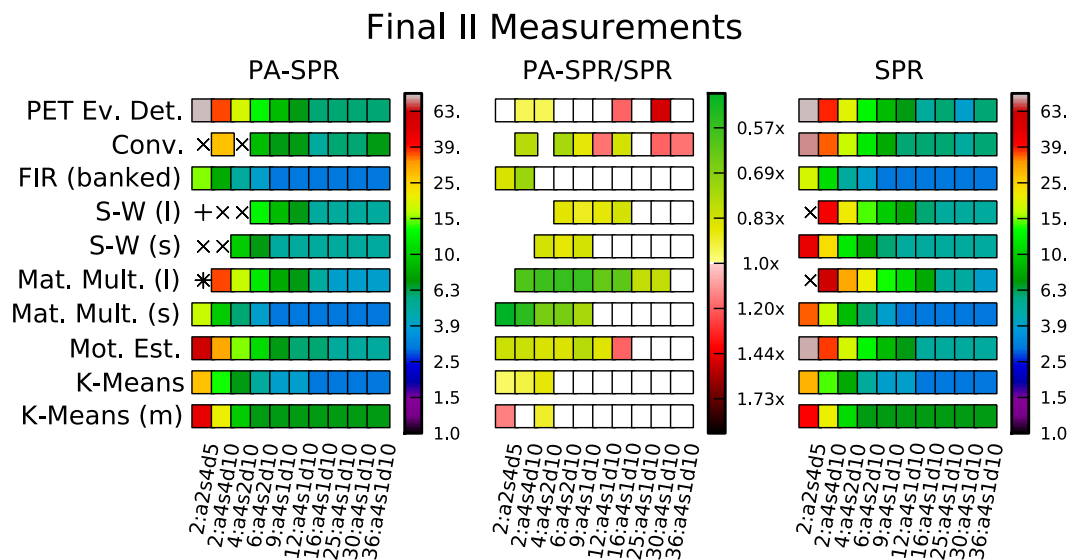


Figure 9.9: Plots showing the final advantage in II of PA-SPR over SPR.

each configuration were run and the best result is used, which is a standard practice in evaluating these algorithms in the FPGA community.

There are more time-out failures in the PA-SPR results than the SPR results, reflecting the slower mapping process due to both the more complex algorithms and the more immature and under-optimized code base. There are a few cases where PA-SPR generates a mapping with a higher II. However, in these cases one can always revert to using the SPR mapping. Overall, when PA-SPR generates a successful mapping, it succeeds in exploiting most of the potential improvement.

Among the runs where PA-SPR reduced the II – the green squares in the middle of Figure 9.9 – there was an average reduction to $.68x$ the baseline SPR II. If this is broadened to all runs where the minimum II indicated there was potential for improvement – averaging the values in the middle of Figure 9.9 wherever there is a green square in the middle of Figure 9.5 – the average reduction is only $.75x$ the baseline. However, the average potential improvement as measured by the minimum II – averaging the green squares in the middle of Figure 9.5 – is $.69x$ the baseline. This means that PA-SPR is able to realize about three-quarters of the potential improvement from predicate-aware sharing. There is probably still some room for improvement, but considering that the minimum II estimate is constructed to be an underestimate of the actual minimum II, the full reduction to $.69x$ may not be possible.

The II losses for the placement/routing portion of PA-SPR and SPR are given in Figure 9.10. There is generally more loss in PA-SPR, which is not surprising given the placement and routing have to satisfy the additional constraints and resource usage of spatio-temporal predicate availability. In the majority of cases, both manage to achieve the initially scheduled II. The cumulative losses from the minimum estimated II to the final mapped II are shown in Figure 9.11. This shows where the losses occurred that result in only a $.75x$ II reduction instead of the full $.69x$ reduction. The losses for PA-SPR that occur in similar configurations as SPR – such as PET event detection, convolution, and motion estimation on larger architectures – are likely to just be situations that are challenging to map. The rest may be either underestimates in the minimum II or opportunities for future algorithmic improvements.

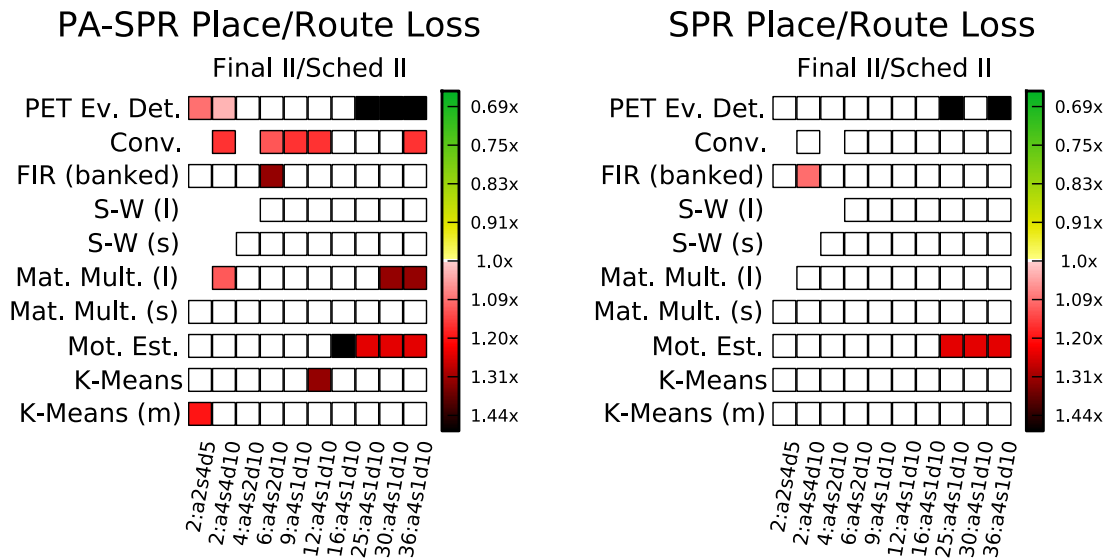


Figure 9.10: Comparison of the increase in II during the placement and routing process between PA-SPR and SPR.

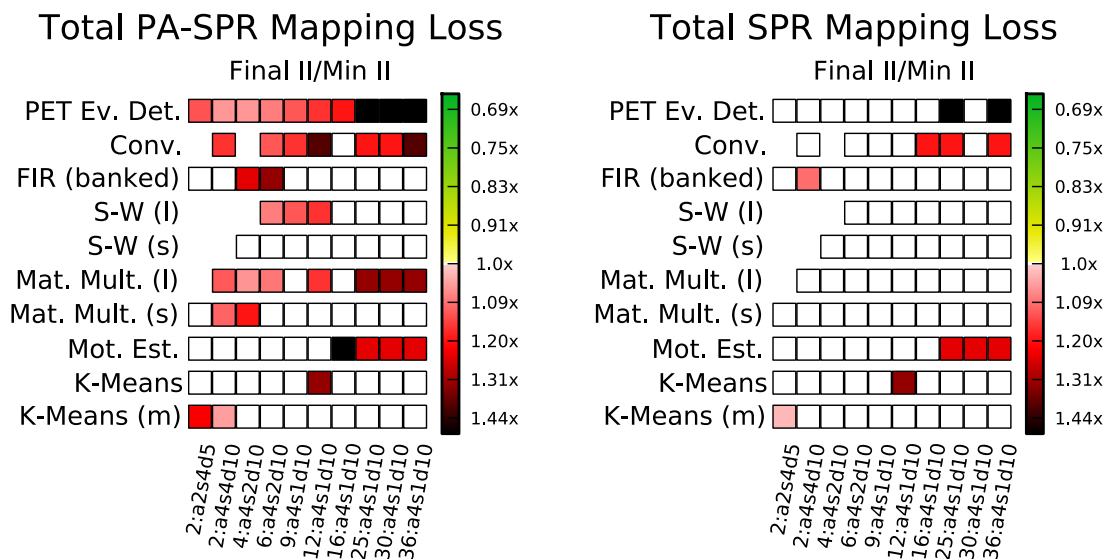


Figure 9.11: Comparison of the total increase in II during mapping in PA-SPR and SPR.

