

NORTHWESTERN UNIVERSITY

Architecture Generation of Customized Reconfigurable Hardware

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Electrical & Computer Engineering

By

Katherine Leigh Compton

EVANSTON, ILLINOIS

December 2003

© Copyright Katherine Leigh Compton 2003
All Rights Reserved

ABSTRACT

Architecture Generation of Customized Reconfigurable Hardware

Katherine Leigh Compton

Reconfigurable hardware is ideal for use in systems-on-a-chip (SoCs), achieving hardware speeds but also flexibility not available with more traditional custom circuitry. Traditional FPGA structures can be used in an SoC, but they suffer from significant overhead due to their generic nature. Alternatively, for cases when the application domain of the SoC is known, the reconfigurable hardware can be optimized for that domain. The Totem Project focuses on the automatic creation of customized reconfigurable architectures, including high-level design, VLSI layout, and associated custom place and route tools.

This thesis focuses on the high-level design phase, or “Architecture Generation”. Two distinct categories of reconfigurable architectures can be created: highly optimized near-ASIC designs with a very low degree of reconfigurability, and flexible architectures with a one-dimensional segmented routing structure. Each of these design methods shows significant improvements through tailoring the architectures to the given application area. The cASIC designs are on average up to 12.3x smaller than an FPGA solution with embedded multipliers and 2.2x smaller than a standard cell implementation. The more flexible architectures, able to support a wider variety of circuits, are on average up to 5.5x smaller than the FPGA solution, and close in area to standard cells.

Acknowledgments

There are a large number of people that have contributed to this dissertation, either in terms of content or support during its creation. First I want to mention my friend and advisor, Scott Hauck. He taught me how to be a successful PhD student, and provided nearly gentle shoves in the right direction when needed.

There were a number of graduate students at the University of Washington that contributed to this work. Akshay Sharma provided the Totem place and route tool. Kim Motonaga and Shawn Phillips provided area numbers for the manual layouts of the logic components I use as well as the areas of the standard cell implementations of the netlists. Ken Eguro and Todd Owen gathered the FPGA data used in my comparisons. Ken was also critical during a late-night struggle to ensure the FPGA area measurements were as fair to the FPGA as possible. I would like to thank Carl Ebeling, Chris Fisher, Larry McMurchie, Darren Cronquist, Mike Scott and others for their work on RaPiD, which provided both a starting point for my work and the RaPiD-C compiler for creating application netlists. Also, several people from the University of Washington were instrumental in writing the RaPiD-C applications that later became the compiled netlists.

On a different note, I am grateful for the funding that supported my graduate work. What could be better than being paid to learn? My funding sources include an NSF Fellowship, a Motorola UPR grant, and a Cabell Dissertation-Year Fellowship.

I would also like to thank Prith Banerjee for providing me with a desk to work at after my research group moved from Northwestern to the University of Washington, and the ECE staff who helped me with administrative functions I suddenly had to perform on my own.

A large number of graduate students, faculty, and other FPGA researchers provided mental and emotional support as well as guidance for both the dissertation writing and job search process. These include, but are in no way limited to: Mark Chang, Scott Hauck, Pramod Joisha, Miriam Leeser, Guy Lemieux, Janak Parekh, Satnam Singh, Russ Tessier, Keith Underwood, and Steve Wilton. I am also grateful for the weekly diversions provided by my friends Bohuš Blahut and Anne Willmore. Finally, I would like to thank my family and husband for making my PhD possible, and for being there when I needed them.

Dedication

I dedicate this dissertation to my advisor, Scott Hauck, to my family (all of them), and most importantly to my loving husband, Jason Compton.

Contents

List of Figures	x
List of Tables	xiv
<i>Chapter 1</i> Introduction.....	1
<i>Chapter 2</i> Reconfigurable Computing.....	7
2.1 Technology	12
2.1.1 Configurable Hardware.....	13
2.1.2 Traditional FPGAs.....	15
2.2 Hardware.....	19
2.2.1 Microprocessor Coupling.....	22
2.2.2 Logic Block Granularity	25
2.2.3 Heterogeneous Arrays.....	29
2.2.4 Routing Resources	31
2.2.5 One-Dimensional Structures.....	34
2.2.6 Hardware Summary	36
2.3 Software	37
2.4 Run-Time Reconfiguration	43
2.4.1 Fast Configuration	45
2.5 Reconfigurable Computing Summary	50
<i>Chapter 3</i> Reconfigurable Hardware in SoCs.....	53
3.1 Reconfigurable Subsystems.....	54
3.2 Systems-on-a-Programmable-Chip (SoPCs)	56
<i>Chapter 4</i> Research Framework	59
4.1 RaPiD.....	60
4.1.1 Datapath Architecture.....	61
4.1.2 Control Architecture	62
4.1.3 RaPiD-C Compiler.....	63
4.2 Totem Project.....	64
4.2.1 High-Level Architecture Design.....	66
4.2.2 Physical Layout.....	67
4.2.3 Place and Route.....	69
4.3 Testing Framework	70
4.3.1 Standard Cells	71
4.3.2 FPGA	72

4.3.3	RaPiD.....	74
4.3.4	Relative Areas.....	76
<i>Chapter 5</i>	Logic Generation.....	79
5.1	Type and Quantity of Units.....	79
5.2	Binding vs. Physical Moves.....	80
5.3	Adapting Simulated Annealing.....	83
<i>Chapter 6</i>	Configurable ASICs.....	89
6.1	Logic Generation.....	90
6.2	Routing Generation.....	91
6.2.1	Greedy.....	94
6.2.2	Bipartite.....	95
6.2.3	Clique.....	99
6.3	Results.....	102
6.4	Summary.....	109
<i>Chapter 7</i>	Flexible Architectures.....	111
7.1	Flexible Architectural Style.....	112
7.2	Logic Generation.....	113
7.3	Routing Generation.....	114
7.3.1	Shared Concepts.....	115
7.3.2	Greedy Histogram.....	122
7.3.3	Regular Architectures.....	127
7.4	Results.....	133
7.4.1	Area.....	133
7.4.2	Flexibility.....	138
7.5	Summary.....	140
<i>Chapter 8</i>	Track Placement.....	142
8.1	Problem Description.....	145
8.2	Track Placement Algorithms.....	150
8.2.1	Brute Force Algorithm.....	151
8.2.2	Simple Spread Algorithm.....	152
8.2.3	Power2 Algorithm.....	153
8.2.4	Optimal Factor Algorithm.....	155
8.2.5	Relaxed Factor Algorithm.....	161
8.3	Algorithm Comparison.....	169
8.4	Summary.....	173
<i>Chapter 9</i>	Flexibility Testing.....	175
9.1	Circuit Generator.....	176

9.1.1	Circuit Profiling	177
9.1.2	Domain Profiling	178
9.1.3	Circuit Creation.....	179
9.2	Synthetic Circuit Validation	184
9.3	Testing Flexibility	186
9.3.1	Single Circuit Flexibility.....	186
9.3.2	Domain Flexibility	188
9.4	Other Uses.....	194
9.5	Summary	195
<i>Chapter 10</i>	Conclusions.....	197
10.1	Contributions.....	198
10.2	Future Work.....	200
References	202

List of Figures

Figure 2.1: Compute-intensive sections of application code are mapped onto the reconfigurable hardware.	9
Figure 2.2: (a) A programming bit for SRAM-based FPGAs [Xilinx94, Hauck98a] and (b) a programmable routing connection.....	13
Figure 2.3: (a) A D flip-flop with optional bypass, and (b) a 3-input LUT [Hauck98a]...14	
Figure 2.4: A basic logic block, with a 4-input LUT, carry chain, and a D-type flip-flop with bypass.....	17
Figure 2.5: A generic island-style FPGA routing architecture.	18
Figure 2.6: Different levels of coupling in a reconfigurable system [Hauck98a]. Reconfigurable logic is shaded.	23
Figure 2.7: The functional unit from a Xilinx 6200 cell [Xilinx96].....	26
Figure 2.8: One cell in the RaPiD-I reconfigurable architecture [Ebeling96].....	28
Figure 2.9: (a) Segmented and (b) hierarchical routing structures.	33
Figure 2.10: (a) A traditional two-dimensional island-style routing structure, and (b) a one-dimensional routing structure..	35
Figure 2.11: Three possible design flows for algorithm implementation on a reconfigurable system.	38
Figure 2.12: Applications which are too large to entirely fit on the reconfigurable hardware can be partitioned into two or more smaller configurations that can occupy the hardware at different times.	43
Figure 4.1: A single cell from the RaPiD architecture [Cronquist99a, Scott01].....	61
Figure 4.2: The three major components of the Totem Project	65
Figure 5.1: (a) <i>Binding</i> assigns instances of a netlist to physical components. (b) <i>Physical moves</i> reposition the physical components themselves.	81
Figure 5.2: Two different example netlists that could be used in architecture generation.	82

Figure 5.3: The initial physical placement and bindings of an architecture created for the netlists of Figure 5.2.	84
Figure 5.4: A physical move performed during the placement operation.	85
Figure 5.5: A rebinding performed during the placement operation.	86
Figure 5.6: The final placement of the architecture created for the netlists from Figure 5.2 after a series of moves such as those illustrated in Figure 5.4 and Figure 5.5.....	86
Figure 6.1: cASIC routing architecture created for the example from Chapter 5.....	92
Figure 6.2: Pseudocode for the Greedy cASIC generation technique.	95
Figure 6.3: An example graph which does not produce the optimal solution when bipartite matching is used recursively.....	96
Figure 6.4: Pseudocode for the recursive maximum weight bipartite matching cASIC technique.....	97
Figure 6.5: Pseudocode of the maximum weight bipartite matching graph algorithm [Shier99] used by the Bipartite cASIC generation algorithm from Figure 6.4.....	98
Figure 6.6: An improved solution to the graph of Figure 6.3 found using clique partitioning.....	100
Figure 6.7: The pseudocode of the Clique cASIC generation algorithm.....	101
Figure 6.8: Pseudocode of the clique partitioning graph algorithm [Dorndorf94] used by the Clique cASIC generation algorithm from Figure 6.7.	102
Figure 6.9: Comparative area results of the different cASIC routing generation algorithms, normalized to the Clique Overlap result for each application.	104
Figure 7.1: Flexible routing architecture created for the example from Chapter 5.	115
Figure 7.2: Examples of the different types of routing tracks, (a) local tracks, and (b) distance tracks with bus connectors (represented by the squares on the tracks).	116
Figure 7.3: An extreme example of a non-distributed routing architecture.....	118
Figure 7.4: A distributed routing architecture.....	118
Figure 7.5: Calculating the unroutable cross-section for the placement of Figure 5.6.	119

Figure 7.6: An example situation where an unmodified left-edge routing algorithm leads a routing generation algorithm to construct a more expensive solution.	120
Figure 7.7: Pseudocode summary of the fast greedy router used within the flexible routing generation algorithms presented in this chapter.	122
Figure 7.8: Pseudocode for the main body of the Greedy Histogram generation algorithm.	123
Figure 7.9: Sub-functions for the Greedy Histogram Algorithm from Figure 7.8.	125
Figure 7.10: Pseudocode for the main body of the Add Max Once flexible routing generation algorithm.	128
Figure 7.11: The pseudocode for a subfunction used by both the Add Max Once algorithm in Figure 7.10 and the Add Min Loop algorithm in Figure 7.13.	129
Figure 7.12: An example of AMO creating a more costly solution than necessary.	130
Figure 7.13: The pseudocode for the Add Min Loop algorithm.	132
Figure 7.14: Comparative area results of the different flexible routing generation algorithms, Greedy Histogram (GH), Add Max Once (AMO), and Add Min Loop (AML).	136
Figure 8.1: Two different track placements for the same type of tracks, (a) a very poor one and (b) a very good one.	143
Figure 8.2: Two examples of reconfigurable architectures with segmented channels. ...	145
Figure 8.3: An alternate placement for the architectures in Figure 8.2 that maintains perfect smoothness of breaks, but is intuitively less routable than the placement in Figure 8.2b.	146
Figure 8.4: Diversity score for two different track placements for the same track placement problem: (a) a poor placement, and (b) a good placement.	147
Figure 8.5: A track placement problem solved using (a) Simple Spread solution and a (b) Brute Force solution.	153
Figure 8.6: An example of the operation of Power2 at each of three S values for a case with one length-2 track, one length-4 track, and three length-8 tracks.	155

Figure 8.7: The breaks from tracks of length S_{\max} are emulated by the breaks of placeholder tracks of length S_{next} for the next iteration.....	159
Figure 8.8: The pseudocode for the Optimal Factor algorithm.	160
Figure 8.9: The Relaxed Algorithm code that replaces the main while loop in the Optimal Algorithm.....	162
Figure 8.10: A track arrangement (top), corresponding topography (middle), and the ideal topography for these tracks (bottom).....	163
Figure 8.11: The relaxed placement function.	164
Figure 8.12: The density based placement function and the function to calculate the number of tracks to add to a given region in the newest plain.	165
Figure 8.13: An example of a <i>breaks</i> [] array for $K = 24$, and the corresponding <i>tracks</i> [] array for $S_{\text{next}} = 6$	168
Figure 8.14: This function does not create actual placeholder tracks to represent tracks with $S > S_{\text{next}}$, but it does fill the <i>tracks</i> [] array in such a way as to simulate all previously placed tracks being converted to segment length S_{next}	168
Figure 8.15: A comparison of Relaxed and Simple Spread to the Brute Force method, with respect to numT (left), numS (center), and maxTS (right).	170
Figure 8.16: Relaxed Factor, Power2, and Simple Spread comparison for cases with only power-of-two S values.....	171
Figure 8.17: The number of tracks in our target architecture required to successfully place and route all netlists in an application using the given track placement algorithm.	173
Figure 9.1: A directed graph of a sample RaPiD netlist to be profiled.....	178
Figure 9.2: Steps in the creation of an example synthetic circuit graph.	180

List of Tables

Table 4.1: Eight applications used to test Totem architectures.	71
Table 4.2: The areas of the eight different applications from Table 4.1 implemented using standard cells.....	72
Table 4.3: The FPGA areas of the eight different applications from Table 4.1.	74
Table 4.4: The RaPiD areas of the eight different applications from Table 4.1.	76
Table 4.5: The areas of all of the netlists from Table 4.1 using each of the implementation methods, normalized to the standard cell area.	78
Table 4.6: The areas of each of the applications from Table 4.1 using each of the implementation methods, normalized to the standard cell area.....	78
Table 5.1: The calculation of the new temperature T_{new} based on the percentage of moves accepted, R_{accept}	88
Table 6.1: The areas of the routing structures created by the Bipartite cASIC generation methods using both the ports and the overlap methods.	103
Table 6.2: The areas, in mm^2 , of the eight different applications from Table 4.1, as implemented with the three cASIC algorithms.....	106
Table 6.3: Area improvements calculated over the reference architectures, then averaged across all applications.	107
Table 7.1: A table of the number of routing tracks created for each application by each routing generation algorithm.....	134
Table 7.2: The areas, in mm^2 , of the eight different applications from Table 4.1, as implemented with the three flexible routing architecture generation algorithms. ...	137
Table 7.3: A summary of area comparisons between the different implementation techniques.	137
Table 7.4: Initial flexibility study of the generated architectures.	139

Table 9.1: A comparison of characteristics of generated synthetic circuits to those of the original circuits.	185
Table 9.2: A table listing the % likelihood that the original parent netlist can be placed and routed onto an architecture created from a synthetic circuit based on the parent characteristics.....	187
Table 9.3: A table indicating the percentage of architectures having enough logic to implement the given original netlists.....	191
Table 9.4: Success rates, in percentages, of routing original netlists onto architectures created by each of the three flexible routing generation algorithms from a set of synthetic benchmarks created from a domain profile.	192

Chapter 1

Introduction

As chip fabrication techniques continue to advance and become more refined, the concept of "System-on-a-Chip" (SoC) will further evolve and grow in popularity. With system components moved from on-board to on-chip, communication times and bandwidth are greatly improved, raising the question of exactly what type of hardware to include on SoCs. Reconfigurable hardware [Compton02a] shows great potential for SoC use, providing hardware speeds, while maintaining a level of flexibility not available with traditional custom circuitry. This flexibility is the key to allowing both hardware reuse and post-fabrication modification.

The core of a reconfigurable architecture is a set of hardware resources, including logic and routing, whose function is controlled by on-chip configuration SRAM. Programming the SRAM, either at the start of an application or during execution, allows the hardware functionality to be configured and reconfigured, permitting reconfigurable systems to implement different algorithms and applications on the same hardware.

This reusability makes reconfigurable hardware a prime candidate as a subsystem for SoCs. Rather than using separate custom circuits to accelerate each potential

application, a single reconfigurable architecture can be used. This reconfigurable logic can implement circuits from each application in hardware as needed.

Field-programmable gate arrays (FPGAs) [Brown92a, Rose93] are a widely available form of reconfigurable hardware. One major difficulty of using FPGAs for DSP, networking, and other applications is their generic design. FPGAs attempt to fulfill the computation requirements of any application that might be needed. However, because different application types have different requirements, a large amount of hardware (and silicon area) is wasted if the applications are actually constrained to a limited range of computations. While the flexibility of general-purpose FPGAs has its place in situations where computational requirements are not known in advance, specialized on-chip hardware is commonly used to obtain greater performance for a specific set of compute-intensive calculations.

Reconfigurable architectures can be made more efficient if the algorithm types are known in advance. In this case, the amount of "useless" hardware and programming points that would otherwise occupy valuable area or slow the computations can be reduced or removed. Architectures such as RaPiD [Ebeling96], PipeRench [Goldstein00], and Pleiades [Abnous96] target multimedia and DSP domains by using coarser-grained units (such as 16-bit ALUs and multipliers in the case of RaPiD), and more restricted routing structures to implement the targeted applications more efficiently.

Even a fixed reconfigurable architecture containing coarse-grained units can suffer overheads when the logic and routing resources deviate significantly from the

needs of the circuits implemented using this hardware. To address this issue, the RaPiD group has proposed the synthesis of custom RaPiD arrays for different application sets [Cronquist99b]. While specialized reconfigurable architectures are theoretically beneficial, they would be impractical in practice if they needed to be manually designed for each application group. Each of these optimized reconfigurable structures can be quite different, depending on the application requirements. Unfortunately, this contradicts a fundamental principle of FPGAs and reconfigurable hardware: quick time-to-market with low design costs.

Therefore, an automatic solution that allows designers to create reconfigurable structures for a given range of computations should be considered. These application domains could include cryptography, DSP or a sub-domain of DSP, specific scientific data analysis, or any other compute-intensive area. This concept is different from traditional ASICs in that some level of hardware programmability is retained. This programmability gives the custom architecture a measure of flexibility beyond what is available in an ASIC, and provides the benefits of run-time reconfigurability. Run-time reconfiguration can then be employed to allow for near ASIC-level performance with a much smaller area overhead due to the re-use of area-intensive hardware components. The resulting automatically-generated reconfigurable hardware will then be embedded into an SoC or ASIC design.

The Totem Project [Compton01, Compton02d, Phillips02, Sharma02, Compton03, Sharma03] is an attempt to automatically generate custom reconfigurable

architectures based on an input set of applications. The project goal is to provide a fully automatic design path, greatly decreasing the cost of new architecture development. This includes the high-level architecture design, the transistor level layout of those architectures, and place and route tools supporting the customized architectures.

The work presented here focuses on the high-level architecture design, also known as architecture generation. Depending on the algorithms and the stated parameters, this architecture generation could provide a design anywhere within the range between ASICs and FPGAs. Very constrained computations would be primarily fixed ASIC logic, while more unconstrained domains would require near-FPGA functionality. The issues involved in these two types of specialized architecture generation will be discussed, and algorithms will be presented that demonstrate significant area savings over less-specialized designs. It should be stressed that these architectures are custom reconfigurable logic and are intended to be implemented directly into silicon, not an FPGA structure. The very constrained ASIC-like architectures, discussed in Chapter 6, are on average up to 12.3x smaller than an FPGA implementation and 2.2x smaller than a standard cell layout. The more flexible architectures, discussed in Chapter 7, support a wider variety of circuits and are on average up to 5.5x smaller than FPGA implementations.

This dissertation is organized as follows:

- *Chapter 2: Reconfigurable Computing* provides technical background, discussing the structure and operation of FPGAs and other types of reconfigurable hardware.
- *Chapter 3: Reconfigurable Hardware in SoCs* describes current systems employing reconfigurable logic in Systems-on-a-Chip.
- *Chapter 4: Research Framework* provides architectural details of RaPiD [Ebeling96, Cronquist99a], the structural basis for the work presented here, and provides a description of the overall Totem tool flow.
- *Chapter 5: Logic Generation* describes the method used to create the logic portion of the reconfigurable architectures, which is common to both architecture generation methods presented in Chapter 6 and Chapter 7. It also seeks to define key terminology required for discussing reconfigurable architecture generation using a set of netlists as the specification.
- *Chapter 6: Configurable ASIC* presents two different algorithms, Greedy and Clique Partitioning, for the generation of very ASIC-like customized reconfigurable architectures.
- *Chapter 7: Flexible Architectures* discusses three algorithms, Greedy Histogram, Add Max Once, and Add Min Loop, used to create more flexible specialized architectures in the RaPiD design style.

- *Chapter 8: Track Placement* focuses on one of the design issues from Chapter 7, the arrangement of a pre-determined quantity of routing resources within a single channel. The track placement problem is defined and a metric is presented to measure track placement quality. An optimal algorithm to perform track placement is given, along with a near-optimal relaxed version.
- *Chapter 9: Flexibility Testing* discusses methods that can be used to test the flexibility of generated reconfigurable architectures. The flexibility of the architecture generation algorithms from Chapter 7 is then analyzed using these techniques.
- Chapter 10: Conclusions summarizes the contributions of this work, and lists a number of areas of future effort.

Chapter 2

Reconfigurable Computing

Two primary methods exist in conventional computing for the execution of algorithms. One method utilizes hardwired technology, either an Application Specific Integrated Circuit (ASIC), or a group of individual components forming a board-level solution, to perform the operations in hardware. ASICs are designed to perform a specific computation quickly and efficiently, but cannot be altered after fabrication. Modification of the circuit requires redesign and re-fabrication of the chip. This is an expensive process, especially when replacing ASICs in a large number of deployed systems. Board-level circuits are also somewhat inflexible, often requiring a board redesign and replacement in the event of changes to the application.

The second method provides a far more flexible solution. Software-programmed microprocessors execute a set of instructions to perform a computation. System functionality can be altered without hardware changes simply by reprogramming the software. However, the price of this flexibility is performance, which is far below that of an ASIC. Also, microprocessors consume more power than an ASIC. The processor must read each instruction from memory, decode its meaning, and only then execute it.

This leads to a high execution overhead for each individual operation. Additionally, the possible instructions that may be used by a program are determined at the processor design time. Any other operations that are to be implemented must be built out of one or more existing instructions.

Reconfigurable computing fills the gap between hardware and software, achieving greater performance than software, while maintaining a higher level of flexibility than hardware. Reconfigurable devices, including field-programmable gate arrays (FPGAs), contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements, also known as logic blocks, are connected using a set of programmable routing resources. Custom digital circuits are mapped to reconfigurable hardware by computing the logic functions of the circuit within the logic blocks, and using the configurable routing to connect the blocks together to form the necessary circuit.

FPGAs and reconfigurable computing have been shown to accelerate a variety of applications. Data encryption can leverage both parallelism and fine-grained data manipulation. An implementation of the Serpent Block Cipher in the Xilinx Virtex XCV1000 shows a throughput increase by a factor of over 18 compared to a Pentium Pro PC running at 200MHz [Elbirt00]. Additionally, a reconfigurable computing implementation of sieving for factoring large numbers (useful in breaking encryption schemes) was accelerated by a factor of 28 over a 200 MHz UltraSparc workstation [Kim00a]. The Garp architecture shows a comparable speed-up for DES [Hauser97], as

does an FPGA implementation of an elliptic curve cryptography application [Leung00]. PNN classification has been accelerated by a factor of 63 using reconfigurable hardware [Chang99], and the SPIHT wavelet-based image compression algorithm has been accelerated by a factor of 457 [Fry02].

Other recent applications shown to exhibit significant speedups using reconfigurable hardware include: automatic target recognition [Rencher97]; string pattern matching [Weinhardt99]; Golomb Ruler Derivation [Dollas98, Sotiriades00]; transitive closure of dynamic graphs [Huelsbergen00]; Boolean satisfiability [Zhong98]; data compression [Huang00]; and genetic algorithms for the traveling salesman problem [Graham96].

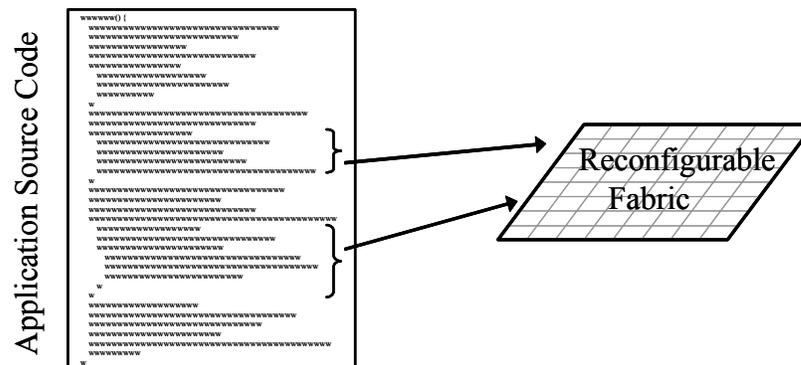


Figure 2.1: Compute-intensive sections of application code are mapped onto the reconfigurable hardware.

In order to achieve these performance benefits while supporting a wide range of applications, reconfigurable systems usually combine reconfigurable logic with a general-purpose microprocessor. The processor performs operations that cannot be done efficiently in the reconfigurable logic, such as data-dependent control and some memory

accesses, while computational cores are mapped to the reconfigurable hardware, as in Figure 2.1. This reconfigurable logic consists of either commercial FPGAs or custom configurable hardware.

Compilation environments for reconfigurable hardware range from tools that assist programmers in hand mapping of a circuit to hardware, to complete automated systems that take a circuit description in a high-level language and translate it into a configuration for a reconfigurable system. The first step in the design process is partitioning a program into the sections that will be implemented in hardware and the sections executed in software on the host processor. Computations destined for reconfigurable hardware are synthesized into a gate level or register transfer level circuit description. This circuit is mapped onto the logic blocks within the reconfigurable hardware during the technology mapping phase. These mapped blocks are then placed into the specific physical blocks within the hardware, and the pieces of the circuit are connected using the reconfigurable routing. After compilation, the circuit is ready to be implemented by the reconfigurable hardware at run-time. These steps, when performed using an automatic compilation system, require little effort by the programmer to utilize the reconfigurable hardware. However, performing some or all of these operations manually frequently results in a more highly optimized circuit for performance-critical applications.

Since FPGAs must pay an area penalty because of their reconfigurability, device capacity is a concern. Assuming that the hardware can only be programmed at power-up,

a very large programmable device might be required to implement all of the functions in a program that can benefit from hardware-acceleration. Alternately, if a smaller device is used not all the functions may fit within the device, leaving some of the acceleration potential untapped.

Additional areas of the program may be accelerated by reusing the reconfigurable hardware during program execution, a process known as run-time reconfiguration (RTR). While this computing style allows for the acceleration of a greater portion of an application, it also limits the potential acceleration by introducing configuration overhead. Because configuration can take milliseconds or longer, rapid and efficient configuration is a critical issue. Configuration compression and configuration caching are examples of methods that can be used to reduce this overhead [Li02].

This chapter provides a brief overview of the hardware and software issues of reconfigurable computing. First is a discussion of the technology required for reconfigurable computing, followed by an examination of the various hardware structures used in reconfigurable systems. Next is a brief look at the software required to implement algorithms on reconfigurable systems. Finally, run-time reconfigurable systems are discussed, which further utilize the intrinsic flexibility of configurable computing platforms by optimizing the hardware not only for different applications, but also for different operations within a single application.

This chapter does not cover every technique and research project in the area of reconfigurable computing. For a comprehensive overview of the field, there are a

number of survey articles on the topic, covering more recent work [Compton02a], or older techniques and systems [Rose93, Hauck96, Vuillemin96, Mangione-Smith97, Hauck98a].

2.1 Technology

Some of the concepts behind reconfigurable computing have existed for some time [Estrin63]. Even general-purpose processors use some of the same basic ideas, such as reusing computational components for independent computations, and using multiplexers to control the routing between these components. However, the term reconfigurable computing, as it is used in current research, refers to systems incorporating some form of hardware programmability—customizing hardware operation using a number of physical control points. These control points can be changed at different points in time, allowing the same hardware to execute different applications. Recent advances in reconfigurable computing are primarily derived from the technologies developed for FPGAs in the mid-1980s. FPGAs were originally created to serve as a hybrid device between PALs and Mask-Programmable Gate Arrays (MPGAs). Like PALs, FPGAs are fully electrically programmable; the physical design costs are amortized over multiple application circuit implementations, and the hardware customizations can occur almost instantaneously. Like MPGAs, FPGAs can implement very complex computations on a single chip, with current devices containing the equivalent of over a million gates. Because of these features, FPGAs had been primarily

viewed as glue-logic replacement and rapid-prototyping vehicles. However, the flexibility, capacity, and performance of these devices has opened up completely new avenues in high performance computation, forming the basis of reconfigurable computing.

2.1.1 Configurable Hardware

Most current FPGAs and reconfigurable devices are SRAM-programmable¹ (Figure 2.2a), meaning that SRAM bits are connected to the configuration points in the FPGA, and programming the SRAM bits configures the FPGA. Thus, these chips can be programmed and reprogrammed about as easily as a standard static RAM. In fact, one research project, the PAM project [Vuillemin96], considers a group of one or more FPGAs to be a RAM unit that performs computation between the memory write (sending the configuration information and input data) and memory read (reading the results of the computation). This led to the term “Programmable Active Memory” or PAM.

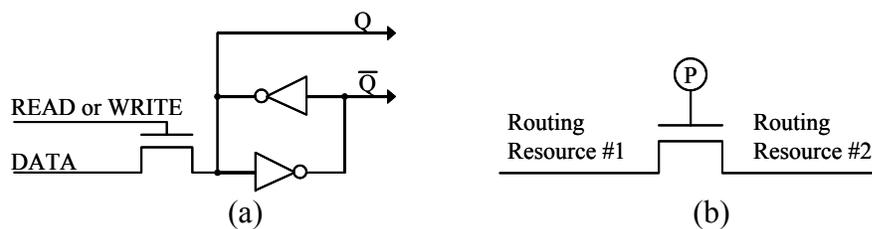


Figure 2.2: (a) A programming bit for SRAM-based FPGAs [Xilinx94, Hauck98a] and (b) a programmable routing connection.

¹ The term “SRAM” is technically incorrect for many FPGA architectures, given that the configuration memory may or may not support random access. In fact, the configuration memory tends to be continually read in order to perform its function. However, this is the generally accepted term in the field and correctly conveys the concept of static volatile memory using an easily understandable label.

One example of how the SRAM configuration points can be used is to control routing within a reconfigurable device [Chow99]. To configure the routing on an FPGA, typically a pass gate structure is employed (Figure 2.2b). Here the programming bit will turn on a routing connection when it is configured with a true value, allowing a signal to flow from one wire to another, and will disconnect these resources when the bit is set to false. With a proper interconnection of these elements, which may include millions of routing choice points within a single device, a rich routing fabric can be created.

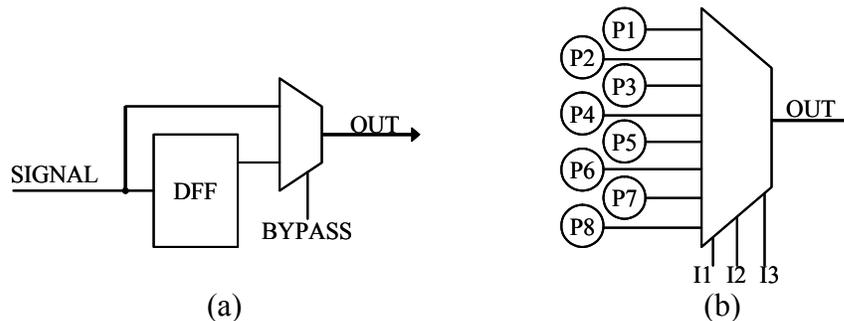


Figure 2.3: (a) A D flip-flop with optional bypass, and (b) a 3-input LUT [Hauck98a].

Another example of how these configuration bits may be used is to control multiplexers, which choose between the output of different logic resources within the array. For example, to provide optional state-holding elements, a D flip-flop (DFF) may be included with a multiplexer to select whether the latched or unlatched signal value will be forwarded (Figure 2.3a). In circuits that require state-holding elements, the programming bits that control the multiplexer are configured to select the DFF output, while circuits not requiring this functionality can choose the bypass route. Similar

structures can choose between other on-chip functionalities, including fixed-logic computation elements, memories, carry chains, or other functions.

Finally, the configuration bits may be used as control signals for a computational unit or as the basis for computation itself. As a control signal, a configuration bit may determine whether an ALU performs an addition, subtraction, or other logic computations. Alternatively, the configuration bits themselves form the result of the computation with a structure such as a lookup table, also known as a LUT (Figure 2.3b). These LUTs are essentially small memories provided for computing arbitrary logic functions. LUTs can compute any function of N inputs (where N is the number of control signals for the LUT's multiplexer) by programming the 2^N programming bits with the truth table of the desired function. Thus, if all programming bits except the one corresponding to the input pattern 111 were set to zero, a 3-input LUT would act as a 3-input AND gate, while programming it with all ones except in 000 would instead compute a NAND.

2.1.2 Traditional FPGAs

Before discussing the detailed architecture design of reconfigurable devices in general, the logic and routing of FPGAs will be described. These concepts apply directly to reconfigurable systems using commercial FPGAs, such as PAM [Vuillemin96] and Splash 2 [Arnold92, Buell96]. Hardware concepts that apply specifically to architectures designed for reconfigurable computing, and variations on the generic FPGA description

provided here, are discussed following this section. More detailed surveys of FPGA architectures can be found elsewhere [Brown92a, Rose93].

Since the introduction of FPGAs in the mid-1980s, many different investigations have examined what computation element(s) should be built into the array [Rose93], including FPGAs with PAL-like product term arrays, multiplexer-based functionality, or basic fixed functions such as simple NAND and XOR gates. Many of these types of architectures have been built. However, it is fairly well established that the best function block for a standard FPGA, a device whose primary role is the implementation of random digital logic, is the one found in the first devices deployed—the LUT (Figure 2.3b). As previously described, an N-input LUT is essentially a memory that can compute any function of up to N inputs when programmed appropriately. This flexibility, with relatively simple routing requirements (each input requires routing to a single multiplexer control input) is very powerful for logic implementation. Although LUTs are less area-efficient than fixed logic blocks, such as a standard NAND gate, most current FPGAs use less than 10% of their chip area for logic, devoting the majority of the silicon real estate to routing resources.

The typical FPGA contains a logic block with one or more 4-input LUT(s), optional D flip-flops (DFF), and some form of fast carry logic (Figure 2.4). The LUTs allow any function to be implemented, providing generic logic resources. The flip-flop can be used for pipelining, registers, state-holding functions for finite state machines, or any other situation where clocking is required. Flip-flops typically include

programmable set/reset lines and clock signals, which may come from global signals routed on special resources, or via the standard interconnect structures from some other input or logic block. The fast carry logic is a special resource provided in the cell to speed up carry-based computations, including addition, parity, wide AND operations, and other functions. These resources bypass the general routing structure, connecting directly between neighbors in the same column. Since very few routing choices exist in the carry chain, this results in less delay on the computation. The inclusion of these resources can significantly speed up carry-based computations.

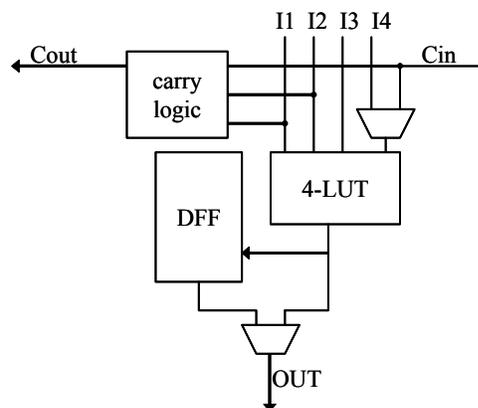


Figure 2.4: A basic logic block, with a 4-input LUT, carry chain, and a D-type flip-flop with bypass.

In addition to experimentation in FPGA logic block architectures, investigation of interconnect structures has also been done. As logic blocks have basically standardized on LUT-based structures, routing resources have become primarily island-style, with logic surrounded by general routing channels.

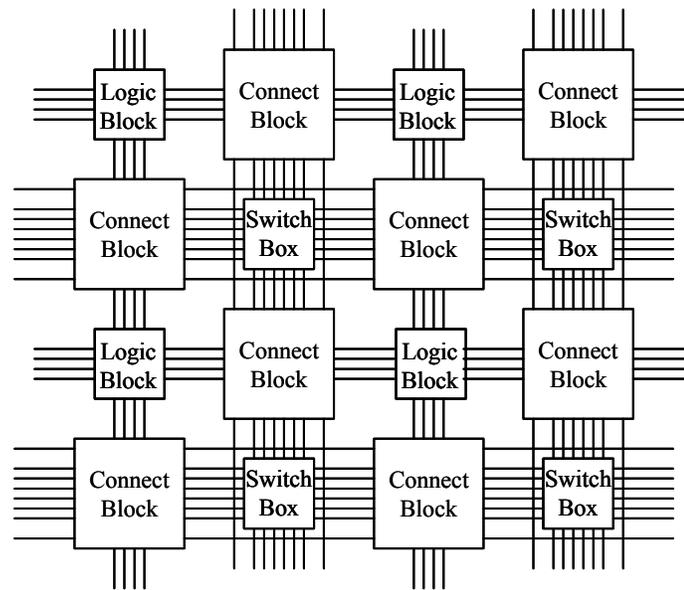


Figure 2.5: A generic island-style FPGA routing architecture.

Most FPGA architectures organize their routing structures as a relatively smooth sea of routing resources, allowing fast and efficient communication along rows and columns of logic blocks. As shown in Figure 2.5, the logic blocks are embedded in a general routing structure, with input and output signals attaching to the routing fabric through connection blocks. The connection blocks provide programmable multiplexers, selecting signals in the given routing channel that will be connected to the logic block's terminals. Signals flow from the logic block into the connection block, then along longer wires within the routing channels. At the switchboxes, connections between the horizontal and vertical routing resources allow signals to change their routing direction. Once a signal has traversed through routing resources and intervening switchboxes, it arrives at the destination logic block through one of its local connection blocks. In this

manner, relatively arbitrary interconnections can be achieved between the logic blocks in the system.

Within a given routing channel, many different lengths of routing resources may exist. Some local interconnections may only move between adjacent logic blocks (i.e. carry chains), providing high-speed local interconnect. Medium length lines may run the width of several logic blocks, providing longer distance interconnect. Finally, long lines that run the entire chip width or height may provide for more global signals. Also, many architectures contain special “global lines” that provide high-speed, and often low skew, connections to all of the logic blocks in the array. These are primarily used for clocks, resets, and other truly global signals.

While the routing architecture of an FPGA is typically quite complex—the connection blocks and switchboxes surrounding a single logic block typically have thousands of programming points—they are designed to support fairly arbitrary interconnection patterns. Most users ignore the exact details of these architectures and allow the automatic physical design tools to choose appropriate resources to achieve a given interconnect pattern.

2.2 Hardware

Reconfigurable computing systems use FPGAs or other programmable hardware to accelerate algorithm execution by mapping compute-intensive calculations to the reconfigurable substrate. These hardware resources are frequently coupled with a

general-purpose microprocessor responsible for controlling the reconfigurable logic and executing program code that cannot be efficiently accelerated. In very closely coupled systems, the reconfigurability lies within customizable functional units on the regular datapath of the microprocessor. Alternatively a reconfigurable computing system can be as loosely coupled as a networked stand-alone unit. Most reconfigurable systems are categorized somewhere between these two extremes, frequently with the reconfigurable hardware acting as a coprocessor to a host microprocessor. The programmable array itself can be comprised of one or more commercially available FPGAs, or can be a custom device designed specifically for reconfigurable computing.

The design of the actual computation blocks within the reconfigurable hardware varies from system to system. Each unit of computation, or logic block, can be as simple as a 3-input lookup table (LUT), or as complex as a 16-bit ALU. This difference in block size is commonly referred to as the granularity of the logic block, where the 3-bit LUT is an example of a fine-grained computational element, and a 16-bit ALU is an example of a coarse-grained unit. Finer grained blocks are useful for bit-level manipulations, while the coarse-grained blocks are better optimized for standard datapath applications. Some architectures employ different sizes or types of blocks within a single reconfigurable array in order to efficiently support different types of computation. For example, memory is frequently embedded within the reconfigurable hardware to provide temporary data storage, forming a heterogeneous structure composed of both logic blocks and memory blocks [Ebeling96, Altera98, Lucent98, Marshall99, Xilinx01].

The routing between the logic blocks within the reconfigurable hardware is also of great importance. Routing contributes significantly to the overall area of the reconfigurable hardware. However, when the percentage of logic blocks used in an FPGA becomes very high, automatic routing tools can have difficulty achieving the necessary connections between the blocks. Therefore, good routing structures are therefore essential to ensuring a design can be successfully placed and routed onto the reconfigurable hardware.

Once a circuit has been programmed onto reconfigurable hardware, it can be used by the host processor during program execution. The run time operation of a reconfigurable system occurs in two distinct phases: configuration and execution. The host processor controls the programming of the hardware by sending it a stream of configuration data, which is used to define the actual hardware operation. Configurations can be loaded either only at the start of the program, or periodically during runtime, depending on the design of the system. Further discussion of run-time reconfiguration (the dynamic reconfiguration of devices during execution) appears in section 2.4.

The actual execution model of the reconfigurable hardware varies among systems. For example, the NAPA system [Rupp98] by default suspends the execution of the host processor during execution on the reconfigurable hardware. However, simultaneous computation can occur with the use of fork and join primitives, similar to multiprocessor programming. REMARC [Miyamori98] is a reconfigurable system that uses a pipelined set of execution phases within the reconfigurable hardware. These pipeline stages

overlap with the pipeline stages of the host processor, allowing for simultaneous execution. In the Chimaera system [Hauck97], the reconfigurable hardware is constantly executing based upon the input values held in a subset of the host processor's registers. A call to the Chimaera unit is in actuality only a fetch of the result value. This value is stable and valid after the correct input values have been written to the registers and have filtered through the computation.

2.2.1 Microprocessor Coupling

Frequently, reconfigurable hardware is coupled with a traditional microprocessor. Programmable logic is sometimes inefficient at implementing certain operations, such as variable-length loops and branch control. In order to run an application in a reconfigurable computing system most efficiently, those areas of the program that cannot be easily mapped to the reconfigurable logic are executed on a host microprocessor. Meanwhile, the areas with a high density of computation that can benefit from implementation in hardware are mapped to the reconfigurable logic. Additionally, current run-time reconfigurable hardware generally requires an external structure, such as a processor, to control when reconfigurations should occur, and which configurations should be loaded.

For the systems that use a microprocessor in conjunction with reconfigurable logic, there are several ways in which these two computation structures may be coupled, as Figure 2.6 shows. First, reconfigurable hardware can be used solely to provide

reconfigurable functional units within a host processor [Razdan94, Wittig96, Hauck97]. This allows for a traditional programming environment with the addition of custom instructions that may change over time. Here, the reconfigurable units execute as functional units on the main microprocessor datapath, with registers used to hold the input and output operands.

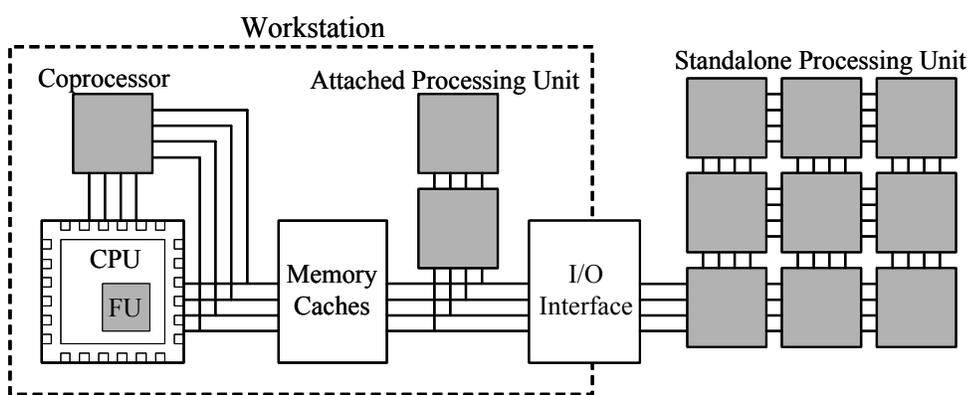


Figure 2.6: Different levels of coupling in a reconfigurable system [Hauck98a]. Reconfigurable logic is shaded.

Second, a reconfigurable unit may be used as a coprocessor [Hauser97, Miyamori98, Rupp98, Chameleon00]. A coprocessor is generally larger than a functional unit, and can perform computations without the constant supervision of the host processor. Instead, the processor initializes the reconfigurable hardware and either sends the necessary data to the logic, or provides information on the location of the data in memory. The reconfigurable unit performs the actual computations independently of the main processor, and returns the results after completion. This type of coupling allows the reconfigurable logic to operate for a large number of cycles without intervention from the

host processor, and generally permits the host processor and the reconfigurable logic to execute simultaneously. This reduces the overhead incurred by the use of the reconfigurable logic, compared to a reconfigurable functional unit that must communicate with the host processor each time a reconfigurable “instruction” is used. An idea that is a hybrid between the first and second coupling methods is the use of programmable hardware within a configurable cache [Kim00b]. In this situation, the reconfigurable logic is embedded into the data cache, which can be used as either a regular cache or as an additional computing resource, depending on the target application.

Third, an attached reconfigurable processing unit [Vuillemin96, Annapolis98, Laufer99] behaves like an additional processor in a multiprocessor system or an additional compute engine accessed semi-frequently through external I/O. The host processor's data cache is not visible to the attached reconfigurable processing unit, leading to a greater delay in communication between the host processor and the reconfigurable hardware when communicating configuration information, input data, and results. The communication is performed through specialized primitives similar to multiprocessor systems. This type of reconfigurable hardware allows a great deal of computation independence by shifting large chunks of a computation over to the reconfigurable hardware.

Finally, the most loosely coupled form of reconfigurable hardware is an external stand-alone processing unit [Quickturn99a, Quickturn99b], which communicates infrequently with a host processor (if present). This model is similar to networked

workstations, where processing can occur for very long periods of time without much communication. However, the large multi-FPGA systems such as those from Quickturn are marketed towards emulation rather than reconfigurable computing.

Each of these styles has distinct benefits and drawbacks. The tighter the integration of the reconfigurable hardware, the more frequently it can be used within an application or set of applications due to a lower communication overhead. However, the hardware is unable to operate for significant portions of time without intervention from a host processor, and the amount of reconfigurable logic available is often quite limited. The more loosely coupled styles allow for greater parallelism in program execution, but suffer from higher communications overhead. In applications that require a great deal of communication, this can reduce or remove any acceleration benefits gained through the use of reconfigurable hardware.

2.2.2 Logic Block Granularity

Most reconfigurable hardware is based upon a set of computation structures that are repeated to form an array. These structures, commonly called logic blocks or cells, vary in complexity from a very small and simple block that can calculate a function of only three inputs, to a structure that is essentially a 16-bit ALU. Some of these block types are configurable – the actual operation is determined by a set of loaded configuration data. Other blocks are fixed structures, and the configurability lies in the

connections between them. Granularity refers to the size and complexity of the computing blocks.

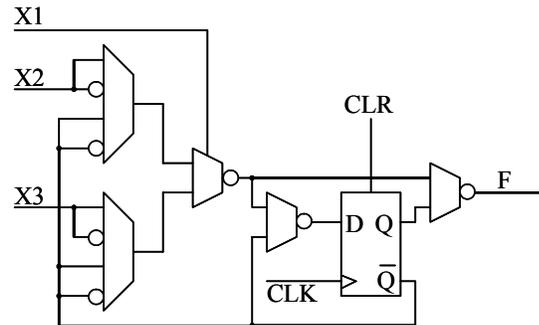


Figure 2.7: The functional unit from a Xilinx 6200 cell [Xilinx96].

An example of a very fine-grained logic block can be found in the Xilinx 6200 series of FPGAs [Xilinx96], shown in Figure 2.7. The functional unit from one of these cells can implement any two-input function and some three-input functions. Although this type of architecture is useful for very fine-grained bit manipulation, it is too fine-grained to efficiently implement many types of circuits, such as multipliers. Similarly, finite state machines are frequently too complex to easily map to a reasonable number of very fine-grained logic blocks. However, finite state machines are also too dependent upon single bit values to be efficiently implemented in a very coarse-grained architecture. This type of circuit is more suited to an architecture that provides more connections and computational power per logic block, while still providing sufficient capability for bit-level manipulation.