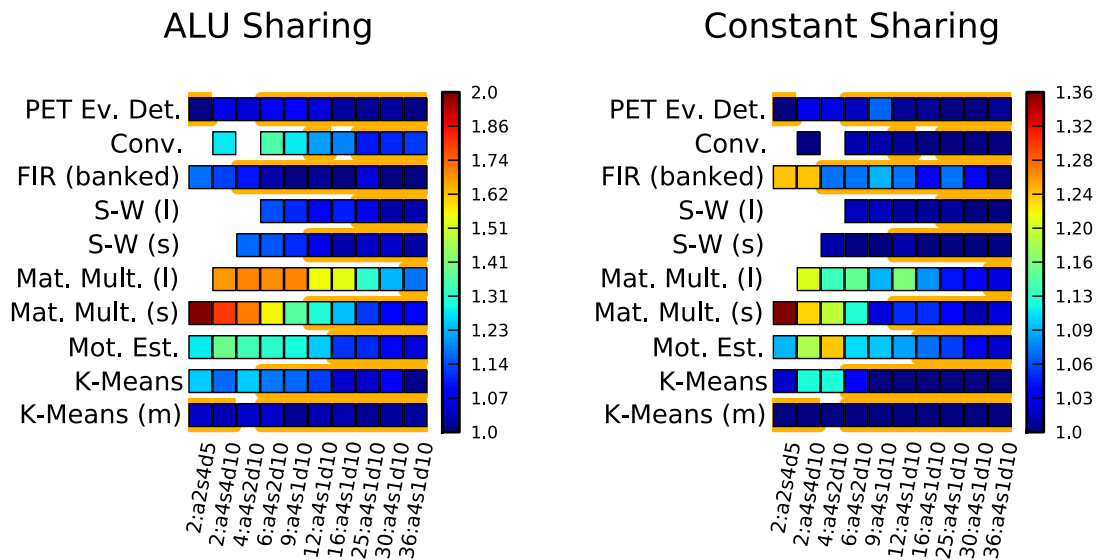


Figure 9.12: Plot of the sharing of compute elements in PA-SPR.

9.5 Sharing in PA-SPR

PA-SPR achieves performance improvement through resource sharing. This section examines the sharing that is the key to PA-SPR boosting the throughput in resource-constrained situations. In this section, the amount of sharing for a resource is counted as the number of operations using a set of device instances divided by the number of device instances being used, where each instance is one phase of a time multiplexed device. Sharing can only happen between operations that can be mapped to the same device, so it is natural to break out the sharing results according to device type. Across many of the devices, there was not much sharing at all. For the bit-wide constants and LUTs, the maximum sharing value was never above 1.02. This is likely because the main source of bit-wide operations in Macah is the compiler synthesized control code for managing the loop flattening – the operations that generate the predicates. Since PA-SPR requires the predicate generation to be unconditional, there is little opportunity for sharing. For the stateful devices, there were never more than 4 instances of sharing a device in any given benchmark.

It is likely the accesses to each resource were easily serializable within the II set by the resource-constrained ALUs.

The sharing for the remaining devices – the ALUs and constant generators – is shown in Figure 9.12. The plots show the sharing for each test with the colors set on a log scale. Note that the top of the scale for ALU sharing is much larger than constant sharing, with ALUs obtaining 200% utilization for the 2:a2s4d5 small matrix multiply test.

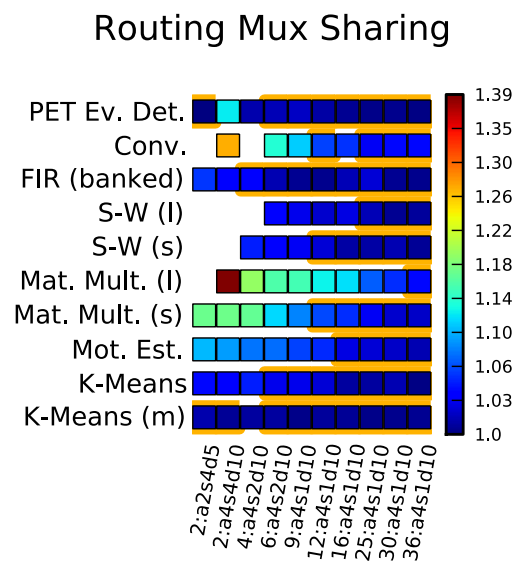


Figure 9.13: Plot of the sharing of routing muxes in the interconnect.

A feature of interest is the sharing beyond where the benchmarks demonstrated a performance improvement. The tests where there was no performance improvement are highlighted in Figure 9.12 and Figure 9.13 to make this easier to see. Even when it was not necessary to reduce the resource usage, PA-SPR is packing operations together on the same devices. As long as the interconnect can support switching the signal routes, packing operations into the same area can help reduce route lengths and latencies between operations. Examining the routing mux sharing, shown in Figure 9.13, also shows the sharing beyond where the II is reduced.

Careful observation of the sharing here shows that there is no ALU or constant sharing on the smallest architecture for PET event detection run. This is because the II rose beyond

64, at which point there is only one predicate line available. It is possible to share with only one predicate line, however, the current implementation of the placer requires a predicate wire for every operation that is sharing a device because it does not handle partial promotion. The router implementation can take advantage of a single predicate bit by sharing with other signals through partial promotion, which is why the large Smith-Waterman exhibits routing mux sharing.

Overall, the performance improvements and sharing results of these experiments demonstrate that the algorithms used in PA-SPR are capable taking advantage of mutually-exclusive control flow to improve performance in resource constrained situations. When compared to an aggressive lower bound, PA-SPR is able to realize roughly three-quarters of the potential reduction in II. Now that PA-SPR provides a strong baseline, future work can focus on gaining the remaining potential improvement. Possibilities for these future improvements are presented in the next chapter.

Chapter 10

CONCLUSIONS AND FUTURE WORK

This dissertation explores different types of resource sharing for modulo-scheduled reconfigurable systems. It has used compiler techniques to share resources in situations at opposite ends of a spectrum – when the configuration memory is limited to a single entry, and when it is abundant. I hope that this exploration helps pave the way for easier and more flexible application acceleration in future reconfigurable systems.

10.1 Summary

The initial back-end compiler that was augmented with the algorithms and techniques presented in this dissertation was built in collaboration with other researchers as part of the Mosaic CGRA tool-chain, described in Chapter 2. This provided the opportunity to evaluate the contributions of the dissertation in a context that provided support for the entire process of accelerating an application, from high-level code compilation all the way to hardware simulation.

The clustering and latency padding techniques introduced in Chapter 3 represent the first novel contribution of this dissertation. While not directly related to resource sharing, they improved the baseline back-end compiler performance by better integrating the VLIW style scheduling with the FPGA oriented placement and routing to create a system that applied well to CGRAs. These created a state of the art platform to use as a starting point for resource sharing and for use by other researchers [VE10, Ylv10].

From there Chapter 4 introduces a way of sharing routing resources that have only a single fixed configuration. Sufficient time multiplexing in the surrounding resources can share the single configuration on the limited resources, as long as they agree on the configuration. By separating the configuration congestion from signal congestion, it becomes possible to use a congestion negotiation process to produce agreement and enable

run-time sharing of these statically configured resources. This sharing process tends to construct larger broadcast networks in place of the more directed communication possible with fully dynamic connections. It also tends to require more interconnect capacity. Thus, the architect must be aware of the trade-off between static and dynamic connections when designing the routing resources. Static resources have reduced configuration area and energy, but increased interconnect area and energy relative to dynamic resources. This makes static resources beneficial in interconnects with narrower bit-widths, such as time-multiplexed FPGAs and the control networks of CGRAs. In larger bit-width interconnects, such as the datapaths of CGRAs, the configuration memory costs have already been amortized across a large bit-width, so maintaining the fully dynamic interconnect is likely the best option.

The most often used method for dealing with control flow when accelerating applications on spatial architectures is to use predicated execution. This is inefficient because it effectively computes the results for all possible control paths and chooses between them at the end. Even though only one of the results will be used at a time, resources and energy is wasted computing all the alternatives. By informing the compiler of alternatives that are mutually exclusive, the compiler has the opportunity to share resources between the computations, reducing this inefficiency. Chapter 5 combines ideas and techniques from work on optimizing compilers and modulo-scheduling for VLIWs to produce a framework for representing mutually exclusive control flow. Certain choices in the representation, such as limiting consideration to a tree structure, were made to produce a representation that allows the compiler to identify and manage sharing opportunities throughout the compilation process. The tree-structure is particularly fruitful because it allows the router to adaptively move a signal up the tree for configuration negotiation, but retain its original mutual-exclusion properties to share wiring with other signals. This chapter also covered the integration of the CDT and time information, which is critical in a modulo-scheduled system where software pipelining introduces interactions between operations that are scheduled to start at different times. Additionally, this chapter introduces an abstraction that will allow the compiler to manage the distribution of the

predicate signals needed to manage the control flow based sharing without introducing any global resources that would create scaling bottlenecks.

With the framework of ideas clearly defining predicate-aware sharing in place, the changes needed to support such sharing are presented in Chapters 6-8, relying heavily on the abstractions from Chapter 5. The scheduling is an adaptation of VLIW predicate-aware scheduling. In order to support the maximum gains in marginally resource constrained situations, a dependence trimming technique is introduced that balances sharing operations against speculatively executing them to ensure that new iterations of the loop can be initiated as fast as possible.

Supporting predicate-aware sharing in the placement process requires adding the proper costs and constraints to the existing simulated-annealing placer. The aggregate execution condition developed in Chapter 5 is used to allow compatible placement of multiple operation on the same devices. A carefully constructed cost model tracks the costs of routing the predicates for sharing into the regions where they are needed within the capacity limitations of the architecture. The cost model is carefully constructed to allow incremental cost updates, maintaining the run-time efficiency of the placer.

Several new techniques are introduced for managing predicate-aware sharing during routing. The separation of signal and configuration congestion from Chapter 4 is used to separate the conditions used for configuration selection from those used to track mutually-exclusive signaling. This enables the negotiation-based router to use partial promotion to negotiate for configuration settings when the predicates needed for configuration switching are not available, while still ensuring that signals can use the most specific conditions for tracking routing resource usage. This makes it possible for signals to follow the same path or diverge anywhere in the architecture as long as they are mutually exclusive, but their paths are allowed to converge only in regions where predicates are available to do the appropriate predicate switching. Routing signal trees in a predicate-aware manner introduces complications for partial route re-use, especially in the cases where the source and sink conditions differ. A new algorithm for partial route re-use is introduced to handle these complications, allowing re-use not only between paths from the same source, but also between paths from mutually-exclusive source op-

erations mapped to the same physical device. This re-use is based on careful creation and initialization of separate search queues based on relationships between signals from the topological positions of their conditions in the CDT.

All of these sharing techniques were evaluated in the context of the Mosaic tool-chain, demonstrating performance improvements for clustering, latency padding, and predicate-aware sharing, as well as channel-width improvements for static sharing.

10.2 Conclusions

The biggest contributions of this dissertation often hinged on simply choosing the right the point of view for a given problem. Once the proper framing for the problem is found, the solution almost presents itself. For example, being able to separate resource contention into the two facets of configuration and time-multiplexed usage was valuable in both the static sharing of Chapter 4 and the predicate-aware “tunneling” of Chapter 8. In fact, because the problems could be seen from the same point of view, the mechanisms for solving them could be easily adapted across two very different concepts of sharing.

Making the decision to identify mutual-exclusion using the limited control-dependence tree instead of the more general control-dependence graph or the more complicated Predicate-Query System enabled flexibility in the predicate-aware placement and routing algorithms. Reasoning over the simpler tree structure allows the compiler to rapidly make safe decisions about *changing* predicate dependence, which enables placement to adjust where predicates must be distributed to and allows routing to accomplish “tunneling” when predicates are not available locally. While the algorithms and techniques presented provide demonstrated benefit, the abstractions themselves hold the potential for future advances through a novel point of view.

Now that many of the complexities and limitations of sharing have been revealed over the course of the work behind this dissertation, it is a good point to look back and try to decipher when the benefits of sharing are worth the costs. The actual benefits that any of these algorithms provide are highly dependent on their context. For example, the static

sharing in Chapter 4 is able to route using statically configured interconnect of almost the same channel width as the fully dynamic interconnect. This is a major improvement over simply using each static resource once, and provides savings in the configuration memory. However, if the optimization metric is area-energy for the top-level 32-bit grid interconnect, overall it is cheaper to just use a fully dynamic interconnect. This is because the shared static configurations broadcast values to all destinations that will be using the static route, instead of only the places that the current signal needs to go. In the top level 32-bit interconnect, that is a significant energy cost that overwhelms the configuration memory savings. This means that static route sharing is much more valuable where the configuration memory costs are higher than the signal broadcast costs. If a CGRA is being designed with a deep configuration memory and a portion of single-bit wide configurable logic, it makes sense to use the static sharing. In a completely 32-bit bus system with long wires and a shallow configuration depth, the sharing is probably not worth the extra compile time, CAD complexity, and architecture design complexity of implementing the sharing algorithms and static configurations.

The context used for evaluation of predicate-aware sharing is very optimistic. The architecture sizing provided ample sharing opportunity, which helps to highlight how well the algorithms realize that opportunity. Predicate-aware sharing is only effective when the following criteria are met:

- The application is resource constrained – the res_{II} is higher than the rec_{II} .
- The application has opportunity for mutually-exclusive sharing – there must be enough control flow such that a large fraction of the computation does not need to be performed on each iteration.
- The hardware has the configuration memory to support the sharing – with the hardware organization chosen in this evaluation, a large portion of it may be wasted.

- Compute elements should be able to share operations – the evaluation used a universal ALU as a base computation unit, maximizing the pool of operations that could be mapped together on the same devices.
- The predicates can be distributed in a timely manner – using predicate aware sharing removes the ability to speculatively execute some operations, increasing the recurrence II.

With many high-performance kernels, it is not difficult to find a resource-constrained problem. Often it is simply a matter of dialing up a tuning knob to work on more of the problem in parallel. Ensuring the kernel has the proper control flow is another matter. Alternatively, there may be situations where many kernels are in use in a system at the same time, each relegated to a small portion of the chip. Predicate-aware sharing will allow more kernels to share the same chip, with less wasted computation.

Almost none of the manually flattened benchmarks were used because they simply did not have any opportunity for sharing. On the other hand, the enhanced loop flattening in Macah tends to build a large partition at the top of the CDT by flattening nested loops into individually mutually-exclusive blocks. This means that kernels that need to adapt and adjust online can be written with natural control flow, and the tool-chain will exploit sharing across that control flow. Another common pattern is setup-compute-reduce/write-back, and if the compute is in a `for` loop, enhanced loop flattening will split these into three mutually exclusive stages. Predicate-aware sharing can then hide some of the setup and tear-down costs.

One big weakness of the predicate-aware sharing as presented in this dissertation is the simple hardware model it uses for switching configurations. The one-hot nature of predicates being directly piped into address bits makes a large portion of the configuration array unusable, as seen in Figure 8.8. I believe a different hardware design could alleviate some of this problem, but would lead to increased complexity in the hardware. Some options are discussed in Section 10.3.2.