The logic cell in the Altera FLEX 10K architecture [Altera98] is a fine-grained structure that is somewhat coarser than the 6200. This architecture mainly consists of a

single 4-input LUT with a flip-flop. Also, there is specialized carry chain circuitry that helps to accelerate addition, parity, and other operations that use a carry chain. These types of logic blocks are useful for fine-grained bit-level manipulation of data, which is frequently found in encryption and image processing applications. Because the cells are fine-grained, computation structures of arbitrary bit widths can be created, which allows the implementation of datapath circuits that are based on data widths not implemented on the host processor (5 bit multiply, 21 bit addition, etc). Reconfigurable hardware can not only take advantage of small bit widths, but also large data widths. When a program uses bit widths in excess of what is normally available in a host processor, the processor must perform the computations using a number of extra steps to accommodate the full data width. A fine-grained architecture can implement the full bit width in a single step, without the fetching, decoding, and execution of additional instructions, provided enough logic cells are available.

A number of reconfigurable systems use a medium-grained logic block [Xilinx94, Hauser97, Haynes98, Lucent98, Marshall99]. Garp [Hauser97] is designed to perform a number of different operations on up to four 2-bit inputs. Another medium-grained structure was designed to be embedded inside a general-purpose FPGA to implement multipliers of a configurable bit-width [Haynes98]. The logic block used in the multiplier FPGA is capable of implementing a 4×4 multiplication, or can be cascaded into larger structures. The CHESS architecture [Marshall99] also operates on 4-bit values, with each cell acting as a 4-bit ALU. Medium-grained logic blocks can

implement datapath circuits of varying bit widths, similar to the fine-grained structures. The ability to perform more complex operations of a greater number of inputs permits this structure to efficiently implement a wider variety of operations.

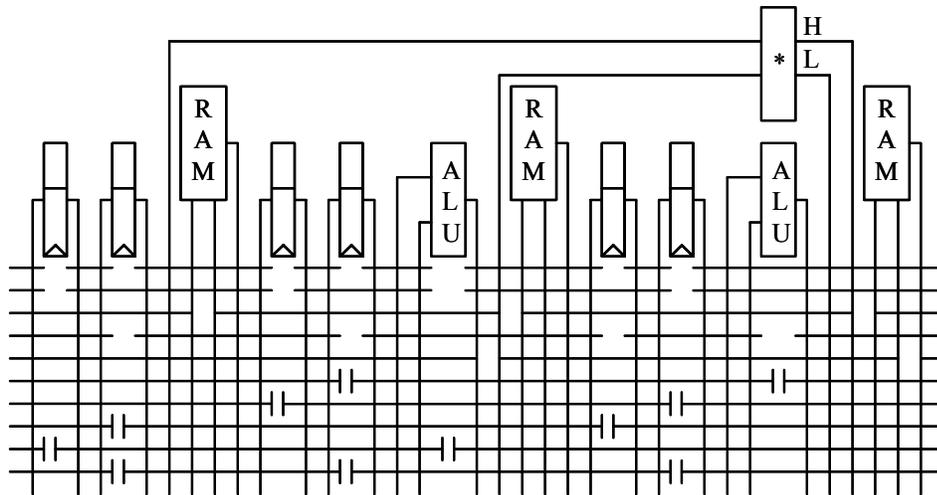


Figure 2.8: One cell in the RaPiD-I reconfigurable architecture [Ebeling96]. The registers, RAM, ALUs, and multiplier all operate on 16-bit values. The multiplier outputs a 32-bit result, split into the high 16 bits and the low 16 bits. All routing lines shown are 16-bit wide busses. The short parallel lines on the busses represent configurable bus connectors.

Very coarse-grained architectures are used primarily to implement word-width datapath circuits. Because the logic blocks used are optimized for large computations, they perform these operations much more quickly (and consume less chip area) than a set of smaller cells connected to form the same type of structure. Because their composition is static, they cannot leverage optimizations in the size of operands. The RaPiD-I architecture [Ebeling96], shown in Figure 2.8, and the Chameleon architecture [Chameleon00], are examples of very coarse-grained designs. Each of these architectures is composed of word-sized adders, multipliers, and registers. Even when adding numbers

smaller than the full word size, all of the bits in the full word size are computed, which can result in unnecessary area and speed overheads. However, these coarse-grained architectures are much more efficient than fine-grained architectures for implementing functions closer to their basic word size.

An alternate form of a coarse-grained system consists of logic blocks that are very small processors, potentially each with its own instruction memory and/or data values. The REMARC architecture [Miyamori98] is composed of an 8×8 array of 16 bit processors. Each of these processors uses its own instruction memory in conjunction with a global program counter. This style of architecture closely resembles a single-chip multiprocessor with much simpler component processors, as the system is meant to be coupled with a host processor. The RAW project [Moritz98] is another example of a reconfigurable architecture based on a multi-processor design.

The granularity of the FPGA can also have an effect on the reconfiguration time of the device. This is an important issue for run-time reconfiguration, discussed in further depth in section 2.4. A fine-grained array has many configuration points to perform very small computations, and thus requires more data bits during configuration.

2.2.3 Heterogeneous Arrays

Greater performance or flexibility in computation can be achieved in reconfigurable systems through the use of a heterogeneous structure, where the capabilities of the logic cells vary throughout the system. For example, reconfigurable

systems may provide multiplier function blocks embedded within the reconfigurable hardware [Ebeling96, Haynes98, Chameleon00, Xilinx02, Altera03a]. Because multiplication is a difficult computation to implement efficiently in a traditional FPGA structure, the custom multiplication hardware embedded within a reconfigurable array allows a system to perform even that function well.

Another common structure used in heterogeneous devices is a memory block. Memory blocks can be scattered throughout the reconfigurable hardware, permitting the storage and quick access of frequently used data and variables due to the proximity of the memory to the logic blocks that access it. Embedded memory structures come in two forms. The first is simply the use of available LUTs as RAM structures, such as in the Xilinx 4000 series [Xilinx94] and Virtex [Xilinx01] FPGAs. Although making these very small blocks into a larger RAM structure introduces overhead to the memory system, it does provide local, variable width memory structures.

The second form is that of the dedicated memory block. Several architectures include memory blocks within their array, including some of the Xilinx [Xilinx01, Xilinx02] and Altera [Altera98, Altera03a] FPGAs, Actel's ProASIC 500K series [Actel02], and the CS2000 RCP (Reconfigurable Communications Processor) device from Chameleon Systems, Inc. [Chameleon00]. These memory blocks have greater performance in large sizes than similar-sized structures built from many small LUTs. While these structures are somewhat less flexible than the LUT-based memories, they also allow some customization. For example, the Altera FLEX 10K FPGA [Altera98]

provides embedded memories with a limited total number of wires, but allows a trade-off between the number of address lines and the data bit width.

When embedded memories are not used for data storage by a particular configuration, their occupied area need not be wasted. By using the address lines of the memory as function inputs and the values stored in the memory as function outputs, logical expressions of a large number of inputs can be emulated [Altera98, Cong98, Wilton98, Heile99]. Since there may be more than one value output from the memory on a read operation, the memory structure can perform multiple different computations (one for each bit of data output), provided all the necessary inputs appear on the address lines. In this manner, the embedded RAM behaves the same as a very large multi-output LUT. Therefore, embedded memories allow a programmer or a synthesis tool to adjust between logic and memory usage in order to achieve higher area efficiency.

Furthermore, some commercial FPGA companies have included entire microprocessors as embedded structures within their FPGAs. Altera's ARM9-based Excalibur device combines reconfigurable hardware with an embedded ARM9 processor core [Altera01], and the Xilinx Virtex-II Pro FPGA includes up to four PowerPC processor cores [Xilinx03a]. These types of devices are also discussed in section 3.2.

2.2.4 Routing Resources

Interconnect resources in a reconfigurable architecture connect the programmable logic elements of the device together. These resources are usually configurable, where

the path of a signal is determined at compile or run-time rather than fabrication time. This flexible interconnect between logic blocks or computational elements allows a wide variety of circuit structures with different interconnect requirements to be mapped to the reconfigurable hardware. For example, the routing for FPGAs is generally island-style, with logic surrounded by routing channels containing several wires of potentially different lengths. This type of routing architecture can vary in a number of features, including the ratio of wires to logic in the system, the length of the wires, and whether the wires are connected in a segmented or hierarchical manner.

Designing efficient routing structures for FPGAs and reconfigurable systems involves examining the logic vs. routing area trade-off within reconfigurable architectures. One group has argued that the interconnect should constitute a much higher proportion of area in order to allow for successful routing under high logic utilization conditions [Takahara98]. However, efficient routing usage may be of more importance than high LUT utilization [DeHon99]. Routing resources occupy a much larger part of the area than the logic resources, so the most area-efficient designs are frequently those that optimize their use of the routing resources. Additionally, the amount of required routing does not grow linearly with the amount of logic present, so larger devices require even greater amounts of routing per logic block than small ones [Trimberger97].

There are two different types of structures used to provide local and global routing resources, as shown in Figure 2.9. Segmented routing [Betz99, Chow99]

accommodates local communications traffic with short wires that can be connected together with switchboxes to emulate longer wires. Segmented routing structures frequently also contain separate longer wires that allow signals to travel efficiently over long distances without passing through a great number of switches.

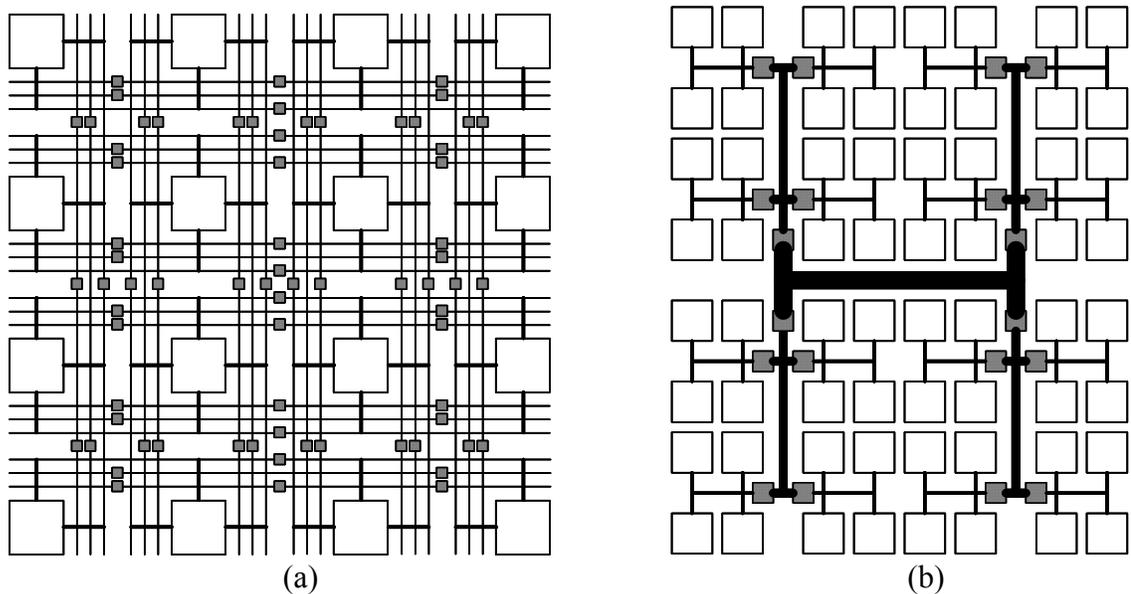


Figure 2.9: (a) Segmented and (b) hierarchical routing structures. The white boxes are logic blocks, while the dark boxes are connection switches.

Hierarchical routing [Aggarwal94, Lai97, Tsu99] is another style of routing architecture. Routing within a group, or cluster, of logic blocks occurs at the local level, and these wires only connect within that cluster. At the boundaries of these clusters longer wires connect the different clusters together. This is potentially repeated at a number of levels. The idea behind using hierarchical structures is that most communication should be local and only a limited amount will traverse long distances (provided good placements are found for implemented circuits).

Because routing can occupy a large percentage of a reconfigurable device, careful selection of the routing structure is critical to FPGA design. If the available wires are much longer than what is needed to route a signal, the excess wire is wasted. On the other hand, if the wires available are shorter than necessary, the signal must either pass through switchboxes connecting short wires into longer ones, or through levels of the routing hierarchy. This induces additional delay, slowing the overall operations of the circuit, and the switchbox circuitry occupies area that could be better used for additional logic or wires.

There are some alternatives to the island-style of routing resources. Architectures such as RaPiD [Ebeling96], Lattice's ORCA4 [Lattice03], and CHESS [Marshall99] have bus-based routing, where multiple bits are routed together as a bundle. This type of routing is also common in the one-dimensional type of architecture, as discussed in the next section.

2.2.5 One-Dimensional Structures

Most current FPGAs are of the two-dimensional variety, as shown in Figure 2.10a. This allows for a great deal of flexibility because a signal can be routed on a nearly arbitrary path, but providing this level of routing flexibility requires a great deal of routing area. It also complicates the placement and routing software, because it must consider a very large number of possibilities.

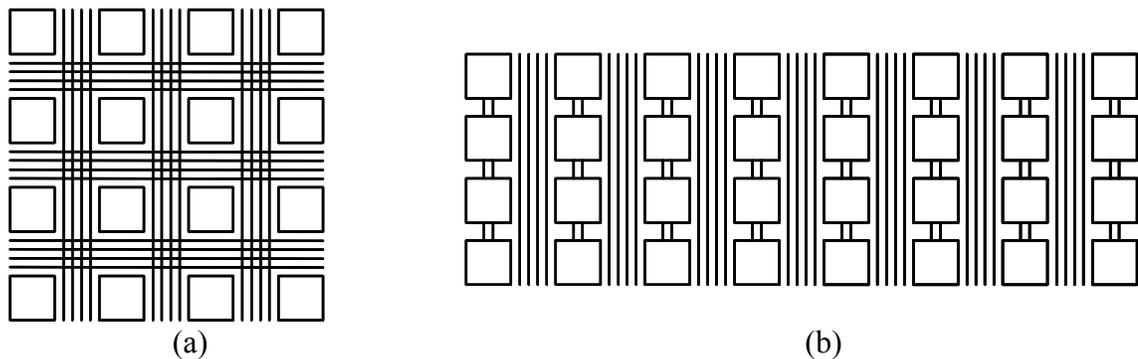


Figure 2.10: (a) A traditional two-dimensional island-style routing structure, and (b) a one-dimensional routing structure. The white boxes represent logic elements.

One solution is to use a more one-dimensional style of architecture, depicted in Figure 2.10b. Here, placement is restricted along one axis. Routing is simplified, because it is generally along a single dimension as well. However, this restriction can become a drawback when implementing circuits with heavy routing requirements. A two-dimensional implementation can sometimes provide a wider variety of connections (with a corresponding increase in silicon area) that can be made between two points, sometimes allowing a tool to route around a bottleneck. If the cross-section of signals in a one-dimensional array exceeds the number of available tracks, there are no other options, and the routing process will fail. However, routing architectures of both types will fail if the routing requirements of the netlist exceeds the available routing resources of the hardware.

Several different reconfigurable systems have been designed with a one-dimensional routing structure. Both Garp [Hauser97] and Chimaera [Hauck97] are structures that provide cells which compute a small number of bit positions, and a row of

these cells together computes the full data word. Since a row can only be used by a single configuration, and each configuration occupies some number of complete rows, these designs are essentially one-dimensional. Although multiple narrow-width computations can fit within a single row, these structures are optimized for word-based computations that occupy the entire row. The NAPA architecture [Rupp98] is similar, with a full column of cells acting as the atomic unit for a configuration, as is PipeRench [Cadambi98, Goldstein00].

In some systems, the computation blocks in a one-dimensional structure operate on word-width values instead of single bits, so busses are routed instead of individual values. This can decrease the required routing time, because the bits of a bus are considered together rather than as individual routes. As shown in Figure 2.8, RaPiD [Ebeling96] is a one-dimensional design that includes only word-width processing elements. The different computation units are organized in a single dimension along the horizontal axis. The general flow of information follows this layout, with the major routing busses laid out in a horizontal manner. All routing is of word-sized values, so all routing is of busses, not individual wires.

2.2.6 Hardware Summary

The design of reconfigurable hardware varies greatly from system to system. The reconfigurable logic may be used as a configurable functional unit or a multi-FPGA stand-alone unit. Within the reconfigurable logic itself, the complexity of the core

computational units, or logic blocks, vary from very simple to extremely complex, some implementing a 4-bit ALU or even a 16×16 multiplication. These blocks are not required to be uniform throughout the array, and using different types of blocks can add high-performance functionality in specialized computation circuitry, or expanded storage in embedded memory blocks. Routing resources also offer a variety of choices, primarily in the amount, length, and organization of the wires. Systems have been developed that fit into many different points within this design space.

2.3 Software

Although reconfigurable hardware has been shown to provide significant performance benefits for some applications, it will be ignored by application programmers unless they can easily incorporate its use into their systems. This requires a software design environment able to create configurations for the reconfigurable hardware. This software can range from a software assist in manual circuit creation to a complete automated circuit design system. Manual circuit description is a powerful method for the creation of high-quality circuit designs. However, it requires extensive knowledge of the particular reconfigurable system employed, plus a significant amount of design time. Alternatively, an automatic compilation system provides a quick and easy way to program for reconfigurable systems. It makes the use of reconfigurable hardware more accessible to general application programmers, but quality may suffer.

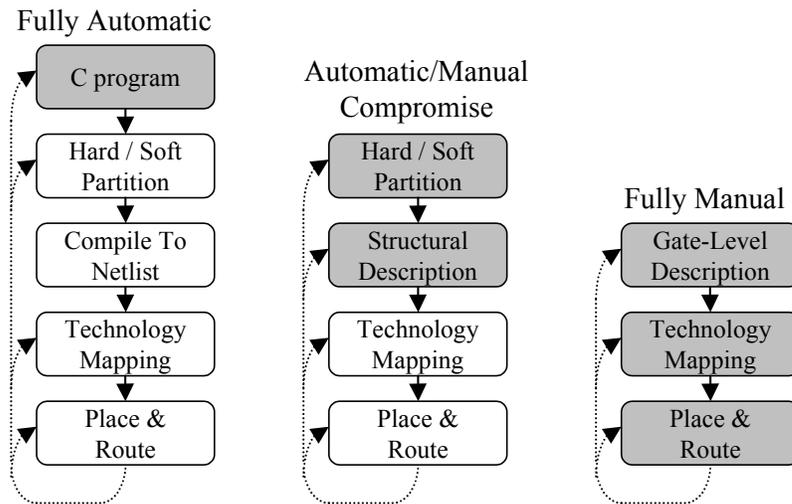


Figure 2.11: Three possible design flows for algorithm implementation on a reconfigurable system. Grey stages indicate manual effort on the part of the designer, while white stages are done automatically. The dotted lines represent paths to improve the resulting circuit. It should be noted that the middle design cycle is only one of the possible compromises between automatic and manual design.

Both for manual and automatic circuit creation, the design process proceeds through a number of distinct phases, shown in Figure 2.11. Circuit specification is the process of describing the functions to be placed on the reconfigurable hardware. This can be done by simply writing a program in C that represents the functionality of the algorithm to be implemented in hardware. It can also be as complex as specifying the inputs, outputs, and operation of each basic building block in the reconfigurable system. Between these extremes is circuit specification using generic complex components, such as adders and multipliers, which will be mapped to the actual hardware later in the design process. For descriptions in a high level language (HLL), such as C/C++ or Java, or ones using complex building blocks, this code must be compiled into a netlist of gate-level components. Gates or computational components (such as ALUs and multipliers) are

created to perform the arithmetic and logic operations within the program, and structures to handle the program control, such as loop iterations and branching operations. Given a structural description, either generated from a HLL or specified by the user, each complex structure is replaced with a network of basic gates that performs that function.

Once a detailed gate-level circuit description is created, it must be translated to the logic elements of the reconfigurable hardware. This stage is called technology mapping, and it is dependent upon the exact target architecture. For a LUT-based architecture, this stage partitions the circuit into a number of small sub-functions, each mapped to its own LUT. Some architectures, such as the Xilinx 4000 series [Xilinx94], contain multiple LUTs per logic cell. These LUTs can be used either separately to generate small functions, or together to generate some wider-input functions. By taking advantage of multiple LUTs and the internal routing within a single logic cell, sub-circuits containing too many inputs to implement using a single LUT can efficiently be mapped into the FPGA architecture.

For reconfigurable structures with embedded memory blocks, the mapping stage may also consider using these memories as logic units when they are not used for data storage. The memories act as very large LUTs, where the number of inputs is equal to the number of address lines. In order to use these memories as logic, the mapping software must analyze how much of the memory blocks are actually used as storage in a given mapping. It must then determine which are available to implement logic, and what parts of the circuit are best mapped to the memory [Cong98, Wilton98].

After mapping the circuit, the resulting blocks must be placed onto the reconfigurable hardware. Each block is assigned to a specific location within the hardware, ideally close to the other logic blocks with which it communicates. As FPGA capacities increase, the placement phase of circuit mapping becomes more and more time consuming. Floorplanning is a technique that can alleviate some of this cost. A floorplanning algorithm first partitions the logic cells into clusters, where cells with a large amount of communication are grouped together. Next, the clusters are placed as units onto regions of the reconfigurable hardware. Once this global placement is complete, the actual placement algorithm performs detailed placement of the individual logic blocks within the boundaries assigned to the cluster [Sankar99].

The use of a floorplanning tool is particularly helpful for situations where the circuit structure being mapped is of a datapath type. Large computational components or macros that are found in datapath circuits are frequently composed of highly regular logic. These structures are placed as entire units, and their component cells are restricted to the floorplanned location [Shi97, Emmert99]. This encourages the placer to find a very regular placement of these logic cells, potentially resulting in a higher performance circuit layout. Another technique for the mapping and placement of datapath elements is to perform these steps simultaneously [Callahan98]. This method also exploits the regularity of the datapath elements to generate mappings and placements quickly and efficiently.

Floorplanning is also important when dealing with hierarchically structured reconfigurable hardware. In these architectures, the available resources are grouped by the logic or routing hierarchy of the hardware. Because performance is best when routing lengths are minimized, the placement should group cells into a logic cluster on the hardware if those cells require a great deal of inter-communication or form part of a critical path of the circuit [Krupnova97, Senouci98].

After the optional floorplanning step, the individual logic blocks are placed into specific logic cells. The simulated annealing technique [Sechen88, Shahookar91, Betz97, Sankar99] is commonly used. This method takes an initial placement of the system, which can be generated (pseudo-) randomly, and performs a series of “moves” on that layout. A move is simply the changing of the location of a single logic cell, or the exchanging of locations of two logic cells. These moves are attempted one at a time using random target locations. If a move improves the layout, then the layout is changed to reflect that move. If a move is considered to be undesirable, then it is only accepted some of the time. Initially, many “bad” moves are accepted. As the algorithm progresses, the likelihood of accepting a bad move decreases until very few undesirable moves are accepted at the end of execution. Also, the degree to which a move increases the cost affects its probability of acceptance, with mildly bad moves more likely to be accepted than very bad moves. Accepting a few bad moves helps to avoid any local minima in the placement space. Other algorithms exist that are more deterministic [Gehring96, Callahan98, Budiu99], although they search a smaller area of the placement

space for a solution, and therefore may be unable to find a solution which meets performance requirements if a design uses a high percentage of the reconfigurable resources.

In the final step, the different reconfigurable components comprising the application circuit are connected during the routing stage. Particular signals are assigned to specific portions of the routing resources of the reconfigurable hardware. This can become difficult if the placement causes many connected components to be placed far from one another, because signals that travel long distances use more routing resources than those that travel shorter ones. A good placement is therefore essential to the routing process.

Limited routing resources present challenges to routing for FPGAs and reconfigurable systems. The goal of general hardware design is to minimize the number of routing tracks used in a channel between rows of computation units, but the channels can be made as wide as necessary. In reconfigurable systems, however, the number of available routing tracks is determined at fabrication time, and therefore the routing software must perform within these boundaries. Thus, FPGA routing concentrates on minimizing congestion within the available tracks [Brown92b, McMurchie95, Chan97, Wu97]. Because routing is one of the more time-intensive portions of the design cycle, it can be helpful to determine if a placed circuit can be routed before actually performing the routing step. This quickly informs the designer if changes need to be made to the layout or if a larger reconfigurable structure is required [Wood97, Swartz98].

2.4 Run-Time Reconfiguration

The areas of a program that can be accelerated through the use of reconfigurable hardware are frequently too numerous or complex to be loaded simultaneously onto the available hardware. In these cases, it is beneficial to be able to swap different configurations in and out of the reconfigurable hardware as they are needed during program execution, as in Figure 2.12. This concept is known as run-time reconfiguration (RTR).