Mapping all word-wide computations onto a single-cycle ALU element is also a very optimistic assumption, as it included operations with significant hardware cost such as

multiply and shift in the same unit. This makes the pool for operations that can share a device bigger, amplifying the opportunity for sharing. In a more heterogeneous architecture, the operations will be forced into separate pools, adding more constraints on what can share. If area utilization is a primary optimization metric, a heterogeneous architecture is a likely starting point, diminishing the opportunity for predicate-aware sharing. However, digital logic scaling is starting to reach a utilization wall, where many more transistors can be put on a chip than can be effectively cooled at full speed and power [VSG⁺10]. This leads to the existence of “dark silicon,” portions of the chip that are not powered or switching. This dark silicon could be exploited in a CGRA by including all operations in a single large ALU where the configuration effectively only powers the needed functionality. While there has been recent research into ways of including the best mix of heterogeneous compute units [AYP⁺06, VE10], the unified ALU design may be more beneficial in the face of this new utilization wall. As long as there is not a significant interconnect power overhead, predicate-aware sharing will be able to exploit these large ALUs and reduce the amount of computation that is speculatively executed and thrown away, wasting power and heating the chip. Unfortunately, route tunneling through regions without predicates can lead to the same signal over-broadcast that happens with static sharing, so it remains to be seen whether or not the computational power savings would outweigh the broadcast costs.

If the predicate delivery is too slow, the resulting increase in recurrence II can offset the benefits of sharing. In the hardware model used for this evaluation the path from the last register a predicate is stored in, through the configuration memory, to a device that needs to be configured will likely be a critical path, setting the maximum clock frequency of the architecture. This critical path can be reduced by inserting a register between the configuration array output and the compute elements, but this requires predicates be available a cycle earlier in order to support sharing. Forcing this extra cycle on predicates is not a problem for operations off the critical recurrence loop. However, this will increase the size of any recurrence loops with predicate signals in them. This will lead to the dependence trimming algorithm needing to adjust the $recII/resII$ balance on smaller architectures, potentially limiting the benefits demonstrated in Figure 9.8 to smaller architectures.

10.3 Future Work

Moving forward, there are many unexplored potential avenues of investigation. The algorithms described in this dissertation were put together to create a complete end-to-end system. This provided a solid demonstration that there is a benefit to PA-SPR. However, instead of finding the optimal solution from a set of alternatives, the easiest solution was often used to simply move forward. This section looks back to present some of those possible alternatives, along with looking forward towards new areas for exploration.

10.3.1 Reducing Predicate Pressure

The hardware model used in this work uses predicate signals to directly control the addressing of configuration memory. This means that only a few predicates will be supported in each region before the size of configuration memory grows to impractical sizes. It is important to make the best use of any predicates required in the region, and to reduce the demand for predicates in a region when possible.

One possible way to reduce the predicate demand is by not requiring predicates when all operations mapped to the same device require the same configuration – for example, they are all AND operations in an ALU. This is similar to an idea presented in [MLC⁺92], where the same instruction in mutually exclusive segments is merged into one instruction that executes unconditionally. This can be supported when transitioning from the placement to the routing phase by not requiring a predicate to be routed when the operations match. This is a straight-forward test, as long as the configuration encoding for operations is known between placement and routing. In PA-SPR, this information is only available in a plug-in configuration generator that runs after the mapping is completed. It is likely that if operations are sharing a resource – even if the operations require the same configuration – then the routes leading to those operations may need the predicates to share the incoming routing resources anyway. Because of this, the extra coding effort required to provide the configuration information earlier did not seem to be worth the potential predicate reduction for this initial study.

The evaluation in Chapter 9 showed that there is a large amount of II increase in the placement and routing sections. Some of this increase is initiated by the placer incrementing the II when it cannot successfully reduce the predicate requirements to within the capacity of the region. In SPR, the local adjustment of latency padding proved to be successful at dealing with constraint violations without resorting to a global II increase. A similar local method might be found for dealing with predicate gateway congestions, such as promoting individual operations from the least useful predicates within a region, or promoting all operations from the least useful predicate, for some yet-to-be-defined notion of “least useful.”

10.3.2 Alternate Hardware Implementations

In the illustration for slot expansion, Figure 8.8, it is apparent that the mutual-exclusivity of the predicates renders many of the configuration memory locations unusable. It may be possible to use an alternate hardware structure for supporting predicate-based configuration switching that will reduce this waste. One possibility is to support more predicate lines than there are address bits, but use some configurable hardware to encode the one-hot predicates into binary encoded values for denser access to the configuration array. Another possibility is to replace the address indexed configuration memory with a content addressable memory (CAM). This allows a configuration to be directly addressed by the predicate values that require it. An even more exotic option would be to allow a partitionable trade-off between the configuration array and local memories. Unfortunately, properly choosing the trade-off would probably either involve a complicated balancing algorithm or programmer allocation.

10.3.3 Optimizing Predicate Usage

In both the placement and the routing, there is the opportunity to choose the predicates that are used within the region to enable sharing. Once the placer has finished and is transitioning to the routing, it establishes routes for the predicates representing the conditions of any operations that are sharing resources. When the router is trying to resolve control congestion, it will examine the signals that have control congestion and

attempt to bring their predicates into the region to support switching configurations. However, in both of these cases, there may be a more optimal choice of predicates that will accomplish the same sharing.

In both of these cases, the goal is to find a set of predicates that will partition a given set of mutually-exclusive operations (or signals) with the least burden on the enclosing region's predicate gateway. The difficult part of this problem is that multiple sets of operations/signals co-exist in the same region and must share the gateway predicate lines. The optimal solution for the whole region may not be the same as the locally optimal solution for each resource.

Finding the locally optimal solution is straightforward once the problem is set up properly. Begin by considering the set of operations on a given resource belonging to mutually-exclusive CDT condition nodes. For routing resources, the operation could be seen as something such as "pass input 2 to the output". All of the operations are mutually exclusive, so the least common ancestor of each pair is a partition node. The CDT can be pruned down so that the nodes with operations in them are leaves, and the greatest least-common ancestor is the root. Any condition or partition node branch that does not contain one of the operations of interest can be pruned as well.

This pruning process greatly simplifies the tree structure. The least common ancestors for any pairs of leaves – the branching points in the simplified tree – are all partition nodes. Every condition node will be left with exactly one child. The operations are mutually exclusive, so no operation will ever be in a condition that is the ancestor of another operations condition, resulting in exactly one operation per leaf.

Each operation will require a specific setting in the configuration word. Some of these settings may be the same, and so operations can be colored by the setting they require. For example, all additions could be colored red, while subtractions are colored yellow, or routing from input 1 is blue and input 2 is green. Operations with the same color, and therefore setting, do not need a predicate to switch between them at run-time, while different colors do. When a predicate is available in the region, it allows all operations below its associated condition to have a different setting than the rest of the tree, which can be seen as cutting the edge from the parent to the condition in the tree. The current

solution used by the placer and the router simply chooses the predicates for all of the leaves, but it may be possible to partition the tree with fewer predicates by taking them from farther up the tree.

A minimum cut that partitions the colored nodes where each of these edges is weighted 1 is equivalent to choosing a minimum number of predicates. This is the Multi-terminal Cut Problem [DJP⁺92, DJP⁺94], which was shown to be NP-hard in the general case. However, Erdős and Székely [ES94] provided an optimal polynomial time algorithm for trees, which is applicable to these simplified graphs. Once again, the decision to limit the mutual-exclusion representation to a tree provides an unexpected benefit when reasoning about the conditions.

This provides an elegant solution to choosing the optimal set of predicates locally. Unfortunately, the union of all of these locally optimal solutions across the region may be far from optimal. The algorithm given in [ES94] is a two pass process that calculates and then uses penalty weights across the tree. It may be possible to communicate weight information across the trees that fall within a region, either through simultaneously processing the trees or through an iterative process that finds local solutions and updates global weights, but that is left for future exploration.

10.3.4 Merging Select Operations Into Routing

As part of the if-conversion and flattening process in Macah, select operations are inserted into the data-flow to choose between the results of two different control paths. These operations take three inputs – the two values to select between and a predicate signal that represents a condition in the CDT. These select operations are essentially routing the appropriate value on in the data-flow. With a predicate-aware router, the router already will use routing muxes as needed, along with a predicate value in the region, to switch between values based on the current control-flow condition. Currently, select operations are performed by an ALU, but it should be possible to take those select operations out and allow the router to manage passing the proper signal along. This would free up the ALUs to perform more useful computations. The select operations that are candidates for merging can be identified by first checking the source operation providing the select input

to see if it is one of the operations providing a predicate in the CDT. If the corresponding CDT condition partitions the conditions of the source values that the select operation is choosing between, then the select operation can be removed, with the source values routed directly to the destinations of the select operation. The router would need to allow a port to have multiple routes to the same input, and set the sink conditions to be the same as the source conditions for the two new routes, but the existing negotiation mechanisms should take care of the rest of the sharing process.

10.3.5 Power Implications of Predicate-Aware Sharing

It would be interesting to investigate the effect that predicate-aware sharing has on power usage. Predicate-aware sharing is effectively eliminating the speculation that a spatial execution would otherwise be performing. This may decrease the power to perform a particular computation because less is speculatively computed and thrown away. However, it can increase computational density, and may have unexpected power implications in terms of routing. Since routing is a significant source of power usage in CGRAs [VE10], it is unclear whether predicate-aware sharing would be a net gain or loss.

10.3.6 Cross-iteration Mutual Exclusion

Currently, PA-SPR is only aware of mutual-exclusion relationships between operations within the same loop iteration. The cross-iteration mutual exclusion discussed in Section 5.2.4 is one possible future source of sharing. A related notion of cross-iteration mutual exclusion may come from the flattening process in Macah. Consider a kernel that consists of three loops in sequence, A , B , and C – perhaps setup, compute, and tear-down loops for the kernel. Macah will turn these three loops into three mutually exclusive blocks, and synthesize the appropriate logic to switch between them at run-time. If B is guaranteed to execute for n iterations, then we know that C will never execute in the same iteration as A , nor within n iterations of the last time that A executed. Instead of being limited to sharing operations from A and C that have the same start time within an iteration, an operation in C with a start time of s can share with an operation in A with start times $s, s + II, \dots, s + n * II$. This widened range will increase the opportunities for sharing, if

the information can be effectively communicated to PA-SPR, and the compatibility checks in the aggregate execution condition can be adapted to track the ranges.

10.3.7 The Counter is Dead – Long Live the Counter!

Predicate-aware sharing can be pushed to the extreme by viewing the phases of the modulo schedule as a set of partitioning conditions, and the modulo-counter as encoded predicates for that set of conditions. The dedicated modulo-counter can be removed from the configuration hardware and replaced with a computed counter in the data-path. In the transition from placement to routing, if the same operations are used across phases, the phase predicates are not needed in the region, and the starting situation from the router's point of view becomes that depicted in Figure 8.6, where all of the phases end up in the same configuration slot. It is more likely that at least some of the settings will differ within the region, at which point the minimum partitioning set of predicates (including the new phase predicates) should be chosen for the region. This will lead to irregular slot tables like those depicted in Figure 8.5, except that there may not be a full partitioning across the phases. After a few iterations of routing, new predicates can be selected for routing to the region, which can also include the phase predicates as candidates. In the end, each region will be able to use its own independent mixture of phase bits and predicate bits to switch between configurations. This can be seen as a form of configuration compression. In fact, with enough similarity across routes and operations within the region, and a way to pack the configurations together such as a CAM, it may be possible to generate a valid mapping for a kernel with an Π that is greater than the depth of the configuration memory. Getting this to all work out efficiently is a big challenge and not likely to be practical, especially once you try to start including stateful operations in all of this, but it an interesting thought experiment to try and understand all of the implications.

10.4 Epilogue

In the end, all of this resource sharing is trying to take advantage of the available hardware in the most efficient way possible. The algorithms in this dissertation have been able to take advantage of flexibility where it is available, and negotiate for cooperation other-

wise. This trade-off is important for allowing the programmer the flexibility to program the natural control flow for a program, but implement it efficiently within the bounds of flexibility that allow fast and scalable architectures. The sharing introduced in this dissertation is not the final solution in striking this balance, but it is certainly a step in the right direction.

BIBLIOGRAPHY

- [ACS03] I. Arsovski, T. Chandler, and A. Sheikholeslami. A Ternary Content-Addressable Memory (TCAM) Based on 4T Static Storage and Including a Current-Race Sensing Scheme. *IEEE Journal of Solid-State Circuits*, 38(1):155–158, Jan 2003.
- [AKPW83] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–189, New York, NY, USA, 1983. ACM.
- [ASN⁺00] Jason Helge Anderson, Jim Saunders, Sudip Nag, Chari Madabhushi, and Rajeev Jayaraman. A Placement Algorithm for FPGA Designs with Multiple I/O Standards. In *International Workshop on Field-Programmable Logic and Applications*, pages 211–220, London, UK, 2000. Springer-Verlag.
- [AYP⁺06] Minwook Ahn, Jonghee W. Yoon, Yunheung Paek, Yoonjin Kim, Mary Kiemb, and Kiyoun Choi. A Spatial Mapping Algorithm for Heterogeneous Coarse-Grained rReconfigurable Architectures. In *Design Automation and Test in Europe*, pages 363–368, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.
- [BBKG07] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array. *Lecture Notes in Computer Science*, 4419:1–13, 2007.
- [BC96] Brenda S. Baker and Edward G. Coffman. Mutual Exclusion Scheduling. *Theoretical Computer Science*, 162(2):225 – 243, 1996.
- [BELP94] G. Bilsen, M. Engels, R. Lauwereins, and JA Peperstraete. Static Scheduling of Multi-Rate and Cyclo-Static DSP-Applications. In *Workshop on VLSI Signal Processing*, pages 137–146, 1994.
- [BELP96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-Static Dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, Feb 1996.
- [BFC00] Michael Bender and Martín Farach-Colton. The LCA Problem Revisited. In *Latin American Theoretical INformatics*, pages 88–94. Springer Berlin / Heidelberg, 2000.

- [BH01] T. Basten and J. Hoogerbrugge. Efficient Execution of Process Networks. In *Communicating Process Architectures: World Occam and Transputer User Group Technical Meeting*, 2001.
- [BHLM01] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*. Kluwer Academic Publishers Norwell, MA, USA, 2001.
- [BJW07] Michael Butts, Anthony Mark Jones, and Paul Wasson. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 55–64, Washington, DC, USA, 2007. IEEE Computer Society.
- [BKM⁺04] D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, July 2004.
- [BL93] Joseph T. Buck and Edward A. Lee. Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model. Technical report, University of California, Berkeley, 1993.
- [Bou06] F. Bouwens. Power and Performance Optimization for ADRES. Master’s thesis, Delft University of Technology, 2006.
- [BR97a] Vaughn Betz and Jonathan Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In *International Workshop on Field-Programmable Logic and Applications*, 1997.
- [BR97b] Oliver Bringmann and Wolfgang Rosenstiel. Resource Sharing in Hierarchical Synthesis. In *IEEE/ACM International Conference on Computer Aided Design*, pages 318–325, Washington, DC, USA, 1997. IEEE Computer Society.
- [Cal02] Timothy J. Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures*. PhD thesis, University of California, Berkeley, 2002.
- [Car05] João M. P. Cardoso. Dynamic Loop Pipelining in Data-Driven Architectures. In *Conference on Computing Frontiers*, pages 106–115, New York, NY, USA, 2005. ACM.
- [CCH⁺00a] E. Caspi, M. Chu, R. Huang, J. Yeh, Y. Markovskiy, A. DeHon, and J. Wawrzynek. Stream Computations Organized for Reconfigurable Execution (SCORE): Introduction and Tutorial. Technical report, University of California, Berkeley, 2000.