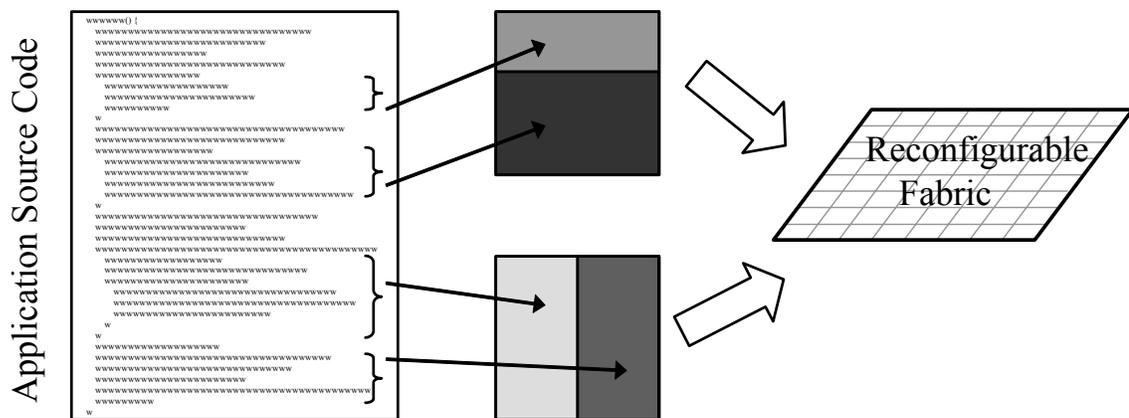


Figure 2.12: Applications which are too large to entirely fit on the reconfigurable hardware can be partitioned into two or more smaller configurations that can occupy the hardware at different times.

Run-time reconfiguration is based upon the concept of virtual hardware, which is similar to virtual memory. Here, the physical hardware is much smaller than the sum of the resources required by each configuration. Instead of reducing the number of mapped configurations, they can instead be swapped in and out of the actual hardware as they are needed. Because run-time reconfiguration allows more sections of an application to be

mapped into hardware than can be fit in a non-run-time reconfigurable system, a greater portion of the program can be accelerated.

During a single program's execution, configurations are swapped in and out of the reconfigurable hardware. Some of these configurations will likely require access to the results of other configurations. Configurations that are active at different periods in time need a method to communicate with one another. This communication can be performed through the use of registers [Ebeling96, Cadambi98, Rupp98, Scalera98], the contents of which can remain intact between reconfigurations. This allows one configuration to store a value, and a later configuration to read back that value for use in further computations. An alternative for reconfigurable systems that do not include state-holding devices is to write the result back to registers or memory external to the reconfigurable array, which is then read back by successive configurations [Hauck97].

Because run-time reconfigurable systems involve reconfiguration during program execution, the reconfiguration must be done as efficiently and as quickly as possible. This is in order to ensure that the overhead of the reconfiguration does not eclipse the benefit gained by hardware acceleration. Stalling execution of either the host processor or the reconfigurable hardware because of configuration is clearly undesirable. In the DISC II system, from 25% [Wirthlin96] to 71% [Wirthlin95] of execution time is spent in reconfiguration, while in the UCLA ATR work this figure can rise to over 98.5% [Mangione-Smith99]. If the delays caused by reconfiguration are reduced, performance

can be greatly increased. Therefore, fast configuration is important for run-time reconfigurable systems, and is discussed more in the following section.

2.4.1 Fast Configuration

There are a number of different approaches that reduce configuration overhead. First, the configuration architecture can make switching configurations faster. Second, loading of the configurations can be timed such that the configuration overlaps as much as possible with the execution of instructions by the host processor. Third, compression techniques can be introduced to decrease the amount of configuration data that must be transferred to the system. Finally, the actual process of transferring the data from the host processor to the reconfigurable hardware can be modified to include a configuration cache, which would provide a faster reconfiguration.

Configuration Architectures

There are several configuration memory styles that can be used with reconfigurable systems. A single context device is a serially programmed chip that requires a complete reconfiguration to change any of the programming bits. This is the most common type of commercial device, yet as discussed later, it is not as conducive to reconfigurable computing as some other designs. A complete reprogram of a single context chip can require milliseconds to seconds.

A multicontext device has multiple layers of programming bits, each of which can be active at a different point in time. Each of these layers can be thought of as a separate

single context memory, and one context can be in execution while another context is being loaded in the background. The benefit is not only overlapping FPGA execution and configuration, but also switching between the programmed contexts very quickly, sometimes even in a single clock cycle.

Devices that can be selectively programmed without a complete reconfiguration are called partially reconfigurable. The partially reconfigurable architecture is also more suited to run-time reconfiguration than the single context, because small areas of the array can be modified without requiring reprogramming of the entire logic array. However, inefficiencies can arise in partially reconfigurable architectures if two partial configurations are supposed to be located at overlapping physical locations on the FPGA. If these configurations are repeatedly used one after another, they must be swapped in and out of the array each time. This type of conflict could negate much of the benefit achieved by partially reconfigurable systems.

A better solution allows the final placement of the configurations to occur at run-time, allowing for run-time relocation of those configurations [Li00, Compton02c]. Using relocation, a new configuration may be placed onto the reconfigurable array where it will cause minimum conflict with other needed configurations already present on the hardware. A number of different systems support run-time relocation, including Chimaera [Hauck97], Garp [Hauser97], and PipeRench [Cadambi98, Goldstein00].

Even with relocation, partially reconfigurable hardware can still suffer from some placement conflicts. Over time, as a partially reconfigurable device loads and unloads

configurations, the location of the unoccupied area on the array is likely to become fragmented, similar to what occurs in memory systems when RAM is allocated and deallocated. There may be enough empty area on the device to hold an incoming configuration, but it may be distributed throughout the array. A configuration normally requires a contiguous region of the chip, so it would have to overwrite a portion of a valid configuration in order to be placed onto the reconfigurable hardware. However, a system that incorporates the ability to perform defragmentation of the reconfigurable array could consolidate the unused area by moving valid configurations to new locations [Diessel97, Compton02c]. This area can then be used by incoming configurations, without overwriting any of the moved configurations.

Configuration Prefetching

Performance improves when the hardware reconfiguration overlaps with computations performed by the host processor, because programming the reconfigurable hardware requires from milliseconds to seconds to accomplish. Overlapping reconfiguration and processor execution prevents the host processor from stalling while it is waiting for the configuration to finish, and hides the configuration time from the program execution. Configuration prefetching [Hauck98b] attempts to leverage this overlap by determining when to initiate reconfiguration of the hardware in order to maximize overlap with useful computation on the host processor. It also seeks to

minimize the chance that a configuration will be prefetched falsely, incorrectly overwriting a configuration that will be needed.

Configuration Compression

Unfortunately, there will always be cases when configuration overheads cannot be successfully hidden using a prefetching technique. This can occur when a conditional branch occurs immediately before the use of a configuration, potentially making a 100% correct prefetch prediction impossible, or when multiple configurations or contexts must be loaded in quick succession. In these cases, the delay incurred is minimized when the amount of data transferred from the host processor to the reconfigurable array is minimized. Configuration compression can be used to compact this configuration information [Hauck98c, Hauck99, Li99, Dandalis01].

One form of configuration compression has already been implemented in a commercial system. The Xilinx 6200 series of FPGA [Xilinx96] contains wildcarding hardware, which provides a method to program multiple logic cells in a partially configurable FPGA with a single address and data value. This is accomplished by setting a special register to indicate which of the address bits should behave as "don't-care" values, resolving to multiple addresses for configuration. For example, suppose two configuration addresses, 00010 and 00110, are both to be programmed with the same value. By setting the wildcard register to 00100, the address value sent is interpreted as 00X10 and both these locations are programmed using either of the two addresses above

in a single operation. Furthermore, “Don’t Care” values in the configuration stream could be used to allow areas with similar but not identical configuration data values to also be programmed simultaneously [Li99]. Wildcarding can be used to reduce configuration time [Hauck98c] as well as the storage required for the configuration. Also, partially reconfigurable systems can take advantage of previously programmed areas of the hardware. If two successive configurations share some configuration data, those configuration locations need not be reprogrammed. Configuration time can be reduced through the identification of these common components and the calculation of the incremental configurations that must be loaded [Luk97, Shirazi98].

Alternately, similar operations can be grouped together to form a single configuration that contains extra control circuitry in order to implement the various functions within the group [Kastrup99]. By creating larger configurations from groups of smaller configurations, more operations can be present on chip simultaneously, reducing the configuration overhead. However, this method imposes some area and execution penalties, creating a trade-off between reduced reconfiguration overhead and faster execution with a smaller area.

Configuration Caching

Because much of the delay caused by configuration is due to the distance between the host processor and the reconfigurable hardware, as well as the reading of the configuration data from a file or main memory, a configuration cache can potentially

reduce the cost of reconfiguration [Deshpande99, Li00]. Storing the configurations in a fast memory very close to the reconfigurable array instead of the main memory of the system accelerates data transfer during reconfiguration and reduces the overall configuration time required. Additionally, a configuration cache can allow for specialized direct output to the reconfigurable hardware [Compton00]. This output can leverage the close proximity of the cache by providing high-bandwidth communications that would facilitate wide parallel loading of the configuration data, further reducing configuration times.

2.5 Reconfigurable Computing Summary

Reconfigurable computing is emerging as an important area of research in computer architectures and software systems. An application can be greatly accelerated by placing the computationally intense portions of an application onto reconfigurable hardware. Reconfigurable computing combines many benefits of both software and ASIC implementations. Like software, the mapped circuit is flexible, and can be changed over the lifetime of the system. Similar to an ASIC, reconfigurable systems provide a method to map circuits into hardware. Reconfigurable systems therefore have the potential to achieve far greater performance than software as a result of bypassing the fetch-decode-execute cycle of traditional microprocessors, and possibly exploiting a greater degree of parallelism.

Reconfigurable hardware systems come in many forms, including a configurable functional unit integrated directly into a CPU; a reconfigurable coprocessor coupled with a host microprocessor; and a multi-FPGA stand-alone unit. The level of coupling, granularity of computation structures, and form of routing resources are all key points in the design of reconfigurable systems. The use of heterogeneous structures can also greatly add to the overall performance of the final design.

Compilation tools for reconfigurable systems range from simple tools that aid in the manual design and placement of circuits, to fully automatic design suites that create circuits and wrapper software executables from program code written in a high-level language. The variety of tools available allows designers to choose between manual and automatic circuit creation for any or all of the design steps. Although automatic tools greatly simplify the design process, manual creation is still important for performance-driven applications.

Finally, run-time reconfiguration provides a method to accelerate a greater portion of a given application by allowing the configuration of the hardware to change over execution time. Because of the delays associated with configuration, this style of computing requires that reconfiguration be performed in a very efficient manner. Multicontext and partially reconfigurable FPGAs are both designed to improve the time required for reconfiguration. Hardware optimizations, such as wildcarding, run-time relocation, and defragmentation, further decrease configuration overhead in a partially

reconfigurable design. Software techniques to enable fast configuration, including prefetching and incremental configuration calculation, can also reduce overhead.

Reconfigurable computing systems provide a high-performance alternative to software-only implementations due to their ability to greatly accelerate program execution, providing a high-performance alternative to software-only implementations. However, no one hardware design has emerged as the clear pinnacle of reconfigurable design. Although general-purpose FPGA structures have standardized into LUT-based architectures, groups designing hardware for reconfigurable computing are also exploring the use of heterogeneous structures and word-width computational elements. Those designing compiler systems face the task of improving automatic design tools to the point where they may achieve mappings comparable to manual design for even high-performance applications. Within both of these research categories lies the additional topic of run-time reconfiguration. While some work has been done in this field as well, research must continue to attain faster and more efficient reconfiguration. Further study into each of these topics is necessary in order to harness the full potential of reconfigurable computing.

Chapter 3

Reconfigurable Hardware in SoCs

Two different architectural categories have emerged for the use of reconfigurable hardware on a system-on-a-chip (SoC). The first is a reconfigurable subsystem, included as one of the many SoC components, and distinct from other fixed components. In this case, the reconfigurable hardware can be offered as a separate component to an SoC designer. Alternately, fully designed SoCs are marketed which include a reconfigurable logic component. Reconfigurable subsystems are described in section 3.1.

The other architectural category is classified as systems-on-a-programmable-chip (SoPCs). Here, the entire SoC is built from a reconfigurable fabric, with a few fixed resources embedded on the chip. Most SoC functionality is implemented using the reconfigurable logic. SoCs with embedded reconfigurable logic have a higher performance potential because fixed blocks tend to be more efficient than programmable logic configured to perform the same function. However, SoPCs provide a much greater flexibility, allowing a larger percentage of the hardware to be reprogrammed. Also, SoPCs can provide a lower cost solution for custom SoCs because programmable logic

can be configured to implement the required circuitry instead of requiring a chip to be fabricated to customize fixed hardware. SoPCs are discussed further in section 3.2.

3.1 Reconfigurable Subsystems

Many academic reconfigurable architecture are either explicitly intended or could be used for SoC designs. Some of these architectures contain reconfigurable functional units within a host processor [Razdan94, Wittig96, Hauck97]. Others are intended to be coprocessors [Ebeling96, Hauser97, Miyamori98, Goldstein00] for a host microprocessor. Finally, some systems such as Pleiades [Abnous98] actually focus on a reconfigurable interconnection network to connect a microprocessor to other computational units, some of which may also be programmable.

Commercial reconfigurable sub-systems provide a designer with a pre-designed reconfigurable logic structure that can be used as an SoC component. Actel's VariCore blocks are one of the available reconfigurable IP blocks that can be used for this purpose [Actel01]. VariCore designs can be used in densities of 5K to 40K gate equivalents, in 2.5K gate increments. M2000 offers a core based on four-input LUTs—the FleXEOS [M2000-02]. The typical FleXEOS core provides logic resources equivalent to approximately 30K ASIC gates. Reconfigurable sub-systems are also under development from LeopardLogic [LeopardLogic03] and QuickSilver Technology [QuickSilver03], though no architectural details are currently available.

A few of the available reconfigurable IP blocks differ significantly from traditional FPGA structures. For example, instead of using LUT-type structures, the basic building block of Elixent's D-Fabrix is a 4-bit ALU [Elixent02]. Silicon Hive's reconfigurable cores incorporate extremely coarse-grained units such as ALUs and multipliers [SiliconHive03]. Few architectural details are currently available for the Silicon Hive cores. They appear to be an amalgam of reconfigurable hardware and small VLIW-based multiprocessors, but still are intended to be coupled with a host processor on an SoC. eASIC's eASICore differs from the previous reconfigurable cores in terms of the degree of reconfigurability. Its logic structures are reprogrammable, but the routing is mask-programmable [eASIC03]. While this leads to a faster overall structure, post-fabrication flexibility is severely limited.

In some cases, available SoCs already incorporate reconfigurable subsystems. This programmable logic is generally accessed by an on-chip CPU on a communication bus shared with other logic and peripherals [Atmel02, QuickLogic02, Triscend02]. The FPSLIC series of devices from Atmel fall into this category [Atmel02]. This design contains an 8-bit RISC microprocessor, surrounded by microcontroller peripherals and RAM, plus a small amount (5K to 40K gate equivalents) of reconfigurable logic based on their AT40K FPGA design.

The QuickMIPS device from QuickLogic includes significantly more reconfigurable logic than many of the other reconfigurable cores – up to 575K gate equivalents – which can be accessed by the on-chip 32-bit MIPS RISC processor or

peripherals through a bus structure [QuickLogic02]. The Triscend A7 series of configurable SoC also uses a 32-bit RISC processor – the ARM7TDMI [Triscend02]. Devices in this series contain approximately 500-2000 logic cells, each containing a single four-input LUT along with fast carry logic and a D flip-flop. Triscend also markets the Triscend E5 [Triscend03], a customizable 8051/52-compatible microcontroller that includes a small amount of reconfigurable logic. The Cypress PSoC device family falls somewhere between SoPCs and SoCs with reconfigurable components [Cypress03]. The reconfigurable logic is separated from the 8-bit microprocessor at the device core, communicating only through an internal system bus. However, it is the reconfigurable logic that has access to off-chip communication, implementing all major SoC peripherals.

3.2 Systems-on-a-Programmable-Chip (SoPCs)

Frequently, the SoCs marketed by FPGA companies fall into the system-on-a-programmable chip category [Altera01, Altera03b, Xilinx03a, Xilinx03b, Xilinx03c, Xilinx03d]. Since these companies focus on reconfigurable logic, it is natural that they would provide solutions based on hardware flexibility, the primary benefit of reconfigurable logic. A custom SoC can be created relatively easily for low cost, and later be modified without re-fabrication. SoPCs are also useful for prototyping SoC designs to be fabricated.

These SoPCs can be further divided into two additional sub-categories: those that include one or more fixed processors, and reconfigurable devices large enough to implement a full system, optionally including “soft” processor cores. A soft processor core is a processor created from the programmable logic within an FPGA, rather than directly into silicon. This type of processor will normally run slower than a fixed processor due to the overhead of programmable logic. However, using soft cores provides a greater deal of flexibility to instantiate the quantity and style of processor cores best suited to the application. Xilinx and Altera both provide soft processor cores as well as SoPCs containing hard processor cores [Altera01, Altera03b, Xilinx03a, Xilinx03b, Xilinx03c, Xilinx03d].

The Xilinx Virtex-II Pro is currently available with up to four PowerPC 405 RISC processor blocks [Xilinx03a]. Xilinx also provides two different soft cores: PicoBlaze, an 8-bit microcontroller [Xilinx03b, Xilinx03c], and MicroBlaze, a 32-bit RISC processor [Xilinx03d]. These soft cores can be implemented in the Spartan-II, Virtex, Virtex-II, and Virtex-II Pro series of FPGAs to create a full SoC using programmable logic.

Altera targets a number of its FPGA models towards SoPC use. The Excalibur devices each contain a single ARM922T RISC processor in a logic array based on the Apex 20KE device [Altera01]. Also, Altera’s Nios soft processor core can be implemented in a variety of Altera FPGAs, including the FLEX, APEX 20K, APEXII,

Mercury, and Cyclone devices [Altera01, Altera03b], and is available in both 16-bit and 32-bit versions.

Chapter 4

Research Framework

Current efforts in the Totem Project for automatic generation focus on coarse-grained architectures suitable for compute-intensive application domains such as digital signal processing, compression, and encryption. The RaPiD architecture [Ebeiling96, Cronquist99a] is presently used as a guideline for the generated architectures due to its coarse granularity, one-dimensional routing structure, and compiler (described further in the next section). Coarse-grained units match the coarse-grained computations currently targeted. The one-dimensional structure is efficient for many DSP applications, but also simplifies the architecture generation process significantly. Future work in the Totem Project will include the two-dimensional case. Finally, a compiler [Cronquist98] for this system is already in place, which aids in the development of application circuits for Totem.

This chapter begins with a description of the RaPiD architecture, emphasizing the datapath architecture, but also providing a brief discussion of the control structures and the compiler. Next, a synopsis of the high-level architectural design work (the subject of this thesis) and its role in the Totem Project is given, followed by a short discussion of

the transistor-level layout and the place and route tools used in the Totem Project. Finally, the area comparison methods that will be used to evaluate the Totem architectures will be discussed.

4.1 RaPiD

The disparity between the coarse-grained nature of many computations, such as those needed for DSP, and the fine-grained nature of traditional FPGAs leads to inefficiencies in implementations. The RaPiD system [Cronquist99a], addresses this problem by using a very coarse-grained structure. The term “RaPiD” actually refers to a style of reconfigurable architecture, not a particular architecture [Ebeling96]. This style of architecture has specialized computational elements such as ALUs, RAM units, and multipliers, each operating on full words of data. The components are then arranged along a one-dimensional axis, and connected by word-width routing tracks. These architectures are heavily pipelined, and perform very fast computations on large amounts of data. While the routing flexibility is somewhat lower, the routing architecture complexity is also lower, reducing routing area as well as simplifying the routing process.

A number of RaPiD implementations exist, including RaPiD-I [Ebeling96] and RaPiD-Benchmark [Cronquist99a], which is the more recent implementation, and is frequently called “RaPiD” for simplicity. From this point forward, whenever the “RaPiD architecture” is discussed, this more recent design is being referenced. The next few sections describe RaPiD’s datapath, control architecture, and compiler in more detail.

4.1.1 Datapath Architecture

As mentioned previously, RaPiD is composed of coarse-grained computational units arranged along a one-dimensional axis and connected through a series of word-width routing tracks. The logic units are grouped into repeating “cells”, as shown in Figure 4.1. The full architecture is formed by tiling cells horizontally to form a longer architecture.

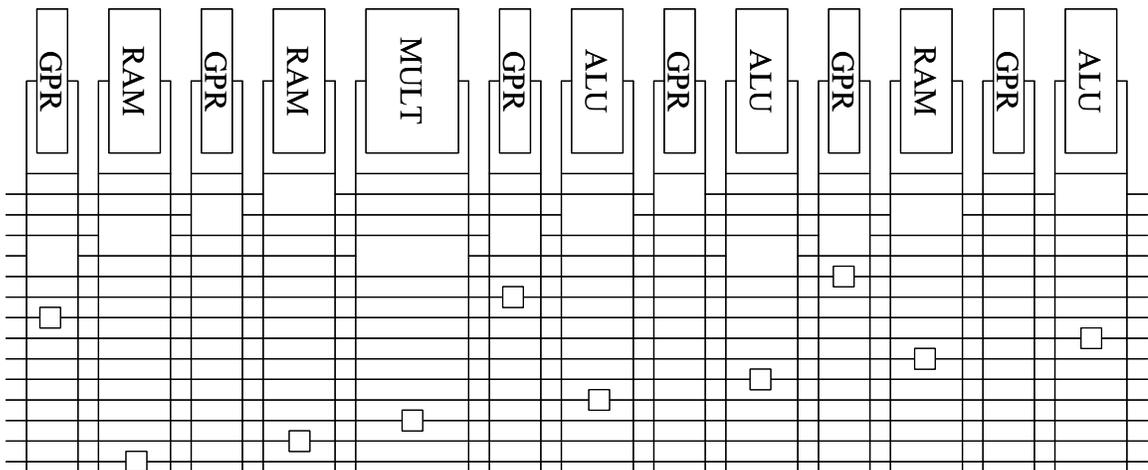


Figure 4.1: A single cell from the RaPiD architecture [Cronquist99a, Scott01]. A full architecture is composed of some multiple of these cells laid end-to-end.

The logic units within the cells operate on full words of data, and include 16-bit ALUs, 16x16 multipliers, 16-bit wide RAM units, and 16-bit registers. Each component contains a multiplexer on each of its inputs that choose between the signals of each routing track. Each component also has a demultiplexer on each of the outputs that allow

the unit to directly output to any of the routing tracks. Inputs are on the left side of a unit, while the outputs are on the right side of the unit.

The routing architecture itself is a one-dimensional segmented design, where each track is composed of as many wires as the word width of the architecture. Full words of data are therefore communicated between the computational units of the architecture. There are 14 routing tracks, plus one additional routing track that only contains "feedback" wires. These feedback wires are only permitted to route an output of a unit back to one or more of the inputs of a unit. Additionally, a word-sized "zero" is also provided as a possible input to each multiplexer. The top five routing tracks are local routing tracks, including the special feedback track. These tracks contain short wires for fast short-distance communication. The bottom ten tracks provide longer distance routing. The small squares on these routing tracks are bus connectors, which allow the wire segments to be optionally connected to form longer wires. Additionally, the bus connectors provide optional pipeline delays to mitigate the delay added through the use of longer wires and routing switches.

4.1.2 Control Architecture

The control architecture for RaPiD is a hybrid of configuration points and dynamically generated signals [Cronquist99a]. Hardware-programmed configuration points require less area than hardware to generate dynamic control signals. However, using only programmed control points reduces the ability to model complex circuit

behavior that changes during execution. Therefore, RaPiD uses a combination of these two methods, with approximately 75% of control points field-programmable, and 25% generated dynamically. These types of control are referred to as “hard” and “soft”, respectively, though it should be noted that the hard control is not permanently fixed, but is instead fixed for any given configuration programmed into RaPiD. While the demultiplexers on the units are controlled by hard bits, the multiplexers use soft control. At different points in execution, a unit may require different signals on an input. Soft control bits enable the multiplexer to select different signals depending on the values of the control.

Soft control bits originate with a number of small parallel instruction generators (four in the current RaPiD implementation), which execute “microprograms” (a series of simple instructions that can include loops) in order to generate instructions for the control path. These instructions are sent through a control path parallel to the datapath of RaPiD, which includes programmable LUTs. Different locations in the RaPiD array can therefore decode the instructions into actual control signals (soft control) in different ways, allowing for very complex control to be implemented.

4.1.3 RaPiD-C Compiler

Implementing complex applications on the RaPiD architecture is facilitated through the use of a special compiler [Cronquist98] that converts application code in a C-like format into circuit netlists. These circuit netlist files include all datapath and control

information (static and dynamic) to allow the circuit to operate on a RaPiD-like architecture. Netlists are somewhat independent of the actual hardware implementation, as they only describe the needs of the circuits, not an actual allocation of resources to be used. The allocation of resources is performed by a place and route tool, described later.

The core of a RaPiD-C program involves one or more `for` loops. A special type of loop indicates what operations should occur in parallel (spatially), through the use of a special inner “Datapath” loop². The iterations of this special loop are all executed simultaneously. The number of iterations is user-specified by setting the number of “stages” at the beginning of the program, where each stage corresponds to a parallel iteration of the loop. For cases where different stages of the circuit perform different tasks, such as initialization/finalization at the edge stages, conditional statements are permitted on the reserved variable `s`, whose value in each stage is that particular stage’s index. This language structure provides a powerful method to specify complex pipelined and parallel application circuits.

4.2 Totem Project

The goal of the Totem Project is to provide a complete automatic path for the creation of custom reconfigurable hardware, targeted for use in systems-on-a-chip

² This syntax is slightly different from that appearing in the cited document [Cronquist98], as the format of the RaPiD-C language has evolved over time, and a more recent description of the language has not been published as of the completion of this thesis.

(SoCs). There are three primary components of the project. The first is the high-level architecture generation, which determines the resource requirements and how those resources should be arranged. The second component is the VLSI layout generator, which takes a description of the architecture from the high-level architecture generator and translates it into actual transistors and layout masks. The final module is the place and route tool that implements circuit netlists on the generated reconfigurable hardware.

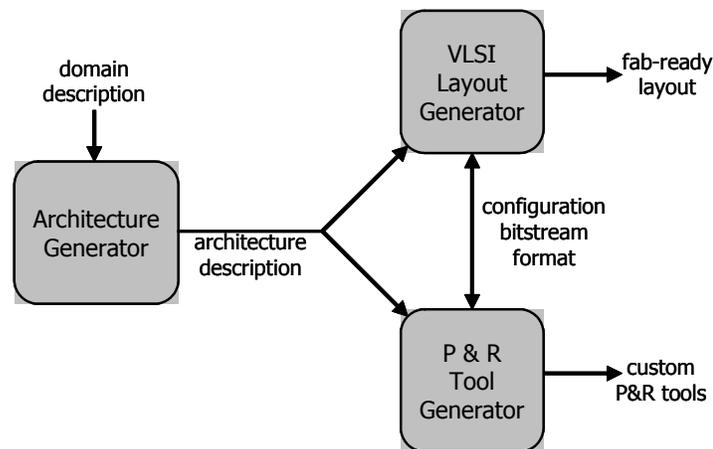


Figure 4.2: The three major components of the Totem Project

The communication between these three components is presented in Figure 4.2. The architecture generator sends a description of the architecture to both of the other tools. The layout generator and the place and route tool generator negotiate the configuration bitstream format. The layout generator then creates the transistor-level layout of the architecture described by the high-level architecture generator. Meanwhile, the place and route tool reads in the architecture description and a set of circuits, and generates the appropriate configurations to implement those circuits on the given

architecture. These three pieces of the Totem Project are discussed further in the following sections.

4.2.1 High-Level Architecture Design

The high-level architecture generation is the subject of this thesis, and is described in depth in the next chapters. Essentially, the architecture generator takes as input a description of the targeted application domain, and outputs a description of the reconfigurable hardware for that domain. Currently, the domain is specified by the user, who provides a set of RaPiD netlists or a set of user-specified characteristics. From this domain information, a datapath is constructed to execute circuits of the types provided. This work focuses on the datapath generation—while control structures will also be necessary to provide full functionality, many of the techniques used in the datapath generation are expected to extend to the control path.

It should also be noted that this work does not yet account for pipelining within the application circuits when a register only has a single input. In these cases, at least some of this pipelining could be implemented by bus connectors on the routing tracks. However, it has not yet been determined how to judge the number of additional data registers that will be required in the datapath structures in order to best implement these types of netlists, and therefore this is an area of future work.

During datapath construction, the netlists or characteristics are analyzed to determine the number and types of computational units required in the architecture. Once

this information is determined, the logic units are created and arranged along the one-dimensional axis so as to minimize area and delay (with area currently the primary goal). These techniques are presented in Chapter 5.

The routing structures that connect the computational components are generated next. Two different styles of routing architectures have been investigated. The first results in highly optimized architectures—ASIC-like, but very inflexible. These architectures are referred to as “configurable ASICs”, or cASICs. The second creates architectures in the RaPiD style, with the amounts and types of routing resources dependent on the communication needs of the different input circuits. Chapter 6 discusses architecture generation in the cASIC style, while Chapter 7 details the RaPiD-style routing generation methods.

4.2.2 Physical Layout

Several techniques are currently under investigation for generating a VLSI layout from a Verilog description of a Totem architecture. These techniques include standard cells, template reduction, and circuit generators. The standard cell method is a traditional layout technique, but with a slightly modified set of standard cells optimized for reconfigurable layouts, such as such as multiplexers, demultiplexers, and D flip-flops [Phillips02].

The next layout technique is template reduction, and is based on a subtractive technique. Instead of adding logic and routing resources as needed, this technique begins

with a manually designed superset architecture and removes unneeded resources. Template architectures would be created for sets of domains. When creating a custom architecture, the closest template is selected. The place and route tool then implements the needed netlists onto the template, and removes unused logic and routing resources. The more similar the needed architecture is to the template, the more the final architecture benefits from the manual layout of the template.

Finally, circuit generators allow for highly customizable computational units. This method uses macro blocks to implement the logic units of the architecture, and calculates where metal wires should be added for the communication structure. These macro blocks can be as simple as a library of defined structures, such as a 16-bit register or ALU. However, a more sophisticated implementation will allow for customizable bitwidth and functionality (i.e., an adder instead of a full ALU). These generators use the regularity within many computational components to achieve efficient layouts.

Together these three layout methods will form the essential path from the architecture description to a working VLSI layout. Architectures close in form to an available tileable template will be implemented using template reduction for a more efficient layout than could be created using standard cells. Next, architectures requiring a high degree of customization will use circuit generators, provided that customization can be accomplished using the available set of functions. Lastly, architectures that do not fit well with either of the previous methods can be implemented using standard cells.

4.2.3 Place and Route

The Totem Project requires a place and route tool for two primary reasons. First, the tool finds a final implementation of the specification netlists on the generated architectures. Since the architecture generator uses simplistic methods to find an implementation, a dedicated place and route tool is likely to find a superior solution. Second, the tool is needed to implement circuits not included in the original architectural specification, which can occur if circuits are designed after the hardware has been fabricated. This tool inputs a simplified compiled RaPiD netlist (which includes only the datapath information with pipelining registers removed) and a Verilog architecture description, and outputs a bitstream that can be used to configure the generated architecture. Note that the bitstream format may require information from the layout tool to obtain the correct ordering of the programming bits.

The Totem place and route tool [Compton02d, Sharma02] performs two operations. First, a netlist has its logical instances placed into physical components in the architecture. The placement phase is based on the simulated annealing algorithm [Sechen88]. The cost function for the simulated annealing operation is based on a combination of the maximum (*max*) and average (*avg*) signal cross-section throughout the architecture [Sharma02].

Second, the signals between the instances are mapped to actual wires in the hardware that connect the instances' physical units. The router uses an iterative, negotiation-based routing algorithm that is a modified version of the Pathfinder algorithm

[McMurchie95]. A route is found for each signal onto the routing architecture. Initially, signals are permitted to share resources. The cost of using each individual resource is determined by the number of signals sharing that resource, and is updated after each iteration. Signals are rerouted each iteration, but with previously shared resources considered increasingly more expensive to use. Eventually, for each shared resource, all but one signal are forced to find alternate routes if possible.

4.3 Testing Framework

Eight different applications (each composed of two or more netlists) were used to compare the area results of the Totem architectures to a number of existing implementation techniques, including standard cell, FPGA, and RaPiD techniques. These applications, along with their member netlists, are listed in Table 4.1. Five of these are real applications used for radar, OFDM, digital camera, speech recognition, and image processing. The remaining three applications are sets of related netlists, such as a collection of different FIR filters.

Table 4.1: Eight applications used to test Totem architectures, each containing two or more distinct netlists. FIR, Matrix, and Sort are collections of similar netlists, while the others are actual applications.

Application	Member Netlists
Radar	decnsr, fft16_2nd, psd
OFDM	sync, fft64
Camera	color_interp, img_filt, med_filt
Speech	log32, fft32, 1d_dct40
FIR	firms, firms2, firms3, firmsymeven, firm_1st, firm_2nd
Matrix	matmult, matmult4, matmult_bit, limited, limited2
Sort	sort_g, sort_rb, sort_2d_g, sort_2d_rb
Image	med_filt, matmult, firm_2nd, fft16_2nd, 1d_dct40

4.3.1 Standard Cells

The standard cell layouts of the netlists (converted from RaPiD netlist format to Verilog) were created using Cadence in a TSMC 0.18 μ m process with 6 metal layers. Generally, the total area for an application set is the sum of the areas required for the netlists. However, for the application sets which are a collection of very similar netlists (FIR, Matrix, and Sort from Table 4.1), this assumption is likely to be incorrect. Therefore, to err on the side of caution for these particular cases, the maximum area required by any one member netlist is instead used, under the assumption that a small amount of additional control circuitry may allow all member netlists to use the same hardware. I/O area is not included, as I/O area is also not measured for the Totem architectures. The standard cell areas for individual netlists as well as the application sets are given in Table 4.2.

Table 4.2: The areas of the eight different applications from Table 4.1 implemented using standard cells in a 0.18 μm process. In most cases, the application area is the sum of the member netlist areas. However, the FIR, Matrix and Sort applications are collections of very similar netlists, so to better estimate the standard cell layout area, the maximum member netlist area is instead used.

Application	Netlists	Netlist Area (mm ²)	Application Area (mm ²)
Radar	decnsr	0.160	4.101
	fft16_2nd	3.073	
	psd	0.867	
OFDM	sync	2.996	9.168
	fft64	6.172	
Camera	color_interp	3.640	7.268
	img_filt	2.858	
	med_filt	0.769	
Speech	log32	22.070	26.523
	fft32	4.050	
	1d_dct40	0.404	
FIR	firms	2.668	2.846
	firms2	2.668	
	firms3	2.674	
	firsyeven	2.846	
	firtm_1st	1.302	
	firtm_2nd	1.214	
Matrix	matmult	1.383	1.785
	matmult4	1.414	
	matmult_bit	1.348	
	limited	1.785	
	limited2	0.517	
Sort	sort_g	1.541	1.541
	sort_rb	1.310	
	sort_2d_g	1.152	
	sort_2d_rb	1.152	
Image	med_filt	0.769	6.843
	matmult	1.383	
	firtm_2nd	1.214	
	fft16_2nd	3.073	
	1d_dct40	0.404	

4.3.2 FPGA

The FPGA solution is based on the Xilinx VirtexII FPGA, which uses a 0.15 μm 8-metal-layer process, with transistors at 0.12 μm [Xilinx02]. In particular, the die area was obtained for an XC2V1000 device [Chipworks02]. This FPGA contains not only

LUT-based logic (“slices”), but also embedded RAM and multiplier units, in a proportion of 128 slices : 1 multiplier : 1 RAM. This proportion of resources is used as a tileable atomic unit when determining required FPGA area for the designs, as manually-designed FPGA cores for SoCs are unlikely to be very customizable except in terms of the quantity of total tileable resources.

The area of an individual tile, which corresponds to approximately 25K system “gates” of logic, was computed (using a photograph of the die) to be 1.141mm^2 . This area was then scaled to a $0.18\mu\text{m}$ process by multiplying by $(.15/.18)^2$ to yield a final tile size of 1.643mm^2 to compare all solutions using the same fabrication process. The Verilog files created from individual netlists were placed and routed onto a VirtexII chip, and the number of tiles required for the applications were measured. In this case, the total area required by an application is the maximum of the areas required by its member netlists, as the hardware resources are reusable. These areas are given in Table 4.3.

Table 4.3: The FPGA areas of the eight different applications from Table 4.1. The resource usage and number of atomic tiles is given for each netlist, and the number of tiles and resulting area (converted to a 0.18 μ m process) is given for each application.

Application	Netlists	Slices	Mults	RAMs	Netlist Tiles	App. Tiles	Application Area (mm ²)
Radar	decnsr	112	0	0	1	12	19.719
	fft16_2nd	489	12	12	12		
	psd	172	4	0	4		
OFDM	sync	2415	2	16	19	36	59.157
	fft64	2442	8	36	36		
Camera	color_interp	512	14	6	14	14	23.006
	img_filt	1017	0	13	13		
	med_filt	29	0	4	4		
Speech	log32	1304	48	0	48	48	78.877
	fft32	626	16	16	16		
	1d_dct40	29	1	0	1		
FIR	firms	368	16	0	16	16	26.292
	firms2	368	16	0	16		
	firms3	360	16	0	16		
	firmsyeven	585	16	0	16		
	firms_1st	211	4	11	11		
	firms_2nd	280	4	8	8		
Matrix	matmult	312	4	12	12	12	19.719
	matmult4	344	4	12	12		
	matmult_bit	956	4	12	12		
	limited	284	8	8	8		
	limited2	131	2	0	2		
Sort	sort_g	1049	0	16	16	16	26.292
	sort_rb	1019	0	9	9		
	sort_2d_g	822	0	12	12		
	sort_2d_rb	781	0	8	8		
Image	med_filt	29	0	4	4	12	19.719
	matmult	312	4	12	12		
	firms_2nd	280	4	8	8		
	fft16_2nd	489	12	12	12		
	1d_dct40	29	1	0	1		

4.3.3 RaPiD

The area required to implement the applications on a static RaPiD architecture was also calculated [Cronquist99a]. The RaPiD results represent a partially-customized FPGA solution. The RaPiD reconfigurable architecture was designed for the types of

netlists used in this testing, and contains specialized coarse-grained computational units used by those netlists. The number of RaPiD cells can be varied, but the resource mix and routing design within the cell is fixed.

To find the area for each application, the minimum number of RaPiD cells needed to meet the logic requirements of the application was calculated. The application's netlists were then placed and routed onto the architecture to verify that enough routing resources were present. If not, and the routing failed, the number of cells was increased by one until either all of the application's netlists could successfully place and route, or place and route still failed with 20% more cells than the application logic required. Table 4.4 lists the results of these tests.

Manual layouts of each of the units and routing structures were created in a TSMC 0.18 μ m process with 5 metal layers. The logic area is simply the sum of the areas of the logic units in the architecture. Routing area is the sum of the areas of the multiplexers, demultiplexers, and bus connectors (segmentation points) in the architecture. Routing tracks are directly over the logic units in a higher layer of metal, and are therefore not counted as contributing to the area. In some cases, the RaPiD architecture did not have sufficient routing resources to implement a circuit. The RaPiD cell would have to be manually redesigned to fit these netlists. This illustrates one of the primary benefits of an automatic architecture generator – provided enough die area is allocated, a solution can always be created.

Table 4.4: The RaPiD areas of the eight different applications from Table 4.1. The area for each application is based on the minimum number of cells required to successfully place and route all of the netlists in the given applications. The number of cells needed by each netlist is also given.

Application	Netlists	Netlist Cells	Netlist Area (mm ²)	App. Cells	Logic Area (mm ²)	Routing Area (mm ²)	Application Area (mm ²)
Radar	decnsr	2	0.833	12	2.838	2.158	4.996
	fft16_2nd	12	4.996				
	psd	6	2.498				
OFDM	sync	---	---	---	---	---	---
	fft64	---	---				
Camera	color_interp	---	---	---	---	---	---
	img_filt	19	7.910				
	med_filt	---	---				
Speech	log32	192	79.937	192	45.401	34.536	79.937
	fft32	16	6.661				
	1d_dct40	3	1.249				
FIR	firms	16	6.661	16	3.783	2.878	6.661
	firms2	16	6.661				
	firms3	16	6.661				
	firmsyeven	16	6.661				
	firtm_1st	4	1.665				
	firtm_2nd	4	1.665				
Matrix	matmult	4	1.665	8	1.892	1.439	3.331
	matmult4	4	1.665				
	matmult_bit	4	1.665				
	limited	8	3.331				
	limited2	2	0.833				
Sort	sort_g	11	4.580	21	4.966	3.777	8.743
	sort_rb	12	4.996				
	sort_2d_g	8	3.331				
	sort_2d_rb	8	3.331				
Image	med_filt	---	---	---	---	---	---
	matmult	4	1.665				
	firtm_2nd	4	1.665				
	fft16_2nd	12	4.996				
	1d_dct40	3	1.249				

4.3.4 Relative Areas

Previously, FPGAs produced solutions approximately 50x larger than standard cell implementations of a given circuit [Stone96]. However, the results of the area tests in this chapter indicate that progress has been made to reduce the overhead of

reconfigurable hardware. Table 4.5 lists the areas of the available netlists for each of the implementation methods discussed. According to this data, FPGA implementations using the VirtexII FPGA are just under 10x larger than the standard cell implementation. Some of the possible reasons for this 5x improvement in the FPGA area include the use of two more metal layers in the VirtexII compared to these standard cell implementation, and the embedded multipliers and RAM units in the VirtexII, which were absent from the FPGA design used in the previous study.

The margin between standard cells and reconfigurable hardware shrinks further when application areas are compared. The FPGA and RaPiD resources can be reused between the different netlists in an application. However, a separate circuit layout must be created for each netlist in an application using a standard cell technique³. While the FPGA areas are on average nearly 10x larger than standard cell for individual circuits, these areas are on average just over 7x larger than standard cell implementations once the FPGA hardware is reused for multiple netlists within an application.

³ Note that as discussed previously, three of the “applications” are collections of similar netlists. To err on the side of caution, it is assumed that the actual standard cell layout for those applications will be approximately the size of the largest netlist in the application, and that only a small amount of support circuitry would be necessary to allow the layout to perform the other functions as well.

Table 4.5: The areas of all of the netlists from Table 4.1 using each of the implementation methods, normalized to the standard cell area.

Netlist	Std Cell Area (mm ²)	FPGA Area (mm ²)	RaPiD Area (mm ²)
1d_dct40	1.00	4.07	3.09
color_interp	1.00	6.32	---
decnsr	1.00	10.25	5.20
fft16_2nd	1.00	6.42	1.63
fft32	1.00	6.49	1.64
fft64	1.00	9.58	---
firms	1.00	9.86	2.50
firms2	1.00	9.86	2.50
firms3	1.00	9.83	2.49
firsyeven	1.00	9.24	2.34
firtm_1st	1.00	13.88	1.28
firtm_2nd	1.00	10.83	1.37
img_filt	1.00	7.47	2.77
limited	1.00	7.36	1.87
limited2	1.00	6.36	1.61
log32	1.00	3.57	3.62
matmult	1.00	14.26	1.20
matmult4	1.00	13.95	1.18
matmult_bit	1.00	14.62	1.24
med_filt	1.00	8.54	---
psd	1.00	7.58	2.88
sort_g	1.00	17.06	2.97
sort_rb	1.00	11.29	3.81
sort_2d_g	1.00	17.11	2.89
sort_2d_rb	1.00	12.92	3.27
sync	1.00	10.42	---
AVERAGE	1.00	9.97	2.42

Table 4.6: The areas of each of the applications from Table 4.1 using each of the implementation methods, normalized to the standard cell area.

Application	Std Cell Area (mm ²)	FPGA Area (mm ²)	RaPiD Area (mm ²)
Radar	1.00	4.81	1.22
OFDM	1.00	6.45	---
Camera	1.00	3.17	---
Speech	1.00	2.97	3.01
FIR	1.00	9.24	2.34
Matrix	1.00	11.04	1.87
Sort	1.00	17.06	3.24
Image	1.00	2.88	---
AVERAGE	1.00	7.20	2.34

Chapter 5

Logic Generation

As stated previously, this work encompasses two categories of customized reconfigurable hardware generation: highly customized near-ASIC (cASIC) architectures and more flexible RaPiD-like designs. In both cases, it is the routing architecture style that differs, while the logic is created using essentially the same methods. The logic architecture is created in two steps: first the number of each type of coarse-grained computational unit is determined; then these units are ordered along the horizontal axis. Both of these operations are based on the netlists provided by the user as a description of the domain for which the reconfigurable architecture is to be created. This architecture contains the logic resources needed to implement each of the netlists one at a time. Performing a reconfiguration of the hardware replaces the current netlist with another.

5.1 Type and Quantity of Units

The generation of logic structures for cASIC architectures involves first determining the type and quantity of functional units required to implement the given netlists. Because the ability to reuse hardware is a key feature of reconfigurable

computing, maximum hardware reuse between netlists is forced. The minimum number of total logic units is chosen such that any one of the netlists given as part of the architectural specification can operate in its entirety. In other words, unit use within a netlist is not modified or rescheduled. Therefore, if netlist A uses 12 multipliers and 16 ALUs, while netlist B uses 4 multipliers and 24 ALUs, a cASIC architecture designed for these two netlists would have 12 multipliers and 24 ALUs. Note that for flexible architecture generation, the designer can specify to include additional logic units beyond the minimum.

5.2 Binding vs. Physical Moves

After the unit quantities and types have been selected, they must be arranged along the horizontal axis. A good ordering will ensure that the number of signals passing through any one vertical cut of the architecture is kept low, which reduces the area consumed by the routing structures. Similarly, units communicating with one another should be located in close proximity to reduce the delay on the wires between them. Therefore, the best physical ordering of the units depends on the communication between them. The communication needs between physical units, however, depend on how the netlists are implemented on that hardware.

Before discussing this issue further, some terminology must be defined. The architectural *components* represent physical structures to be implemented in silicon. These differ from netlist *instances*, which are implemented by the physical components.

A netlist instance represents a “need” for a given type of computation at a given point of the circuit. In traditional FPGAs, the LUTs are the physical components, while the netlist instances are low-level gates or small logic functions. In the Totem Project, coarser-grained architectures and netlists are currently used. For example, a multiply-accumulate netlist contains a multiplier instance followed by adder and register instances. These instances must be implemented by the appropriate type of physical components in the hardware.

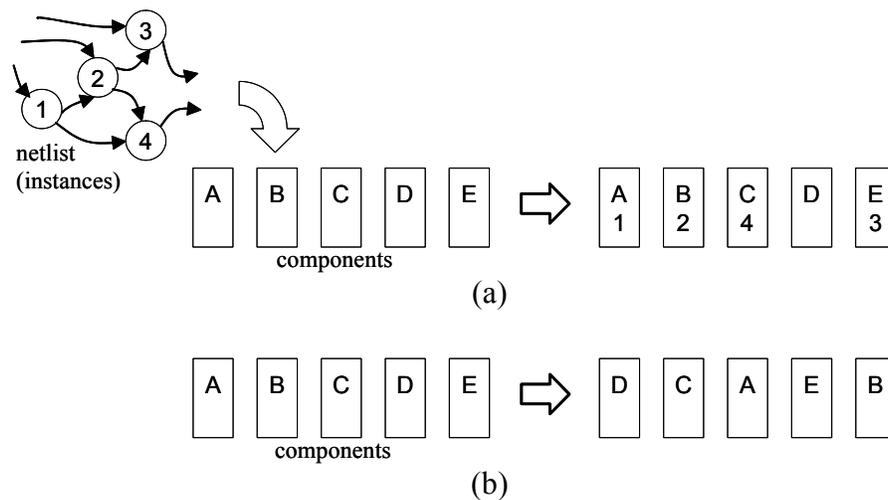


Figure 5.1: (a) *Binding* assigns instances of a netlist to physical components. (b) *Physical moves* reposition the physical components themselves.

There may be multiple units appropriate for a circuit instance, in which case the instance must be matched to a specific physical unit. When using traditional FPGAs, this matching is referred to as placement or binding. For this work, the terms *binding* or *mapping* are used to describe the process of matching an instance to a component. A *physical move* describes the act of assigning a physical location to a physical component.

Figure 5.1 illustrates the difference between binding and placement. Using this terminology, traditional synthesis for FPGAs requires only bindings, whereas synthesis for standard cells involves only physical moves.

Reconfigurable architecture generation is a unique situation in which both binding and physical moves must be considered. The locations of the physical units must be known in order to find the best binding, and the binding must be known to find the best physical moves. Since these processes are inter-related, both binding and physical moves are performed simultaneously in this work. The term *placement* is used here to refer to the combined process of determining a binding of netlists to units and physical locations for the units. The next section describes the algorithm used to find the final placement. An example will be presented that demonstrates the logic generation and placement for an architecture created from the two netlists in Figure 5.2.

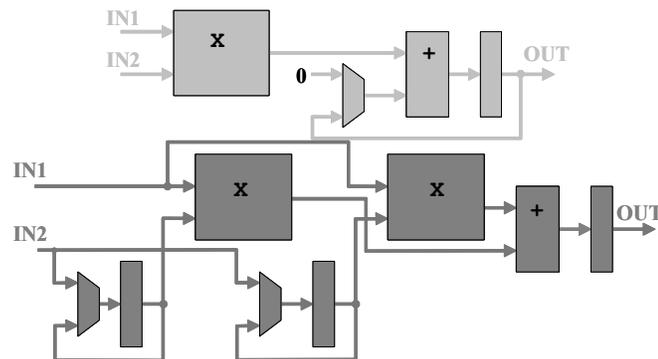


Figure 5.2: Two different example netlists that could be used in architecture generation. The light netlist performs a multiply-accumulate (MAC), while the dark netlist is a 2-tap FIR filter. These two netlists are used in the example placement process given in the next few figures.