- [CCH⁺00b] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE). In *International Workshop on Field-Programmable Logic and Applications*, pages 605–614, 2000.
- [CCP06] D. Chen, J. Cong, and P. Pan. FPGA Design Automation: A Survey. *Foundations and Trends® in Electronic Design Automation*, 1(3):195–334, 2006.
- [CE06] Allan Carroll and Carl Ebeling. Reducing the Space Complexity of Pipelined Routing Using Modified Range Encoding. In *International Workshop on Field-Programmable Logic and Applications*, September 2006.
- [CFBE98] 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*, pages 116–125, Apr 1998.
- [CFF⁺99] D.C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths. In *Conference on Advanced Research in VLSI*, pages 23–40, Atlanta, 1999.
- [CFS90] Ron Cytron, Jeanne Ferrante, and V. Sarkar. Compact Representations for Control Dependence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–351, New York, NY, USA, 1990. ACM.
- [CFV⁺07] Allan Carroll, Stephen Friedman, Brian Van Essen, Aaron Wood, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Designing a Coarse-grained Reconfigurable Architecture for Power Efficiency. Technical report, Department of Energy NA-22 University Information Technical Interchange Review Meeting, 2007.
- [CHB07] Eric Cheung, H. Hsieh, and F. Balarin. Automatic Buffer Sizing for Rate-Constrained KPN Applications on Multiprocessor System-on-Chip. In *IEEE International High Level Design Validation and Test Workshop*, pages 37–44, Nov. 2007.
- [CHM08] Nathan Clark, Amir Hormati, and Scott Mahlke. VEAL: Virtualized Execution Accelerator for Loops. In *IEEE International Symposium on Computer Architecture*, pages 389–400, Washington, DC, USA, 2008. IEEE Computer Society.
- [CHW00] T.J. Callahan, J.R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *IEEE Computer*, 33(4):62–69, Apr 2000.

- [CLG02] Josep M. Codina, Josep Llosa, and Antonio González. A Comparative Study of Modulo Scheduling Techniques. In *Proceedings of the International Conference on Supercomputing*, pages 97–106, New York, NY, USA, 2002. ACM.
- [CLJ⁺01] W.S. Coates, J.K. Lexau, I.W. Jones, S.M. Fairbanks, and I.E. Sutherland. FLEETzero: An Asynchronous Switching Experiment. In *International Symposium on Asynchronous Circuits and Systems*, pages 173–182, 2001.
- [CS00] Pak K. Chan and Martine D. F. Schlag. New Parallelization and Convergence Results for NC: A Negotiation-Based FPGA Router. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 165–174, New York, NY, USA, 2000. ACM.
- [CSEM00] P.K. Chan, M.D.F. Schlag, C. Ebeling, and L. McMurchie. Distributed-Memory Parallel Routing for Field-Programmable Gate Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(8):850–862, aug 2000.
- [CW04] Danny Z. Chen and Xiadong Wu. Efficient Algorithms for k-Terminal Cuts on Planar Graphs. *Algorithmica*, 38:299–316, 2004. 10.1007/s00453-003-1061-2.
- [Den80] J.B. Dennis. Data Flow Supercomputers. *IEEE Computer*, 13(11):48–56, Nov 1980.
- [DGH⁺01] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, et al. *Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java*. Electronics Research Laboratory, College of Engineering, University of California, 2001.
- [DHB89] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped Loop Support in The Cydra 5. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, New York, NY, USA, 1989. ACM.
- [Din97] Benoît Dupont de Dinechin. Parametric Computation of Margins and of Minimum Cumulative Register Lifetime Dates. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 231–245, London, UK, 1997. Springer-Verlag.
- [DJP⁺92] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The Complexity of Multiway Cuts (extended abstract). In *ACM Symposium on Theory of Computing*, pages 241–251, New York, NY, USA, 1992. ACM.

- [DJP⁺94] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The Complexity of Multiterminal Cuts. *SIAM Journal on Computing*, 23:864–894, August 1994.
- [DSCVAM08] Bjorn De Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei. Placement-and-Routing-Based Register Allocation for Coarse-Grained Reconfigurable Arrays. In *ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 151–160, New York, NY, USA, 2008. ACM.
- [Ebe97] Carl Ebeling. Whither configurable computing? In *Hawaii International Conference on System Sciences*, volume 1, pages 713–713, 1997.
- [Ebe02a] Carl Ebeling. Compiling to Coarse-Grained Adaptable Architectures. Technical Report UW-CSE-02-06-01, University of Washington Technical Report, 2/22/2002 2002.
- [Ebe02b] Carl Ebeling. The General Rapid Architecture Description. Technical Report UW-CSE-02-06-02, University of Washington Technical Report, 2/22/2002 2002.
- [EBLP94] M. Engels, G. Bilson, R. Lauwereins, and J. Peperstraete. Cycle-Static Dataflow: Model and Implementation. In *Asilomar Conference on Signals, Systems and Computers*, volume 1, 1994.
- [ECF96] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. In Reiner W. Hartenstein and Manfred Glesner, editors, *International Workshop on Field-Programmable Logic and Applications*, pages 126–135. Springer-Verlag, Berlin, 1996.
- [ECF⁺97a] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg. Mapping Applications to the RaPiD Configurable Architectures. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, page 106, Washington, DC, USA, 1997. IEEE Computer Society.
- [ECF97b] C. Ebeling, D.C. Cronquist, and P. Franklin. Configurable Computing: The Catalyst for High-Performance Architectures. In *International Conference on Application-Specific Systems, Architectures and Processors*, pages 364–372, 1997.
- [EH06] Ken Eguro and Scott Hauck. Armada: Timing-Driven Pipeline-Aware Routing for FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 169–178, New York, NY, USA, 2006. ACM Press.

- [EH11] Carl Ebeling and Scott Hauck. Mosaic Research Group, University of Washington. [Online] <http://www.cs.washington.edu/research/lis/mosaic/>, August 2011.
- [EKLN07] Roe Engelberg, Jochen K nemann, Stefano Leonardi, and Joseph (Seffi) Naor. Cut Problems in Graphs with a Budget Constraint. *Journal of Discrete Algorithms*, 5(2):262 – 279, 2007. 2004 Symposium on String Processing and Information Retrieval.
- [EMHB95] C. Ebeling, L. McMurchie, S.A. Hauck, and S. Burns. Placement and Routing Tools for the Triptych FPGA. *IEEE Transactions on Very Large Scale Integration Systems*, 3(4):473 –482, dec. 1995.
- [ES94] P ter L. Erdős and L szl  A. Sz kely. On Weighted Multiway Cuts in Trees. *Mathematical Programming*, 65:93–105, 1994. 10.1007/BF01581691.
- [FCVE⁺09] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. SPR: An Architecture-Adaptive CGRA Mapping Tools. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 191–200, New York, NY, USA, 2009. ACM.
- [FGTC07] R. Ferreira, A. Garcia, T. Teixeira, and J.M.P. Cardoso. A Polynomial Placement Algorithm for Data Driven Coarse-Grained Reconfigurable Architectures. In *IEEE Computer Society Annual Symposium on VLSI*, pages 61–66, 2007.
- [FH06] Johannes Fischer and Volker Heun. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In Moshe Lewenstein and Gabriel Valiente, editors, *Combinatorial Pattern Matching*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer Berlin / Heidelberg, 2006.
- [FKPM05] Kevin Fan, Manjunath Kudlur, Hyunchul Park, and Scott Mahlke. Cost Sensitive Modulo Scheduling in a Loop Accelerator Synthesis System. In *Conference on Computing Frontiers*, pages 219–232, Washington, DC, USA, 2005. IEEE Computer Society.
- [FO83] Jeanne Ferrante and Karl J. Ottenstein. A Program Form Based on Data Dependency in Predicate Regions. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 217–236, New York, NY, USA, 1983. ACM.
- [FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

- [GB03] M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In *European Symposium on Programming (ESOP)*, volume 2618/2003, pages 7–11. Springer, 2003.
- [GLP06] Padmini Gopalakrishnan, Xin Li, and Lawrence Pileggi. Architecture-Aware FPGA Placement Using Metric Embedding. In *Design Automation Conference*, pages 460–465, New York, NY, USA, 2006. ACM.
- [GNVV04] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Vissers. A Quantitative Analysis of the Speedup Factors of FPGAs Over Processors. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 162–170, New York, NY, USA, 2004. ACM.
- [GSM⁺99] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *IEEE International Symposium on Computer Architecture*, 1999.
- [GTK⁺02] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, New York, NY, USA, 2002. ACM.
- [Har01] Reiner Hartenstein. Coarse Grain Reconfigurable Architecture (embedded tutorial). In *Asia and South Pacific Design Automation Conference*, pages 564–570, Yokohama, Japan, 2001. ACM Press.
- [HD08] S. Hauck and A. DeHon, editors. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Pub, 2008.
- [HHHN00] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. KressArray Xplorer: A New CAD Environment to Optimize Reconfigurable Datapath Array Architectures. In *Asia and South Pacific Design Automation Conference*, pages 163–168, 2000.
- [HLL⁺03] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng. Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M03/25, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, 2003.
- [Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