5.3 Adapting Simulated Annealing

Simulated annealing [Sechen88] is an algorithm commonly used in FPGA placement (binding) to assign netlist instances to physical computation units, and standard cell placement to determine locations for actual physical cells. This algorithm operates by taking a random initial placement of elements, and repeatedly attempts to move the location of a randomly selected element. The move is accepted if it improves the overall cost of the placement. To avoid settling in a local minima of the placement space, moves that do not improve the cost of the placement are sometimes accepted. The probability of accepting a non-improving move is governed by the current “temperature”. At the beginning of the algorithm, the temperature is high, allowing a large proportion of bad moves to be accepted. As the algorithm progresses, the temperature decreases, and therefore the probability of accepting a bad move also decreases. At the end of the algorithm almost no bad moves are permitted.

For reconfigurable architecture generation, the simulated annealing algorithm must be adapted, changing what constitutes a “move”. In FPGA placement a move is changing which physical structure implements a specific portion of the netlist logic (rebinding), and in ASIC placement a move involves giving a new physical position to a standard cell. For this work, a move can be either of these two possibilities – either rebinding a netlist computational instance from one physical unit to another compatible

physical unit, or changing the position (ordering) along the 1D axis of a computational component.

The instances of each netlist are arbitrarily assigned initial bindings to physical components, which are ordered arbitrarily along the 1D axis. An example initial placement created for the two netlists presented in Figure 5.2 appears in Figure 5.3. Like simulated annealing, a series of moves are used to improve the placement/binding. The probability of attempting a re-binding versus a physical component movement is equal to the fraction of total “elements” that are instances instead of physical components. A physical move is shown for the example architecture in Figure 5.4, and a rebinding in Figure 5.5.

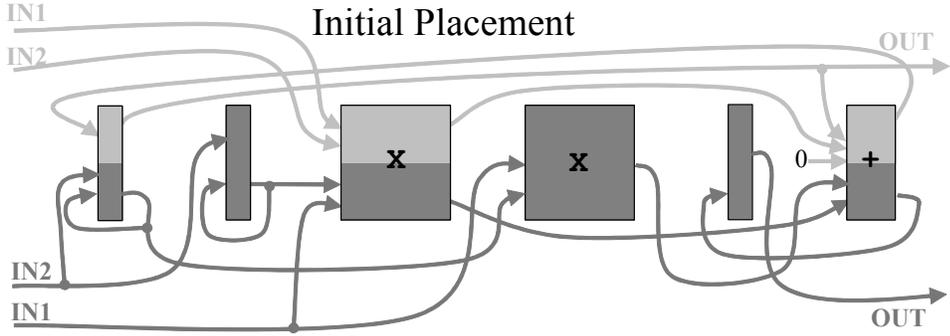


Figure 5.3: The initial physical placement and bindings of an architecture created for the netlists of Figure 5.2. Color shading indicates component use by the netlists. For example, when the light netlist is in operation, only the left multiplier is used. When the dark netlist is in operation, both multipliers are used. The architecture can be reconfigured to switch between the light and dark netlists.

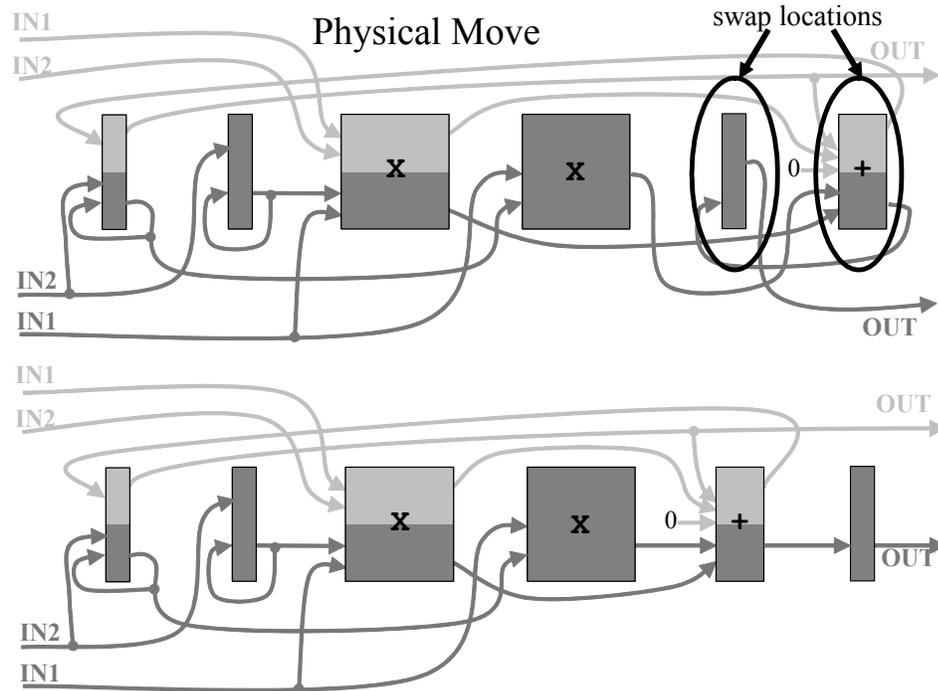


Figure 5.4: A physical move performed during the placement operation. Top: Before. Bottom: After.

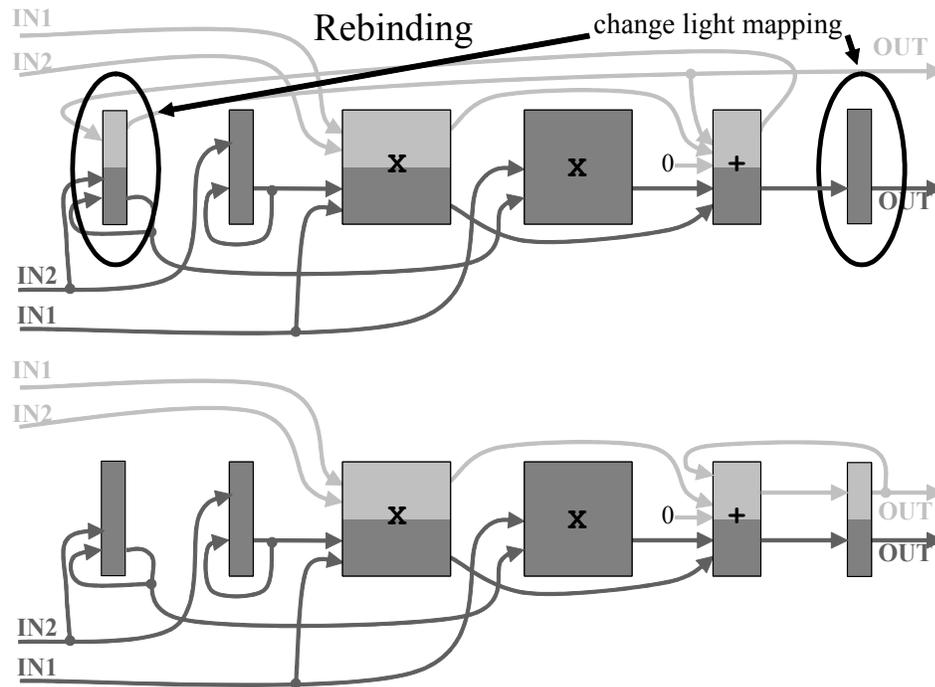


Figure 5.5: A rebinding performed during the placement operation. Top: Before. Bottom: After.

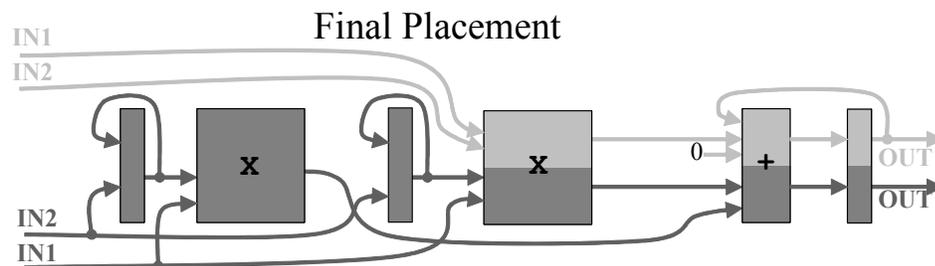


Figure 5.6: The final placement of the architecture created for the netlists from Figure 5.2 after a series of moves such as those illustrated in Figure 5.4 and Figure 5.5. The signal cross-section has been greatly reduced from the initial placement shown in Figure 5.3.

The cost metric is based on the cross-section of signals communicating between the bound instances. At each physical unit location, the cross-section of signals for each netlist is determined. The maximum across the netlists becomes the overall cross-section value at that point. After the cross-section value is calculated for each location, the

values are squared, then summed across the locations to yield the overall cost value. By squaring the values before summing across positions, areas with a high cross-section are heavily penalized. The goal in reducing these cross-sections is primarily to minimize the area of the routing structure that will be created, because a larger cross-section can lead to a taller architecture. A secondary goal is to decrease the delay of the nets, because the longer (and slower) a wire is, the more likely it is to share a span with other wires and contribute to a larger cross-section.

The guidelines presented for VPR [Betz97], a place and route tool for FPGAs, govern the initial temperature calculation, number of moves per temperature, and cooling schedule. These values are based on N_{blocks} , the number of “blocks” in a netlist. Since both netlist instances and physical components are being used, N_{blocks} is calculated as the sum of the instances in each netlist provided plus the number of components created, as shown in Equation 5.1. The initial temperature is calculated by performing N_{blocks} moves, and finding the standard deviation (*stddev*) of the cost of these N_{blocks} bindings/placements. The initial temperature calculation is given in Equation 5.2. Equation 5.3 presents the VPR calculation of the number of moves (*nMoves*) performed at each temperature. This calculation is not changed apart from the calculation of N_{blocks} as previously described.

$$\text{Equation 5.1: } N_{blocks} = N_{components} + \sum_{i=0}^{\# \text{ netlists}} N_{instances}_i$$

Equation 5.2: $T_{init} = 20 * stddev$ [Betz97]

Equation 5.3: $nMoves = 10 * (N_{blocks})^{1.33}$ [Betz97]

Finally, the cooling schedule specified by VPR is also used. The new temperature T_{new} is calculated according to the percentage of moves that were accepted (R_{accept}) at the old temperature T_{old} . Table 5.1 details the cooling schedule, indicating how the temperature should be updated according to R_{accept} .

Table 5.1: The calculation of the new temperature T_{new} based on the percentage of moves accepted, R_{accept} . This cooling schedule is used by VPR [Betz97].

$R_{accept} > .96$	$T_{new} = .5 * T_{old}$
$.8 < R_{accept} \leq .96$	$T_{new} = .9 * T_{old}$
$.15 < R_{accept} \leq .8$	$T_{new} = .95 * T_{old}$
$R_{accept} \leq .15$	$T_{new} = .8 * T_{old}$

After determining the logic structure and placement, the components must be connected together with some sort of routing structure. As will be discussed in Chapter 6, the routing can be very specialized, only providing the connections that the input circuit set requires. This leads to a highly optimized architecture with very low flexibility. The other option, discussed in Chapter 7, is to create a more generic, flexible, routing structure. Some specialization is retained, yet sufficient flexibility exists to implement circuits beyond the specification set.

Chapter 6

Configurable ASICs

While the flexibility of traditional FPGA structures is one of their greatest assets, it is also one of the largest drawbacks—greater flexibility leads to greater area, delay, and power overheads. Creating customized reconfigurable architectures presents the opportunity to greatly reduce these overheads by discarding excess flexibility. This chapter discusses taking this idea to the extreme end of the spectrum – removing *all* unneeded flexibility to produce an architecture as ASIC-like as possible. This style of architecture is referred to as “configurable ASIC”, or cASIC.

Like RaPiD [Cronquist99a], cASIC architectures are very coarse-grained, consisting of optimized components such as multipliers and adders. Unlike RaPiD, cASICs do not have a highly flexible interconnection network—the only wires and multiplexers available are those which are required by the netlists. This is because cASICs are designed for a specific set of netlists, and are not intended to implement netlists beyond the specification. Hardware resources are still controlled by configuration bits, though there are much fewer present. These configuration bits allow for hardware

reuse among the specification netlists. Each netlist in the specification is implemented in turn by programming these bits with the appropriate set of values.

The cASIC architecture generation occurs in two distinct phases. In the placement stage of the generation the computation needs of the algorithms are determined, the computational components (ALUs, RAMs, multipliers, registers, etc) are created, and the physical elements are ordered along the one-dimensional datapath. Also, the netlist instances must be bound to the physical components. Both of these operations are performed as specified in Chapter 5. In the routing stage, the actual wires and multiplexers needed to connect the different components, including the I/Os, are created. The routing stage is discussed in more depth in section 6.2. After the generation algorithms are discussed, an area comparison is given between the architectures created with the cASIC techniques, and the standard cell, FPGA, and RaPiD solutions described in section 4.1.

6.1 Logic Generation

As stated previously, cASIC architectures use very coarse-grained logic blocks operating on full words of data. Computations can be implemented far more efficiently by dedicated units, like ALUs and multipliers, than by generic LUTs. This optimization allows the Totem (and RaPiD) architectures to implement algorithms in much less area than required by a traditional FPGA. Totem provides an additional benefit over static RaPiD architectures: the mix and arrangement of the logic resources can be customized

to the application netlists. The architecture generator reads in the target netlists, determines the type and quantity of logic units required, and orders these units along the one-dimensional axis, as described in Chapter 5. After the logic is in place, the routing must be created to allow the units to be connected together.

6.2 Routing Generation

While RaPiD uses a series of regular routing tracks, multiplexers, and demultiplexers to connect the units, cASIC architectures provide a more specialized communication structure. The only routing resources included are those which are explicitly required by one or more of the netlists. This section discusses the different techniques developed to create the routing structures.

The routing structure of a custom generated architecture will depend on the placement achieved using the techniques of Chapter 5. At this point, the physical locations of the components are fixed, as are the bindings of the netlist instances to those components. The specification netlists define the signals that connect the netlist instances to form a circuit. These instances have been bound in the placement stage, so the physical locations of the ports of the signals are known. Wires are then created to implement these signals, allowing each netlist to execute individually on the custom hardware.

In addition to wires, multiplexers and demultiplexers on the ports of the components may be created to accommodate the different requirements of the

specification netlists. For example, if netlist A needs an adder to output to a register, but netlist B needs the adder to output to a RAM, a demultiplexer is instantiated on the output of the adder to direct the signals properly for each netlist. Similarly, if netlist A has a register that receives an input from an adder, but netlist B needs that register to input from a multiplier, a multiplexer is created to choose the register input based on which netlist is currently active in the architecture. Figure 6.1 continues the example in Chapter 5, showing the generated routing structure for the placement of Figure 5.6. Note that several of the wires here are used to implement signals from both netlists. Like the logic resources, the wires are only used by one netlist at a time—whichever netlist is currently programmed onto the architecture. “Sharing” of routing resources between netlists is critical, as the routing architecture can become extremely large if each signal is implemented by a dedicated wire.

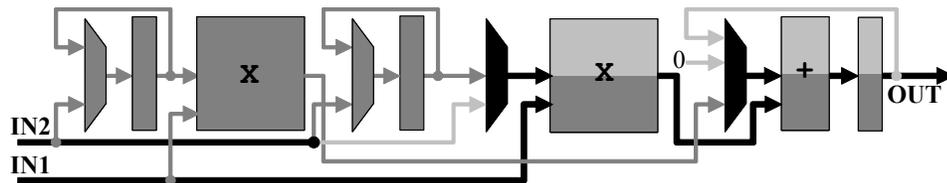


Figure 6.1: cASIC routing architecture created for the example from Chapter 5. Black wires are used by both netlists of Figure 5.2, while light wires are only used by the light netlist, and dark grey wires and logic are only used by the dark grey netlist. Logic resources used by both netlists show both colors.

The object of the routing generation phase is to minimize area by sharing wires between netlists, while adding as few multiplexers and demultiplexers as necessary. Heuristics are used to group signals with similar connections from different netlists into

wires. In order to understand the motivations for the algorithms presented below, the routing problem itself must first be discussed. As with the placement problem, creating the routing is two problems combined into one: creating the wires, and assigning of the signals to wires. In many current FPGA architectures, wire lengths can be adjusted for each netlist by taking advantage of programmable connections (segmentation points) between lengths of wire, potentially forming a single long wire out of several short wires. For simplicity, the current Totem cASIC generation algorithms do not provide this flexibility.

The algorithms must somehow determine which sets of signals belong together within a wire. One method is to simply not share at all, which is explored in the No Sharing algorithm. The remaining algorithms, Greedy, Bipartite and Clique, use heuristics to determine how the wires should be shared between signals. The heuristics operate by placing signals with a high degree of similarity together into the same wire. However, “similarity” can be computed several different ways. In this work, two different definitions of “similarity” were used. *Ports* refers to the number of input/output locations the signals or wires have in common. *Overlap* refers to a common “span”, where the span of a signal or wire is bounded by the leftmost source or sink and the rightmost source or sink in the placed architecture. Results for each of these similarity types are given in section 6.3. The procedures used by the Greedy, Bipartite, and Clique heuristics are described in the next sections.

6.2.1 Greedy

The greedy algorithm operates by repeatedly merging wires that are very similar. To begin, each signal is assigned to its own wire. Next, a list of correlations between all compatible wire pairs (wires that are not both used in the same netlist) is created. The highest correlation value is selected at each iteration, and those two wires are merged. All other correlations related to either of the two wires that have been merged are updated according to the characteristics of the new shared wire. If any of the correlations now contain a conflict due to the new attributes of the merged wire (i.e., both wires in the correlation hold a signal from the same netlist), these correlations are deleted from the list, as they are no longer valid. This process continues until the correlation list is empty, and no further wires may be merged. The pseudocode for this operation is given in Figure 6.2.

```

Greedy() {
  Let overlap( $w_i, w_j$ ) return shared span of wires  $w_i$  and  $w_j$ 
  Let ports( $w_i, w_j$ ) return shared ports of wires  $w_i$  and  $w_j$ 

  Let S = the set of all signals
  Let W = a set of wires (initially empty)
  Create one wire for each signal in S, assign the signal to the wire,
    and add the wire to set W
  Let L = a list of all pairs of wires in W

  // greedily merge wires
  Loop {
    For each pair ( $w_i, w_j$ ) of wires in L {
      If  $w_i$  and  $w_j$  contain signals from the same netlist,
        Delete pair from L
      Else if ( $\text{overlap}(w_i, w_j) = 0$  and  $\text{ports}(w_i, w_j) = 0$ ),
        Delete pair from L
    }
    If L empty, quit
    Find pair ( $w_i, w_j$ ) from L with greatest overlap() or ports() value,
      depending on similarity method used, breaking ties using other
      similarity
    Remove pair from L
    Remove  $w_i$  and  $w_j$  from W
    Make new wire  $w_k = w_i \cup w_j$ , and add it to W
    Update all pairs in L that refer to  $w_i$  or  $w_j$  to refer to  $w_k$  instead
  }
  Return W
}

```

Figure 6.2: Pseudocode for the Greedy cASIC generation technique.

6.2.2 Bipartite

The merging of netlists into cASIC architectures is a form of matching problem. This might naturally lead one to consider the use of bipartite matching to solve the problem. One group has already used maximum weight bipartite matching to match netlist instances together to form the components [Huang01]. However, there are two significant problems with this approach. The first is that this type of logic construction does not consider the physical locations of the instances or their components. The physical locations of components and mapped instances determines the length of wires

needed to make the connections between units, and is therefore critical to effective logic construction. Furthermore, that particular work indicated that wires would not be shared between signals after the logic was constructed using the bipartite matching method.

Second, the bipartite matching algorithm was used recursively, matching two netlists together, then matching a third netlist to the existing matching, and so on. While any individual matching can be guaranteed to have the maximum weight, the cumulative solution may not. The order in which the netlists are matched can affect the quality of the final solution. This is true even if bipartite matching is not used for the logic construction but only for routing construction.

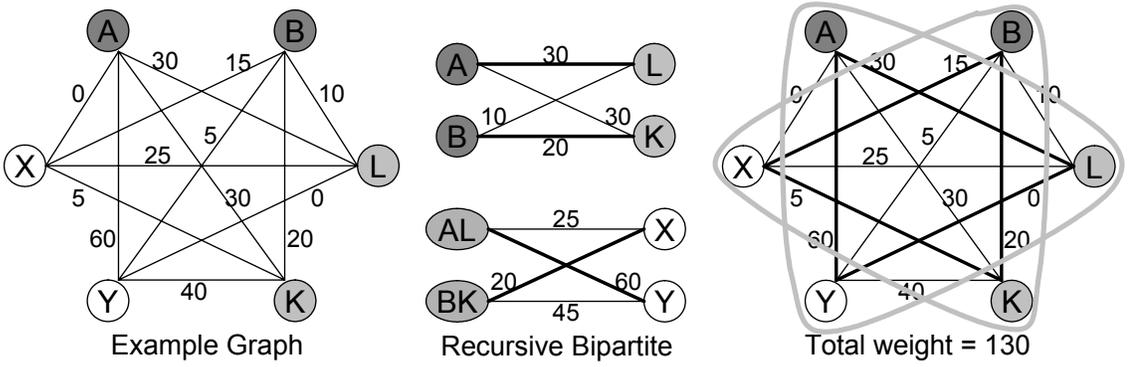


Figure 6.3: An example graph which does not produce the optimal solution when bipartite matching is used recursively. First the dark grey nodes are matched to the light grey nodes, and then the matching is matched to the white nodes. The final grouping formed by the bipartite matching process is shown at right. The total weight is the sum of the weights of all the edges completely contained within a grouping. Figure 6.6 shows a superior solution found using clique partitioning.

To demonstrate that the ordering of the netlists has an effect on the quality of the resulting solution, a cASIC generation algorithm was created which uses recursive maximum weight bipartite matching. Logic for these architectures is still constructed as

discussed in Chapter 5 because of the location issue mentioned previously, but the merging of signals into wires is performed using bipartite matching.

```

Bipartite() {
  Let overlap( $w_i, w_j$ ) return shared span of wires  $w_i$  and  $w_j$ 
  Let ports( $w_i, w_j$ ) return shared ports of wires  $w_i$  and  $w_j$ 

  Let N = the set of netlists
  Let Order = a matrix containing the indices of the netlists in the
    order they are to be merged, ( $|Order|$  = the number of netlists)

  Let S = the set of all signals
  Let W = a set of wires (initially empty)
  Create one wire for each signal in S, assign the signal to the wire,
    and add the wire to set W

  Let L be the set of all wires in W that were created specifically for
    the signals in netlist  $N_{Order[0]}$ 

  For  $netIndex = 1 \dots |Order| - 1$  {
    // L is the cumulative bipartite solution, R is the new netlist
    Let Weights be  $|L| \times |R|$  matrix of integers
    Let R be the set of all wires in W that were created specifically
      for the signals in netlist  $N_{Order[netIndex]}$ 
    For each  $l_i$  in L {
      For each  $r_j$  in R {
        If method is overlap,
          Weights[i][j] = overlap( $l_i, r_j$ )
        Else if method is ports,
          Weights[i][j] = ports( $l_i, r_j$ )
      }
    }

    // perform the maximum weight bipartite matching
    Let Match = the solution from bipartite_match(L,R,Weights)
    Empty set L
    For each pair ( $l_i, r_j$ ) in Match {
      Make new wire  $w_k = l_i \cup r_j$ , and add it to L
    }
  }
  Return L, the cumulative bipartite solution
}

```

Figure 6.4: Pseudocode for the recursive maximum weight bipartite matching cASIC technique. The pseudocode for the maximum weight bipartite matching algorithm itself appears in Figure 6.5.

```

bipartite_match(L,R,Weights) {
  // perform a maximum weight bipartite matching between the vertices
  // in L and R with edge weights in Weights[][]
  Let M be the solution set of pairs  $(l_i, r_j)$  to the problem
  (initially empty)
  Let X and Y be sets of vertices from L and R respectively,
  (initially empty)
  Let PX[i] and PY[j] be arrays of sets of vertex pairs (initially
  empty), where  $i = 0 \dots |L|$  and  $j = 0 \dots |R|$ 

  Loop {
    Let X be the set of all vertices in L that have not been matched
    Let DX[k] = 0 if vertex  $l_k$  is in X,  $-\infty$  otherwise
    Let DY[k] =  $-\infty$  for all vertices  $r_k$  in R

    While X not empty {
      // Phase 1: see if adding an edge to the matching helps
      Empty Y
      For all pairs  $(l_x, r_y)$  not in M with  $l_x$  in X {
        If  $DX[x] + Weights[x][y] > DY[y]$ ,
           $DY[y] = DX[x] + Weights[x][y]$ 
          Add vertex  $r_y$  to Y
          // keep track of the set of changes to the matching
           $PY[y] = PX[x] \cup (l_x, r_y)$ 
        }
      }

      // Phase 2: see if subtracting an edge from the matching helps
      Empty X
      For all pairs  $(l_x, r_y)$  in M with  $r_y$  in Y {
        If  $DY[y] - Weights[x][y] > DX[x]$ ,
           $DX[x] = DY[y] - Weights[x][y]$ 
          Add vertex  $l_x$  to X
          // keep track of the set of changes to the matching
           $PY[y] = PY[y] \cup (l_x, r_y)$ 
           $PX[x] = PY[y]$ 
        }
      }
    }

    Let  $r_y$  be an unmatched vertex from R with maximum  $DY[y]$ 
    If  $DY[y] > 0$  {
      // add path  $PY[y]$  to M
      For all vertex pairs  $(l_x, r_y)$  in  $PY[y]$  {
        If  $(l_x, r_y)$  is not in M
           $M = M \cup (l_x, r_y)$ 
        Else
           $M = M - (l_x, r_y)$ 
        }
      } Else return M
    }
  }
}

```

Figure 6.5: Pseudocode of the maximum weight bipartite matching graph algorithm [Shier99] used by the Bipartite cASIC generation algorithm from Figure 6.4.

6.2.3 Clique

The downside of the Greedy and Bipartite techniques is that they merge wires based on short-term local benefits, without considering the problem as a whole. There may be cases where merging the two most similar wires at one point prevents a more ideal merging later in the algorithm. Therefore, the Clique algorithm has been implemented to globally address the routing creation problem.

Clique partitioning is a concept from graph algorithms whereby vertices are divided into completely connected groups. In our algorithm each wire is represented by a vertex, and the "groups", or cliques, represent physical wires. The algorithm uses a weighted-edge version of clique partitioning to group highly correlated signals together into wires, where the correlation value between signals is used as the edge weight. The cliques are then partitioned such that the weight of the edges connecting vertices within the same clique is maximized. Signals that cannot occupy the same wire (signals from the same netlist) carry an extremely large negative weight that will prevent them from being assigned to the same clique. Therefore, although signal A may be highly correlated with signal B, and signal B is highly correlated with signal C, they will not all be placed into the same wire (clique) if signal A conflicts with signal C, due to the large negative weight between those vertices. Figure 6.6 shows the clique partitioning solution to the weighted-edge graph from the example of Figure 6.3..

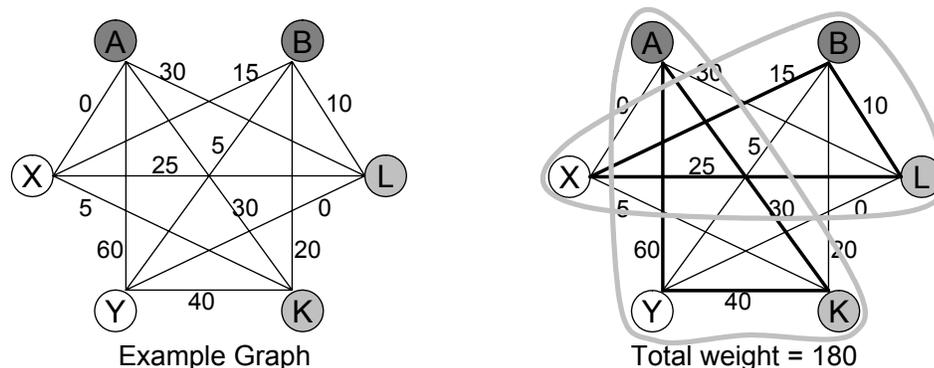


Figure 6.6: An improved solution to the graph of Figure 6.3 found using clique partitioning. The nodes in this graph represent the signals, and each clique (grouping) represents a wire shared by the vertices (signals) in the cliques. The total weight is the sum of all edge weights completely contained within a clique.

Given that weighted clique partitioning of a graph with both negative and positive edge weights is NP-Complete, an ejection chain heuristic based on tabu search [Dorndorf94] is used. Vertices are initially assigned to random cliques (where the number of cliques equals the number of vertices). Some cliques are allowed to be empty, but all vertices must be assigned to a clique. The algorithm then iteratively moves each vertex from its current clique to a different one. This is done by each time selecting a non-tabu vertex and a new clique for that vertex that will produce the maximum overall (not necessarily positive) gain in total weight for the graph. Once a vertex is moved, it is marked tabu until the next iteration.

After all the vertices have been moved in an iteration, the list of cumulative solutions after each move is examined, and the one with the highest total weight is chosen. This solution is then used as the base for the next iteration of moves, and all

vertices are marked non-tabu. This loop continues until none of the cumulative solutions in an iteration produces a total weight greater than the base solution for that iteration.

```

Clique() {
  Let S = the set of all signals
  Let W = the set of wires, initially empty
  Let Weights be |S|x|S| matrix of integers

  // load weights between signals (vertices) into matrix
  For each signal si in S {
    For each signal sj in S {
      If si = sj, Weights[i][j] = 0
      Else if si and sj are from the same netlist, Weights[i][j] = -∞
      Else Weights[i][j] = edge_weight(si,sj)
    }
  }

  // perform the partitioning
  Let CliqueSol = the solution set of cliques from
  clique_partition(S,Weights)

  // translate cliques to wires
  For each clique Ci in CliqueSol {
    Create a new wire wi, and add it to W
    For each vertex signal sk in Ci {
      Assign sk to wi, adding sk's connections to wk
    }
  }
  Return W
}

edge_weight(si,sj) {
  // Find the edge weight between signals (vertices) s1 and s2
  If method is overlap {
    Let l(sm) return length of signal sm
    Let shared(sm,sn) return shared span of signals sm and sn
    Return
      2*shared(si,sj) - (l(si)-shared(si,sj)) - (l(sj)-shared(si,sj))
  } Else if method is ports {
    Let p(sm) return the number of ports on signal sm
    Let shared(sm,sn) return number of ports in common of sm and sn
    Return
      2*shared(si,sj) - (p(si)-shared(si,sj)) - (p(sj)-shared(si,sj))
  }
}

```

Figure 6.7: The pseudocode of the Clique cASIC generation algorithm. The pseudocode of the clique partitioning graph algorithm itself appears in Figure 6.8.

```

clique_partition(S,Weights) {
  // perform a clique partitioning of vertices in S
  // with edge weights in Weights[][]

  Let weight( $C_i$ ) = the sum of weights of all edges between the
    vertices contained by clique  $C_i$ 
  Let weight(Sol) = the sum of weights of all cliques in solution Sol

  Let SolList = a list of clique solutions, initially empty
  Let Sol = clique solutions (i.e., a set of cliques)

  Initialize OldSol to hold |S| cliques
  Assign each vertex  $s_i$  in S to random clique in OldSol
  Add Sol to SolList

  Loop {
    Mark all vertices  $s_i$  in S as not tabu

    While there are non-tabu vertices {
      For each vertex  $s_i$  that is not tabu {
        Let  $C_i$  be the clique  $s_i$  currently belongs to
        For each clique  $C_j \neq C_i$  {
           $gain_j = \Delta \text{weight}(C_i) + \Delta \text{weight}(C_j)$  if  $s_i$  were moved to  $C_j$ 
        }
        Let  $C_{max}$  = clique with maximum gain
        Mark vertex  $s_i$  as tabu, and move it to clique  $C_{max}$ 
        Let Sol = the updated set of cliques, and add it to SolList
      }
    }
    Find solution BestSol with the greatest total weight in SolList
    If BestSol is first solution in SolList, return BestSol
    Sol = BestSol
    Empty SolList
  }
}

```

Figure 6.8: Pseudocode of the clique partitioning graph algorithm [Dorndorf94] used by the Clique cASIC generation algorithm from Figure 6.7.

6.3 Results

For the cASIC architecture generation methods, the areas are computed based on the manual layouts used for the RaPiD area calculation from section 4.3.3. Logic area is computed using the same method, but the routing area is a more complex computation. Area used by multiplexers and demultiplexers are again computed according to manual

layouts. But unlike RaPiD, wire area can add to the total area of the architecture. A wire cross-section of up to 24 can be routed directly over the logic units, so as with RaPiD, this routing area is considered “free”. However, when the routing cross-section is greater than 24, the additional cross-section adds to the height of the architecture.

Table 6.1: The areas of the routing structures created by the Bipartite cASIC generation methods using both the ports and the overlap methods. All possible orderings of the netlists were considered, and the minimum, average, and maximum areas are given. The percent difference between the minimum and maximum areas is also given.

	Radar	OFDM	Camera	Speech	FIR	Matrix	Sort	Image
# Netlists	3	2	3	2	6	5	4	5
Ports Min	0.075	0.731	0.958	0.476	0.520	0.093	0.183	0.573
Ports Avg	0.076	0.731	0.959	0.476	0.582	0.094	0.185	0.582
Ports Max	0.077	0.731	0.959	0.476	0.629	0.097	0.187	0.589
% Diff	2.273	0.000	0.089	0.000	20.914	4.587	2.337	2.780
Overlap Min	0.093	0.429	0.301	0.371	0.247	0.131	0.248	0.573
Overlap Avg	0.094	0.429	0.305	0.373	0.263	0.140	0.251	0.582
Overlap Max	0.094	0.429	0.307	0.374	0.273	0.145	0.255	0.589
% Diff	0.917	0.000	1.983	0.690	10.345	10.390	2.749	2.780

First, the Bipartite technique was examined to determine the effect of the order that netlists are merged into the cumulative solution. Table 6.1 lists, for each application, the minimum, average, and maximum areas across the solutions for each ordering of the netlists. The percent difference between the minimum and maximum areas is also given. When there are only two netlists, there is only one possible ordering, and the minimum and maximum values are identical. However, these results indicate that for any cases with more than two netlists, the ordering can affect the final area. In one case, the routing area varies by 20%. Therefore, this technique may not be appropriate for cases with more than two netlists.

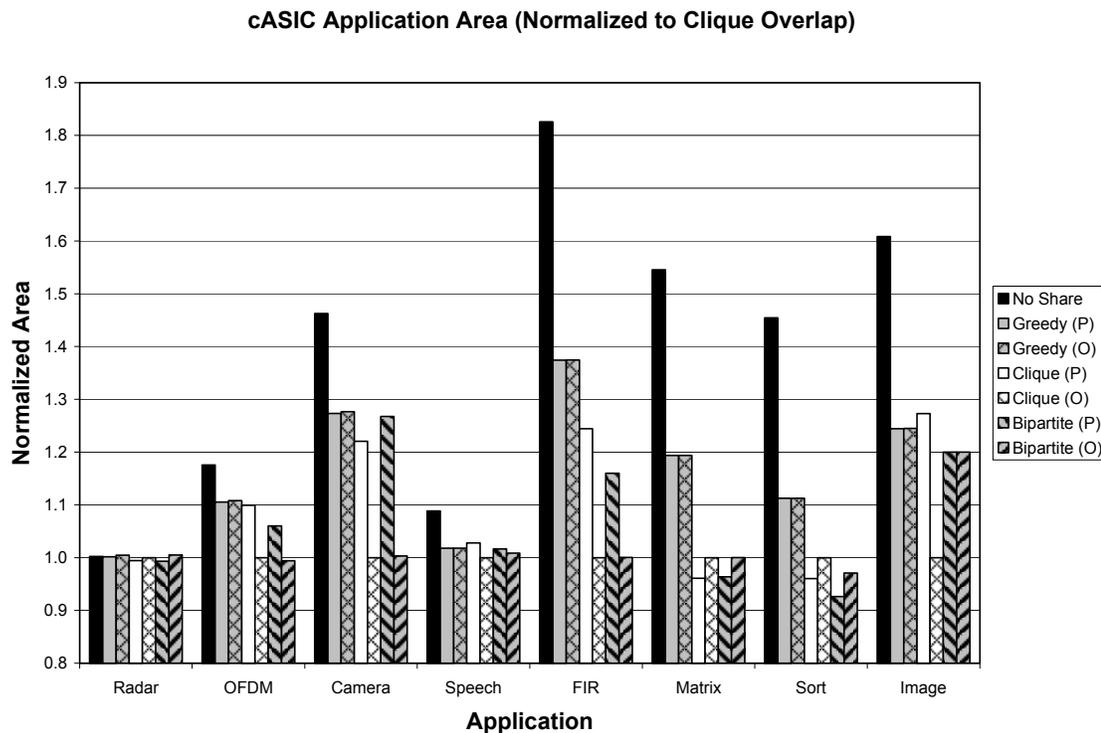


Figure 6.9: Comparative area results of the different cASIC routing generation algorithms, normalized to the Clique Overlap result for each application. The Bipartite results given are the average across orderings.

The No Sharing algorithm creates a separate wire for every signal—a completely different set of wires is used depending on which netlist is in operation. This method is included to demonstrate the importance of sharing routing resources between netlists. Areas are listed for Greedy, the average Bipartite case, and Clique, each with two categories: Ports, and Overlap. As stated previously, Ports indicates that the similarity between signals is computed according to the number of sources and sinks shared by those signals. Overlap indicates that the similarity is computed according to common location and length of the signals.

An area comparison of the cASIC methods is given in Figure 6.9, which has been normalized to the area result of Clique Overlap (which on average produces the smallest architectures). As expected, all three heuristic techniques of both similarity types perform better than the No Share algorithm for all applications. Generally, Clique performs better than the other methods, with Clique Overlap on average 2% smaller than Bipartite Overlap, 6% smaller than Bipartite Ports, and 13% smaller than Greedy Ports or Greedy Overlap. However, there is clearly room for improvement of the weighting calculation used by the Clique Partitioning method, as both Greedy and the average Bipartite produce a smaller area in some situations. Additionally, Clique sometimes performs better using Ports, and other times using Overlap. An improved similarity (weight) calculation for the Clique method should therefore incorporate both Ports and Overlap similarity techniques.

Table 6.2: The areas, in mm², of the eight different applications from Table 4.1, as implemented with the three cASIC algorithms. The results from Table 4.2 though Table 4.4 are included for reference. For the Greedy and Clique Partitioning cASIC method, results for both Ports and Overlap style similarity computations are given. A summary of these results appears in Table 6.3.

		Radar	OFDM	Camera	Speech	FIR	Matrix	Sort	Image
Std. Cell	Total	4.101	9.168	7.268	26.523	2.846	1.785	1.541	6.843
FPGA	Total	19.719	59.157	23.006	78.877	26.292	19.719	26.292	19.719
RaPiD	Logic	2.838	---	---	45.401	3.783	1.892	2.838	---
	Routing	2.158	---	---	34.536	2.878	1.439	2.158	---
	Total	4.996	---	---	79.937	6.661	3.331	4.996	---
No Share	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.090	1.259	1.441	1.413	1.917	0.829	0.970	1.331
	Total	1.523	5.377	3.619	14.161	3.658	1.952	2.163	2.947
Greedy Ports	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.089	0.938	0.973	0.493	1.011	0.385	0.462	0.664
	Total	1.522	5.055	3.151	13.242	2.753	1.508	1.655	2.280
Greedy Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.093	0.950	0.981	0.493	1.012	0.385	0.462	0.664
	Total	1.526	5.068	3.159	13.242	2.754	1.508	1.655	2.281
Clique Ports	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.078	0.909	0.842	0.622	0.752	0.090	0.236	0.716
	Total	1.511	5.027	3.020	13.370	2.493	1.214	1.428	2.333
Clique Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.086	0.456	0.297	0.261	0.262	0.140	0.294	0.216
	Total	1.520	4.574	2.475	13.010	2.004	1.264	1.487	1.833
Bipartite Min Ports	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.075	0.731	0.958	0.476	0.520	0.093	0.183	0.573
	Total	1.509	4.849	3.136	13.224	2.262	1.217	1.376	2.190
Bipartite Avg Ports	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.076	0.731	0.959	0.476	0.582	0.094	0.185	0.582
	Total	1.509	4.849	3.136	13.224	2.324	1.218	1.378	2.199
Bipartite Max Ports	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.077	0.731	0.959	0.476	0.629	0.097	0.187	0.589
	Total	1.510	4.849	3.137	13.224	2.370	1.221	1.380	2.206
Bipartite Min Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.093	0.429	0.301	0.371	0.247	0.131	0.248	0.573
	Total	1.526	4.547	2.479	13.120	1.989	1.255	1.441	2.190
Bipartite Avg Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.094	0.429	0.305	0.373	0.263	0.140	0.251	0.582
	Total	1.527	4.547	2.483	13.121	2.005	1.264	1.444	2.199
Bipartite Max Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.094	0.429	0.307	0.374	0.273	0.145	0.255	0.589
	Total	1.527	4.547	2.485	13.122	2.015	1.269	1.448	2.206

Table 6.3: Area improvements calculated over the reference architectures, then averaged across all applications. The Bipartite results are the average across netlist orderings.

Improvement Over Std. Cells		Improvement Over RaPiD		Improvement Over FPGA	
Method	Area	Method	Area	Method	Area
FPGA	0.20	Std Cells	2.34	Std Cells	7.20
RaPiD	0.48	FPGA	0.38	RaPiD	4.01
No Share	1.63	No Share	2.95	No Share	9.00
Greedy (P)	1.87	Greedy (P)	3.39	Greedy (P)	10.63
Greedy (O)	1.87	Greedy (O)	3.39	Greedy (O)	10.62
Clique (P)	1.94	Clique (P)	3.64	Clique (P)	11.50
Clique (O)	2.16	Clique (O)	3.75	Clique (O)	12.30
Bipartite (P)	1.98	Bipartite (P)	3.72	Bipartite (P)	11.77
Bipartite (O)	2.08	Bipartite (O)	3.76	Bipartite (O)	12.14

Table 6.2 gives the areas found by the different cASIC routing generation algorithms, with the corresponding standard cell, FPGA, and RaPiD areas listed for comparison. These results are summarized in Table 6.3. As these tables indicate, cASIC architectures are significantly smaller than the corresponding FPGA area for the same netlists. The heuristics range on average from a 10.6x improvement to an 12.3x improvement in area, while even the No Sharing algorithm results in a 9x improvement. Given the results presented in section 4.3.4, FPGAs without custom embedded multipliers and RAMs would be expected to require even more area than the VirtexII for these applications.

Comparisons of cASIC techniques to RaPiD also yield favorable results, with area improvements of 3.4x to 3.8x for the cASIC heuristics. These applications were created for RaPiD, and RaPiD has been hand-optimized for DSP-type operations, which make it more efficient (2.8x smaller) than a generic FPGA for these applications.

Finally, the cASIC heuristic methods also created architectures on average half the size of standard cell implementations of the applications. One of the reasons the cASIC architectures are able to achieve such small areas is because the tools use full-custom layouts for the computation blocks. The FIR, Matrix, and Sort architectures demonstrate the value of the full-custom units. In these applications, the standard cell area is estimated to be the size of the largest member netlist (as explained in section 4.3.1). Even with the overhead of adding reconfigurability, these cASIC area results are close to or slightly better than the standard cell implementation. The largest benefits, however, occur in the cases where an application has several differently-structured netlists, and a separate circuit must be created for each member netlist in a standard cell implementation. By reusing components for different netlists, the cASIC architectures achieve areas on the order of a full-custom implementation (generally assumed 2-3x smaller than standard cells).

The cASIC method of architecture creation therefore has significant area benefits for situations in which standard cells are generally considered instead of FPGAs for efficiency reasons. Finally, a full-custom manual layout could be created for these applications that might be smaller than the cASIC architectures. However, this would require considerably more design time, which can be quite expensive, and may not always be possible due to production deadlines.

6.4 Summary

This chapter described the cASIC style of architecture, and presented three different heuristics to create these designs. The first uses a greedy approach, the second uses recursive maximum weight bipartite matching, while the third uses a more sophisticated graph-based algorithm called clique partitioning to merge groups of similar signals into wires. Two different methods to measure this signal similarity were discussed, one based on the common ports of the signals, and the other based on the common span (overlap). The results of the chapter indicate that a better similarity measurement would be a combination of the two, incorporating both ports and signal overlap.

The area comparison also demonstrates the inefficiencies introduced by the flexibility of FPGAs. While the generic structure is critical for implementing as wide a variety of circuits as possible, it is that flexibility that causes it to require 12x more area than a cASIC architecture. The VirtexII FPGA does, however, perform much better than earlier designs due to the use of coarse-grained multiplier and RAM units. The RaPiD architecture extends the use of coarse-grained units to the entire architecture, but is customized to DSP as a whole. If the application set is only a subset of DSP, further optimization opportunities exist, with cASIC techniques achieving up to 3.8x area improvements over the RaPiD solution. This chapter also demonstrated another key benefit cASIC generation has over the use of a static architecture such as RaPiD. In

cASIC generation, if enough area is allotted on the SoC die, an architecture can be created for any set of netlists. On the other hand, the RaPiD resource mix is fixed. For some applications, this structure may not have the correct logic mix for the application, leading to copious wasted area. Alternately, a static structure may not provide a rich enough routing fabric, as was demonstrated in this chapter. Finally, the area results presented here indicate that cASIC architecture design is not only an excellent alternative to FPGA structures when the target circuits are known, but also a viable alternative to standard cell implementations, with architectures under half the size.

Chapter 7

Flexible Architectures

Chapter 6 concerned the automatic creation of coarse-grained architectures with only the minimum amount of reconfigurability required to implement a given application set. The applications, in RaPiD netlist format, formed the architecture specification, and the generation tool created a very ASIC-like design. This type of design is unlikely to be flexible enough to handle any changes, such as bug-fixes, upgrades, or additional algorithms. Instead the focus was to provide for a high degree of hardware re-use among the applications, assuming the full application set was known in advance.

This chapter instead concentrates on the generation of flexible reconfigurable routing architectures for the Totem Project, exploring a number of different heuristics. The automatic generation of routing architectures is critical to reconfigurable architecture design. An architecture could contain an overabundance of logic units, but unless these units can be connected to implement a required circuit, the architecture is useless. On the other hand, an architecture with the minimum number of computational units required by the application can be used, provided enough routing resources are available.

The next few sections describe the structure of both the logic and the routing of the flexible architectures, with the primary focus on the generation of the routing structures. The algorithms used to generate the routing structures will be discussed in depth, with accompanying pseudocode. The area results of the architecture generation algorithms are then given, and compared to the standard cell, FPGA, and RaPiD results from Chapter 6. Finally, a quick flexibility comparison of the architectures is presented, with a more in-depth flexibility examination given in Chapter 9.

7.1 Flexible Architectural Style

The goal of the Totem flexible architecture generation is to automatically create an architecture customized for a particular application set, with some extra resources if desired for future flexibility. These flexible architectures are in the same style as RaPiD [Ebeling96, Cronquist99a], with coarse units, 1D design, word-sized routing, bus connectors, and multiplexers/demultiplexers. A detailed description of the RaPiD architecture is given in section 4.1. There are some differences between RaPiD and the generated architectures, as well as between distinct generated architectures. The first difference is in terms of the logic composition, including quantity of units, proportions of types of units, and arrangements along the 1D axis of the architecture. The routing architecture also changes, with the quantity, types (local/distance), and segment lengths of the tracks altered depending on the needs of the netlists. In RaPiD, each local track contains some wires that span four logic units and some wires that span five, whereas

Totem architectures are restricted to only one wire length per track (wire length is defined as the number of logic units spanned). Like RaPiD, Totem architectures restrict each track to only be one type: local or distance.

While RaPiD has a fixed logic mix and set of routing tracks, the Totem algorithms can vary these particular features to achieve a more customized architecture. Essentially, RaPiD becomes one architectural instance that the flexible architecture generation tool could create. However, the tool should be able to create architectures both larger and smaller, depending on the specified domain. For example, RaPiD was designed specifically for DSP operations. If a user only needs a small set of FIR filters, flexible architecture generation will be able to automatically create an architecture more optimized than RaPiD for FIR filters. Likewise, if the tool is presented with a set of netlists which require more resources than the RaPiD architecture might provide, the generation tool will still be able to create an architecture which can implement those netlists.

7.2 Logic Generation

The logic generation step for flexible architecture generation is a slightly modified version of the logic generation described in Chapter 5 and used for the cASIC designs. Among the changes is the ability to specify additional logic resources. The tool outputs the minimum number of each type of unit required to implement the given netlists, and the user can specify if he or she wishes to add more. Simulated annealing

[Sechen88] is then used to simultaneously find a physical placement of the units and a mapping of the netlists to those physical components.