- [HSM03] P. Heysters, G. Smit, and E. Molenkamp. A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems. *The Journal of Supercomputing*, 26(3):283–308, 2003.
- [Huf93] Richard A. Huff. Lifetime-Sensitive Modulo Scheduling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–267, New York, NY, USA, 1993. ACM.
- [HW97] J.R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21. IEEE Computer Society Press, 1997.
- [HZ04] M. Hagog and A. Zaks. Swing Modulo Scheduling for GCC. In *GCC Developer's Summit*, volume ?, pages 55–65, 2004.
- [IBY⁺07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, New York, NY, USA, 2007. ACM.
- [Joh75] Donald B. Johnson. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.
- [JS96] R. Johnson and M. Schlansker. Analysis Techniques for Predicated Code. In *Conference on Computing Frontiers*, page 100, Los Alamitos, CA, USA, 1996. IEEE Computer Society.
- [Kar72] Richard M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103, New York, NY, 1972. Plenum Press.
- [KDR⁺00] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient Conditional Operations for Data-Parallel Architectures. In *Conference on Computing Frontiers*, pages 159–170, New York, NY, USA, 2000. ACM.
- [KDR⁺02] U.J. Kapasi, W.J. Dally, S. Rixner, J.D. Owens, and B. Khailany. The Imagine Stream Processor. In W.J. Dally, editor, *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 282–288, 2002.
- [KGV83] S. Kirkpatrick, Jr. Gelatt, C. D., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

- [KJ97] Sabine Öhring Klaus Jansen. Approximation Algorithms for Time Constrained Scheduling. *Information and Computation*, 132:143–157, 1997.
- [KKP⁺05] Yoonjin Kim, Mary Kiemb, Chulsoo Park, Jinyong Jung, and Kiyong Choi. Resource Sharing and Pipelining in Coarse-Grained Reconfigurable Architecture for Domain-Specific Optimization. In *Design Automation and Test in Europe*, pages 12–17, Washington, DC, USA, 2005. IEEE Computer Society.
- [KM77] G. Kahn and DB MacQueen. Coroutines and Networks of Parallel Processes. In *International Federation for Information Processing Congress*, volume 77, pages 993–998. North-Holland Publishing Co., 1977.
- [KSG⁺07] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable Lightweight Processors. In *Conference on Computing Frontiers*, pages 381–394, Washington, DC, USA, 2007. IEEE Computer Society.
- [LAG⁺01] J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, and J. Eckhardt. Lifetime-Sensitive Modulo Scheduling in a Production Environment. *IEEE Transactions on Computers*, 50(3):234–249, 2001.
- [Lam88] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, New York, NY, USA, 1988. ACM.
- [LBF⁺98] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machines. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 32(5):46–57, 1998.
- [LCD03] Jong-eun Lee, Kiyong Choi, and N.D. Dutt. Compilation Approach for Coarse-Grained Reconfigurable Architectures. *IEEE Design & Test of Computers*, 20(1):26–33, 2003.
- [LE04] Song Li and C. Ebeling. QuickRoute: A Fast Routing Algorithm for Pipelined Architectures. In *IEEE International Conference on Field-Programmable Technology*, pages 73–80, Queensland, Australia, 2004.
- [LGAV96] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing Module Scheduling: A Lifetime-Sensitive Approach. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, 1996.
- [LM87a] E.A. Lee and D.G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.

- [LM87b] Edward Ashford Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.
- [LP95] EA Lee and TM Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.
- [LS05] H. Lange and H. Schroder. Evaluation Strategies for Coarse Grained Reconfigurable Architectures. In *International Workshop on Field-Programmable Logic and Applications*, pages 586–589, 2005.
- [MCC⁺06] M. Mishra, T.J. Callahan, T. Chelcea, G. Venkataramani, S.C. Goldstein, and M. Budiu. Tartan: Evaluating Spatial Computation for Whole Program Execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 163–174. ACM New York, NY, USA, 2006.
- [MD96] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 157–166, Apr 1996.
- [ME95] Larry McMurchie and Carl Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *ACM International Symposium on Field-Programmable Gate Arrays*, pages 111–117. ACM Press, 1995. Monterey, California, United States.
- [MG88] S. Mallela and L.K. Grover. Clustering Based Simulated Annealing for Standard Cell Placement. In *Design Automation Conference*, volume 25, pages 312–317, Jun 1988.
- [MG07] M. Mishra and S.C. Goldstein. Virtualization on the Tartan Reconfigurable Architecture. In *International Workshop on Field-Programmable Logic and Applications*, pages 323–330, Aug. 2007.
- [MGAk03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *ACM SIGGRAPH Conference on Computer Graphics and Interactive Techniques*, pages 896–907, New York, NY, USA, 2003. ACM.
- [MH01] Matthew C. Merten and Wen-mei W. Hwu. Modulo Schedule Buffers. In *Conference on Computing Frontiers*, pages 138–149, Washington, DC, USA, 2001. IEEE Computer Society.

- [MLC⁺92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Conference on Computing Frontiers*, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
- [MLV⁺05] B. Mei, A. Lambrechts, D. Verkest, J. Mignolet, and R. Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design & Test of Computers*, 22(2):90–101, 2005.
- [Moh06] Nitin Mohan. *Low-Power High-Performance Ternary Content Addressable Memory Circuits*. PhD thesis, Univeristy of Waterloo, 2006.
- [MS04] N. Mohan and M. Sachdev. Low Power Dual Matchline Ternary Content Addressable Memory. In *IEEE International Symposium on Circuits and Systems*, volume 2, pages II–633–6 Vol.2, May 2004.
- [MSK⁺99] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A Reconfigurable Arithmetic Array for Multimedia Applications. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 135–143, New York, NY, USA, 1999. ACM.
- [MVV⁺02] Bingfeng 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*, pages 166–173, 2002.
- [MVV⁺03a] 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 Workshop on Field-Programmable Logic and Applications*. Springer, 2003.
- [MVV⁺03b] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling. In *Design Automation and Test in Europe*, pages 255–61, 2003.
- [NCV⁺03] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *International Parallel and Distributed Processing Symposium*, pages 7 pp.–, 2003.
- [NE98] E. Nystrom and A.E. Eichenberger. Effective Cluster Assignment for Modulo Scheduling. In *Conference on Computing Frontiers*, pages 103–114, Nov-2 Dec 1998.

- [OBM90] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-driven Interpretation of Imperative Languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 257–271, New York, NY, USA, 1990. ACM.
- [OEPM09] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. Recurrence Cycle Aware Modulo Scheduling for Coarse-Grained Reconfigurable Architectures. *SIGPLAN Notices*, 44:21–30, June 2009.
- [Par95] T.M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, 1995.
- [PB07] E.M. Panainte and U.P. Bucuresti. *The Molen Compiler for Reconfigurable Architectures*. PhD thesis, TU Delft, 2007.
- [PFM⁺08] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong seok Kim. Edge-Centric Modulo Scheduling for Coarse-Grained Reconfigurable Architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 166–176, New York, NY, USA, 2008. ACM.
- [PPL95] T.M. Parks, J.L. Pino, and E.A. Lee. A Comparison of Synchronous and Cyclo-Static Dataflow. In *Asilomar Conference on Signals, Systems and Computers*, volume 29, page 204, 1995.
- [Rau94] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Conference on Computing Frontiers*, pages 63–74, New York, NY, USA, 1994. ACM.
- [RG81] BR Rau and CD Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *Conference on Computing Frontiers*, pages 183–198. ACM New York, NY, USA, 1981.
- [RKR95] H.R. Rabiee, R.L. Kashyap, and H. Radha. Multiresolution Image Compression with BSP Trees and Multilevel BTC. In *International Conference on Image Processing*, volume 3, pages 600–603 vol.3, October 1995.
- [RLTS92] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register Allocation for Software Pipelined Loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–299, New York, NY, USA, 1992. ACM.

- [RVL96] H. Radha, M. Vetterli, and R. Leonardi. Image Compression Using Binary Space Partitioning Trees. *IEEE Transactions on Image Processing*, 5(12):1610–1624, 1996.
- [SEH03] Akshay Sharma, Carl Ebeling, and Scott Hauck. PipeRoute: A Pipelining-Aware Router for FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 68–77, New York, NY, USA, 2003. ACM.
- [SEH06] A. Sharma, C. Ebeling, and S. Hauck. PipeRoute: A Pipelining-Aware Router for Reconfigurable Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):518 – 532, march 2006.
- [SHA00] John W. Sias, Wen-Mei W. Hwu, and David I. August. Accurate and Efficient Predicate Analysis with Binary Decision Diagrams. In *Conference on Computing Frontiers*, pages 112–123, New York, NY, USA, 2000. ACM.
- [SHE05] A. Sharma, S. Hauck, and C. Ebeling. Architecture-Adaptive Routability-Driven Placement for FPGAs. In *International Workshop on Field-Programmable Logic and Applications*, pages 427–432, 2005.
- [SL96] Mark G. Stoodley and Corinna G. Lee. Software Pipelining Loops with Conditional Branches. In *Conference on Computing Frontiers*, pages 262–273, Washington, DC, USA, 1996. IEEE Computer Society.
- [SLL⁺00] H. Singh, M.H. Lee, G. Lu, FJ Kurdahi, N. Bagherzadeh, and EM Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.
- [SMD04] M. Smelyanskiy, S. Mahlke, and E.S. Davidson. Probabilistic Predicate-Aware Modulo Scheduling. In *International Symposium on Code Generation and Optimization*, pages 151–162, 2004.
- [SMDL03] Mikhail Smelyanskiy, Scott A. Mahlke, Edward S. Davidson, and Hsien-Hsin S. Lee. Predicate-Aware Scheduling: A Technique for Reducing Resource Constraints. In *International Symposium on Code Generation and Optimization*, pages 169–178, Washington, DC, USA, 2003. IEEE Computer Society.
- [SMSO03] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Conference on Computing Frontiers*, pages 291–302, Dec. 2003.
- [Sny82] L. Snyder. Introduction to the Configurable, Highly Parallel Computer. *IEEE Computer*, 15(1):47–64, 1982.

- [SPMS01] A. Singh, G. Parthasarathy, and M. Marek-Sadowska. Interconnect Resource-Aware Placement for Hierarchical FPGAs. In *IEEE/ACM International Conference on Computer Aided Design*, pages 132–136, 2001.
- [SS] Sun Microsystems and Static Free Software. Electric VLSI Design System. <http://www.staticfreesoft.com/>, Version 8.08a.
- [Sta00] Judith Alyce Stafford. *A Formal, Language-Independent, and Compositional Approach to Interprocedural Control Dependence Analysis*. PhD thesis, University of Colorado at Boulder, Boulder, CO, USA, 2000. Director-Wolf, Alexander L.
- [Sut89] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.
- [SV88] Baruch Schieber and Uzi Vishkin. On Finding Lowest Common Ancestors: Simplification and Parallelization. In John Reif, editor, *VLSI Algorithms and Architectures*, volume 319 of *Lecture Notes in Computer Science*, pages 111–123. Springer Berlin / Heidelberg, 1988.
- [TLAA05] M.D. Taylor, W. Lee, S.P. Amarasinghe, and A. Agarwal. Scalar Operand Networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, 2005.
- [TLM⁺04] MB Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, et al. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *IEEE International Symposium on Computer Architecture*, pages 2–13, 2004.
- [TLS90] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of Loops with Exits on Pipelined Architectures. In *ACM/IEEE Conference on Supercomputing*, pages 200–212, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.
- [TMJ⁺99] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek, and André DeHon. HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 125–134. ACM Press, 1999.
- [TSL00] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, 290(5500):2319–2323, 2000.

- [TZB99] Jürgen Teich, Eckart Zitzler, and Shuvra S. Bhattacharyya. 3D Exploration of Software Schedules for DSP Algorithms. In *International Workshop on Hardware/Software Codesign*, pages 168–172, 1999.
- [VE10] Brian Van Essen. *Improving the Energy Efficiency of Coarse-Grained Reconfigurable Arrays*. PhD thesis, University of Washington, 2010.
- [VEPW⁺10] Brian Van Essen, Robin Panda, Aaron Wood, Carl Ebeling, and Scott Hauck. Managing Short-Lived and Long-Lived Values in Coarse-Grained Reconfigurable Arrays. In *International Workshop on Field-Programmable Logic and Applications*, pages 380–387, Los Alamitos, CA, USA, 2010. IEEE Computer Society.
- [VEPW⁺11] Brian C. Van Essen, Robin Panda, Aaron Wood, Carl Ebeling, and Scott Hauck. Energy-Efficient Specialization of Functional Units in a Coarse-Grained Reconfigurable Arrays. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 107–110, New York, NY, USA, 2011. ACM, ACM.
- [VEWC⁺09] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling, and S. Hauck. Static versus Scheduled Interconnect in Coarse-Grained Reconfigurable Arrays. In *International Workshop on Field-Programmable Logic and Applications*, pages 268–275. IEEE, 2009.
- [VSG⁺10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 38, pages 205–218, New York, NY, USA, March 2010. ACM.
- [WBS07] Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs. In *Design Automation Conference*, pages 658–663, New York, NY, USA, 2007. ACM.
- [WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value Dependence Graphs: Representation without Taxation. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–310, New York, NY, USA, 1994. ACM.
- [WELP96] P. Wauters, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-Dynamic Dataflow. In *Euromicro Workshop on Parallel and Distributed Processing (PDP)*, pages 319–326, Jan 1996.

- [WHSB92] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. Enhanced Modulo Scheduling for Loops with Conditional Branches. In *Conference on Computing Frontiers*, pages 170–179, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
- [Wil97] S.J.E. Wilton. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*. PhD thesis, University of Toronto, 1997.
- [WKMB07] K. Wu, A. Kanstein, J. Madsen, and M. Berekovic. MT-ADRES: Multithreading on Coarse-Grained Reconfigurable Architecture. *Lecture Notes in Computer Science*, 4419:26, 2007.
- [WKMV04] S.J.E. Wilton, N. Kafafi, B. Mei, and S. Vernalde. Interconnect Architectures for Modulo-Scheduled Coarse-Grained Reconfigurable Arrays. In *IEEE International Conference on Field-Programmable Technology*, pages 33–40, 2004.
- [WPP95] Nancy J. Warter-Perez and Noubar Partamian. Modulo Scheduling with Multiple Initiation Intervals. In *Conference on Computing Frontiers*, pages 111–119, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.
- [WSZS07] Kilian Weinberger, Fei Sha, Qihui Zhu, and Lawrence Saul. Graph Laplacian Methods for Large-Scale Semidefinite Programming, with an Application to Sensor Localization. In B. Schölkopf, J. Platt, and Thomas Hofmann, editors, *Advances in Neural Information Processing Systems*. MIT Press, Cambridge, MA, 2007.
- [WTS⁺97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, 1997.
- [XSAH10] J. Xu, N. Subramanian, A. Alessio, and S. Hauck. Impulse C vs. VHDL for Accelerating Tomographic Reconstruction. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 171–174, May 2010.
- [Ylv10] Benjamin Ylvisaker. *“C-Level” Programming of Parallel Coprocessor Accelerators*. PhD thesis, University of Washington, 2010.
- [ZLCP04] Ce Zhu, Xiao Lin, L. Chau, and Lai-Man Po. Enhanced Hexagonal Search for Fast Block Motion Estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(10):1210–1214, oct. 2004.
- [ZMLM08] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering Hidden Loop Level Parallelism in Sequential Applications. In *International Symposium on High-Performance Computer Architecture*, 2008.