Unlike in Chapter 5, logic components can be forced to be spaced evenly throughout the architecture. Some of the routing generation algorithms focus on the creation of a very regular structure (for which there should be a regular logic structure). In this case, the physical units are first spread evenly though the architecture. The simulated annealing phase is then used to bind the netlist instances to physical components, much like traditional FPGA placement. The physical components are not moved during this operation. The physical placement and netlist mapping are needed to determine the connectivity requirements of the circuits for the routing generation stage.

7.3 Routing Generation

After this placement is performed, the routing algorithms heuristically generate the configurable routing structure based on the signal locations and lengths within the application netlists. At the highest level of these algorithms, each essentially iteratively adds routing tracks until all signals from all the netlists in the problem can be routed onto the architecture. After all the tracks have been added, the user is given the opportunity to increase the routing tracks over the calculated minimum to provide additional flexibility. Figure 7.1 illustrates an example flexible routing architecture that might be created from the final placement of Figure 5.6.

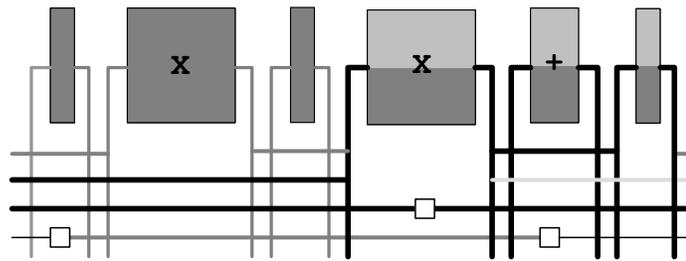


Figure 7.1: Flexible routing architecture created for the example from Chapter 5. The vertical lines represent the multiplexers and demultiplexers that are on every port of each unit. Black wires are used by both netlists of Figure 5.2, while light wires are only used by the light netlist. Dark grey wires and logic are only used by the dark grey netlist. Logic resources used by both netlists show both colors.

7.3.1 Shared Concepts

Each of the routing generation algorithms shares a few key concepts. The first is the difference between local and distance routing tracks. Local tracks, which are the upper routing tracks in RaPiD (Figure 4.1), and also appear in Figure 7.2a, are used for short connections. The length of a wire can be determined by finding the indices of the furthest units a wire can reach, and subtracting the left index from the right index. A wire that spans from the outputs of unit 11 to the inputs of unit 13 would therefore be considered a length 2 wire. A special track, the topmost shown in Figure 4.1 and Figure 7.2, is a track containing length 0 wires. These wires are also called "feedback" wires, as they only route from a unit's outputs to that unit's inputs.

Distance routing tracks are the bottom ones in RaPiD, and also appear in Figure 7.2b. These tracks include the added flexibility that longer wires can be created from shorter ones through the bus connectors. Each bus connector can be independently programmed to provide either a connection or a disconnect, and also provides an optional

pipeline delay. This allows for a great deal of routing flexibility, but can add delay as a signal passes through the bus connector, and adds a not-insignificant area penalty.

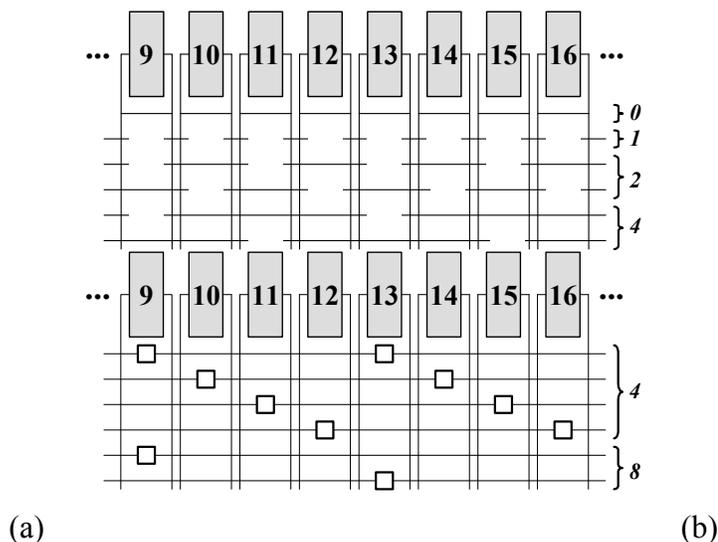


Figure 7.2: Examples of the different types of routing tracks, (a) local tracks, and (b) distance tracks with bus connectors (represented by the squares on the tracks). The “wire length” of each track is given to the right of the track.

Some arbitrary restrictions have been placed on the wire lengths allowable for the two different types of tracks, based on experience with the RaPiD architecture. The local tracks are allowed to have wire lengths of up to 8. The motivation here is that the local track wires are intended for short, fast connections, and at 8 or more units apart, the connections are no longer all that "local". For distance wires, there are two separate restrictions. First, distance wires have a minimum length of 8, because any shorter wire length in a distance track would add significant area overhead with the resulting bus connectors (which would potentially also slow down longer connections significantly). Finally, the distance routing tracks have also been restricted to have a wire length no

greater than 16. The architecture generation algorithms did not create very many length-16 tracks at all, and therefore, longer wire lengths are not permitted.

Another important concept, along with track type and wire length, is the "offset" of the track. This offset determines the left-right shifting (or "placement", as discussed in Chapter 8) of the track once its wire length has been determined. Figure 7.3 demonstrates a type of routing architecture where all tracks have the same offset value. This type of routing architecture is referred to as a "non-distributed" architecture. Note how the breaks between local wires and the connectors separating distance wires are clustered at identical horizontal positions. With this type of architecture, the placement of a circuit would greatly affect its routability. The routing choices available to a signal can be very dissimilar (and potentially undesirable) depending on the location of the signal's source and sink components.

On the other hand, by carefully choosing offset values for the tracks, an architecture closer in design to the one shown in Figure 7.4 can be achieved, which is referred to as a "distributed" routing architecture. Note how the shifting of the tracks allows the breaks and bus connectors to be evenly distributed horizontally through the architecture. This type of routing architecture should provide more flexibility in routing, as it provides a variety of routing choices for signals connecting to each component. A more in-depth study of track placement issues, along with a description of the algorithm used here, is presented in Chapter 8.

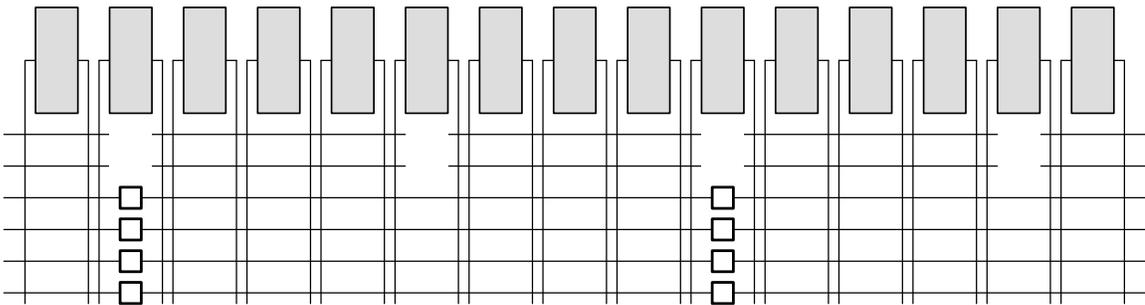


Figure 7.3: An extreme example of a non-distributed routing architecture. Bus connectors and breaks between local wires are clustered at specific horizontal locations. This leads to a limited variety of routing possibilities for each position.

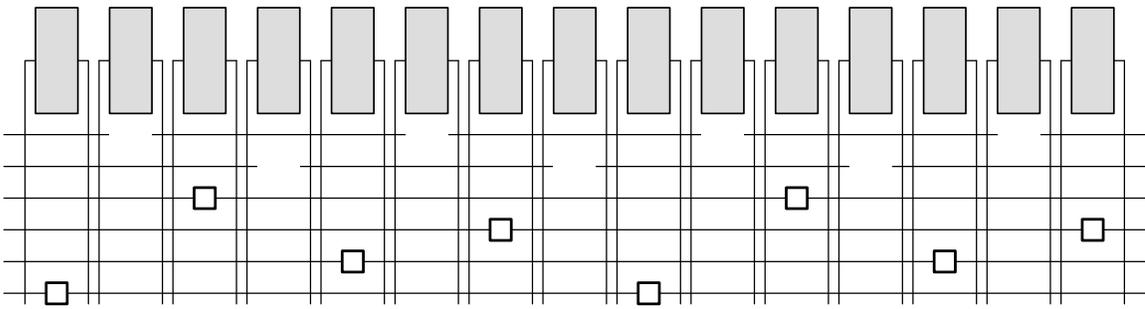


Figure 7.4: A distributed routing architecture. The breaks and bus connectors are distributed through different horizontal locations of the architecture, providing a variety of routing options at each position.

Evaluating Progress During Routing Construction

Each of the algorithms that will be presented is constructive—tracks are added until all signals in the application netlists can be routed onto physical wires. These algorithms operate in a greedy fashion, adding the track type and track length that provides the most benefit for the lowest cost. Cost is assumed to be area and delay, where bus connectors consume area and increase delay, and shorter wires are faster than long wires. However, the relative “benefit” of adding different track types or lengths must also be measured.

At the start of the execution of a routing generation algorithm, no tracks have yet been constructed, and therefore all signals are “unroutable”. As tracks are added, the number of unroutable signals decreases. Therefore, the cross-section of signals that cannot yet be routed onto the routing architecture is used to monitor the progress of the algorithms. This is calculated by finding the maximum signal cross-section at any component location for each individual netlist, then finding the maximum of these values across all netlists.

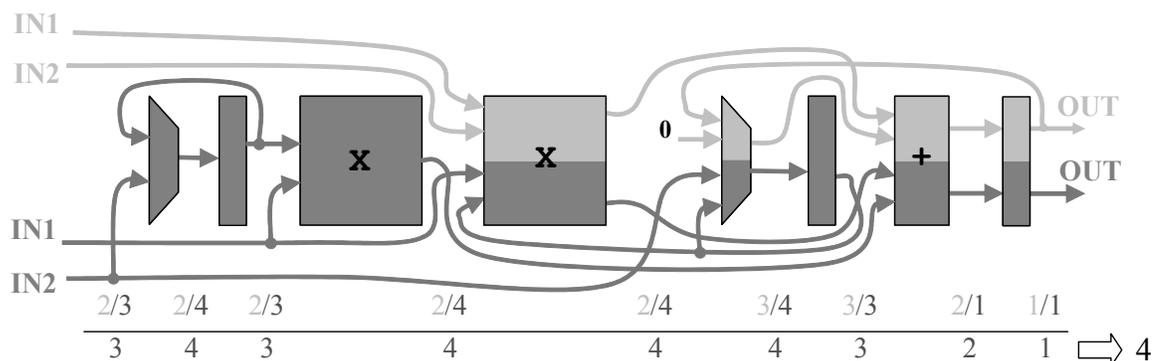


Figure 7.5: Calculating the unrouteable cross-section for the placement of Figure 5.6. The signal cross-section is given for each netlist (light grey netlist / dark grey netlist) at each point in the placement. The maximum cross-section across the netlists appears below, and the overall cross-section at right.

This value is also used as a lower bound on the number of tracks that need to be added to implement the source netlists. When the lower bound is determined before any tracks are created, it represents the minimum total number of tracks required to make the architecture routable for the given netlists. This lower bound would, however, be an improbable solution due to the very high number of bus connectors that would likely be required in order to use that few tracks. Each of the algorithms also uses this lower

bound calculation throughout architecture generation as one of the indicators of whether or not a particular track is providing any "benefit" to the architecture. The algorithms attempt to add tracks that will decrease the unroutable signal cross-section value.

Fast Integrated Router

In order to calculate the cross-section of unroutable signals, the generator must determine which signals are and are not routable. Because this is a frequent operation performed within loops, a very fast router (faster than the higher-quality router used for "final" routing presented in section 4.2.3) that provides reasonable results is needed. Initially, a left-edge algorithm [Hashimoto71] was considered. However, because not all signals will be routable most of the time, and routing decisions affect the construction of tracks, this algorithm would not be appropriate.

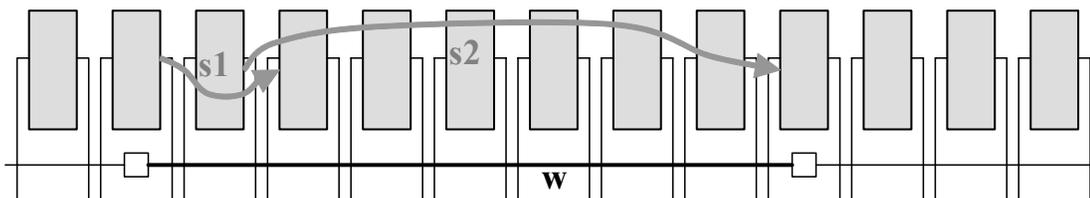


Figure 7.6: An example situation where an unmodified left-edge routing algorithm leads a routing generation algorithm to construct a more expensive solution. A left-edge algorithm will route s_1 onto wire w , which causes the creation of another long track to implement signal s_2 .

Figure 7.6 illustrates why this is the case. Signal s_1 with a length of 2 appears before the length-7 signal s_2 in the list of signals sorted by left edge. Therefore, s_1 would be routed onto wire w , even though signal s_2 might "fit" the wire better. Signal s_2 would be considered "unroutable" in this example, and later a routing track would have to be

created to implement this signal. However, if s_2 were routed onto wire w , the routing generator could instead create a cheaper track to implement s_1 , leading to an overall more efficient solution.

Therefore, the left-edge algorithm was modified to use a greedy heuristic. Signals are still considered by increasing left edge. However, rather than routing each signal to the leftmost unassigned wire that can implement it, the algorithm also considers how closely the span of the signal matches the span of the wire. Each time a wire/signal pair is considered, other signals are considered which could also be routed onto the wire. The signal with the closest “fit” is routed onto the wire. If the original signal from the wire/signal pair was not chosen, that signal is reconsidered on the next iteration. For the routing operation, each netlist is considered a separate problem—signals from different netlists can be assigned to the same wire. The pseudocode for this routing operation is given in Figure 7.7.

The next few sections describe three different routing architecture generation algorithms. The first, Greedy Histogram, obtains solutions where the wire length and spacing within each track is uniform, but the breaks between local track wires and the bus connectors between distance track wires are not restricted to be distributed throughout the architecture. The remaining two algorithms, Add Max Once and Add Min Loop, focus on generating regular architectures, where the logic and routing (breaks and spaces) are very evenly distributed. The goal is to provide some uniformity in the number and type of routing resources available to a given signal regardless of where its source instance is

actually placed in the architecture. The details of each algorithm are presented below, followed by a comparison of the results obtained by each.

```

route_signals(T,S) {
  // route signals in S onto tracks in T

  Let S' = array of signals in S, sorted by increasing left edge
  Let W = array of wires in tracks in T, sorted by increasing left edge

  While S' not empty {
    Let s0 = first signal in S'
    For each wi in W {
      If wi already implements signal from s0's netlist, go to next wi
      If wi is from a local track and s0 fits in wi {
        // see if there's a better signal choice for wi
        Let sj be the signal in S' that is from a netlist not
          already using wi, fits in wi and shares the most span
          with wi
        Assign sj to wi and remove it from S'
        Break the For Loop
      } Else if wi is from a distance track and the left edge of s0
        lies within segment wi {
        Let sj be the signal in S' that is from a netlist not
          already using wi that has a left edge falling within wi
          and shares the most span with wi and any other needed
          distance segments
        Assign sj to wi and any other segments from that distance
          Track that it requires
        Break the For loop
      }
    }

    If the For loop completed without finding a wire that could
    implement s0 (regardless of whether or not s0 was the chosen
    sj), mark s0 as unroutable and remove it from S'
  }
}

```

Figure 7.7: Pseudocode summary of the fast greedy router used within the flexible routing generation algorithms presented in this chapter.

7.3.2 Greedy Histogram

This algorithm attempts to keep the overall number of tracks low, while encouraging the use of local tracks over distance tracks in order to avoid introducing a large number of bus connectors. Each track has a specific type and wire length used for

all wire segments within that track. However, no restrictions are made on the offsets that can be used. This creates a potentially non-distributed routing architecture which may not have uniform connectivity, and thus may not be as flexible as a more regular interconnect architecture. However, the goal of the algorithm is to customize the routing architecture significantly for the applications given (within the limitations of the previously described architectural style), and may also handle modified versions of these target applications. The pseudocode for this algorithm is given in Figure 7.8, with its sub-functions in Figure 7.9.

```

Greedy_Histogram() {
  Let S = the set of all signals
  Let U = the set of unroutable signals (initially all signals)
  Let T be the set of tracks (initially empty)
  While U not empty {
    Let H = histogram of U (unroutable signals) by signal length
    Let length = highest index of H that contains the max value in H

    If (length == 0)
      T = T + get_feedback_track(U)
    Else if (length < MIN_DISTANCE_LENGTH)
      T = T + get_local_track(U, length)
    Else if (length > MAX_LOCAL_LENGTH) {
      T = T + get_distance_track(U, length)
    } Else {
      Let tlocal = get_local_track(U, length)
      Let tdist = get_distance_track(U, length)
      Let localsol = routing solution after routing U onto tlocal
      Let distsol = routing solution after routing U onto tdist
      If distsol's unroutable cross-section < localsol's
        T = T + tdist
      Else T = T + tlocal
    }
    route_signals(T,S), and update U
  }
}

```

Figure 7.8: Pseudocode for the main body of the Greedy Histogram generation algorithm. Sub-functions appear in Figure 7.9.

In this algorithm, tracks are added one at a time within an infinite loop. The loop is broken when all of the netlists can be fully routed onto the architecture using the routing procedure discussed in section 7.3.1. The algorithm chooses the wire length for a "new" track by looking at the histogram of the lengths of the unroutable signals. The actual track creation method depends in part upon the wire length chosen, as indicated by the pseudocode in Figure 7.8 and its use of the sub-functions of Figure 7.9.

```

get_offset_guess(U, length) {
  Let H' = a histogram with indices 0...length-1
  For all i, 0...length-1 {
    Let H'[i] = the number of signals of U that would be routable onto
      a track of the given length at offset i.
  }
  Return index offsetGuess of the largest value in the array H'
}

get_feedback_track(U) {
  // feedback signals are efficiently implemented either by special
  // feedback tracks or regular length-2 local tracks
  offsetGuess = get_offset_guess(U, 2);
  Let t1 = length-2 local track with offset offsetGuess
  Let t2 = feedback track
  Let sol1 = routing solution after routing U onto t1
  Let sol2 = routing solution after routing U onto t2
  If sol1 has lower cross-section than sol2, or cross-sections are same
    but sol1 routed more signals from U, return t1
  Else return t2
}

get_local_track(U, length) {
  offsetGuess = get_offset_guess(U, length);
  Let besttrack = local track of given length using offsetGuess
  Let bestsol = routing solution after routing U onto besttrack
  For all toffset, 0...length-1 {
    Let t = local track of given length with offset toffset
    Let tsol = routing solution after routing U onto t
    If tsol has lower cross-section than bestsol, or cross-sections
      are same but tsol routed more signals
      Let besttrack = t
      Let bestsol = tsol
  }
  Return besttrack
}

get_distance_track(U, length) {
  offsetGuess = get_offset_guess(U, length);
  Let besttrack = distance track of given length using offsetGuess
  Let bestsol = routing solution after routing U onto besttrack
  For all tlength, MIN_DISTANCE_LENGTH...length {
    For all toffset, 0...tlength-1
      t = distance track of length tlength using toffset
      tsol = routing solution after routing U onto t
      If tsol has lower cross-section than bestsol, or cross-sections
        are same but tsol routed more signals
        Let besttrack = t
        Let bestsol = tsol
    }
  }
  Return besttrack
}

```

Figure 7.9: Sub-functions for the Greedy Histogram Algorithm from Figure 7.8.

If the length from the histogram is 0, this indicates a feedback signal, which can be implemented either on a wire with length 0 or a wire of length 2. Wires of length 1 will not fit this type of signal, because they only connect one unit's outputs to the inputs of the unit directly to the right. Both possibilities are tested to see which gives the best results in terms of reducing the cross-section of unroutable signals.

For lengths smaller than the minimum wire length of a distance routing track, all possible offsets (from 0 to $length-1$) are checked to find the best one for that length. An offset "guess" is also calculated by profiling the signals of the chosen length to determine the most common required track offset to route those signals. If none of the other offsets that are checked provide better results than the offset guess, that guess is used as the chosen offset.

For lengths greater than the maximum allowable length for local routing tracks, all wire lengths allowable for the distance routing tracks are compared, along with all possible offsets for each length, to find the track that reduces the histogram the most at the chosen length. The last remaining case is the one where the chosen length could be implemented by either a local track or a distance track. Here the best of each type of routing track is found using the methods just described. These two tracks are then compared to find the one that results in the smallest cross-section of unroutable signals, choosing the local track in the case of a tie in order to avoid unnecessarily adding bus connectors.

7.3.3 Regular Architectures

The next two algorithms generate a distributed routing architecture, where not only are the breaks or connectors evenly distributed within each track, but also across tracks. In other words, track offsets are chosen to provide a somewhat consistent level of connectivity regardless of location in the architecture. Figure 7.4 shows an example of this type of routing architecture.

In addition to specifying that the logic and routing be distributed, these algorithms also require that wire lengths must be a power of two. This restriction is common in FPGA architectures, and is intended to further generalize the architectures. Thus, the track possibilities are local tracks of length 0, 2, and 4, and distance tracks of length 8 and 16.

Add Max Once

The Add Max Once algorithm is more simple in organization than the Greedy Histogram algorithm. The algorithm begins at the shortest track length and progresses to the longest track length, seeing how many tracks of each type can be added while still improving the cross-section of unroutable signals. Tracks of the given type are added until no further reductions are possible with more tracks of that type. One issue with the Add Max Once algorithm is that only one type of distance routing track can be added. Given enough distance routing tracks of any length, all signals can eventually be routed by using the bus connectors to form longer wires when necessary. Therefore, the

algorithm would add enough tracks of the first distance length considered to route all remaining signals—no other distance track length would be reached. Length 8 distance routing tracks were chosen because a length of 16 would cause too many tracks to be created, and because this is closer to the length chosen by the RaPiD designers. The pseudocode for the algorithm appears below.

```

Add_Max_Once() {
  Let T be the set of tracks (initially empty)
  Let S = the set of all signals

  // add any feedback tracks that help unroutable cross-section
  nTracks = get_maxtracksbenefit(T, S, 0, LOCAL)
  Add nTracks feedback tracks to T
  route_signals(T,S)

  // add any length-2 local tracks that help unroutable cross-section
  nTracks = get_maxtracksbenefit(T, S, 2, LOCAL)
  Add nTracks local length-2 tracks to T
  route_signals(T,S)

  // add any length-4 local tracks that help unroutable cross-section
  nTracks = get_maxtracksbenefit(T, S, 4, LOCAL)
  Add nTracks local length-4 tracks to T
  route_signals(T,S)

  // add length-8 distance tracks to route remaining signals
  Let U be the set of unroutable signals
  While U not empty {
    Add one distance length-8 track to T
    route_signals(T,S) and update U
  }
}

```

Figure 7.10: Pseudocode for the main body of the Add Max Once flexible routing generation algorithm. The pseudocode for the `route_signals()` function appears in Figure 7.7, and the pseudocode for the `get_maxtracksbenefit()` function appears in Figure 7.11.

```

get_maxtracksbenefit(T, S, length, type) {
    // find the point at which adding more tracks of the given type
    // (distance or local) and given length does not improve
    // cross-section of unroutable signals

    Let U = the set of all unroutable signals
    Let maxTracks = |U|
    Let minCrossSection = cross-section of unroutable signals if
        maxTracks tracks of given length were to be added to T

    For all nTracks, maxTracks-1...0 {
        Let crossSection = cross-section of unroutable signals if nTracks
            tracks of given length were to be added to T
        If crossSection > minCrossSection {
            Return nTracks+1
        }
    }
    // cannot improve unroutable cross-section with this length
    Return 0
}

```

Figure 7.11: The pseudocode for a subfunction used by both the Add Max Once algorithm in Figure 7.10 and the Add Min Loop algorithm in Figure 7.13.

Add Min Loop

The previous algorithm tends to weight towards the use of distance routing tracks because it only considers each wire length and type combination once. However, it is possible that once a distance track is added, using additional local tracks will once again reduce the unroutable signal cross-section. Figure 7.12 gives an example where this is the case. In this example, AMO would not add a length-2 track for signal *s2* because that would not reduce the cross-section of unroutable signals—the cross-section would remain 1. Instead, AMO would implement all of the signals using two length-8 distance tracks, as shown in Figure 7.12b. However, the same problem could be solved using one length-8 distance track and one length-2 local track. Therefore, the Add Min Loop algorithm has been created in an effort to more accurately generate tracks with local wires. After

each track is added, this algorithm will return to the cheaper tracks to determine if any would be useful. Its pseudocode is presented in Figure 7.13.

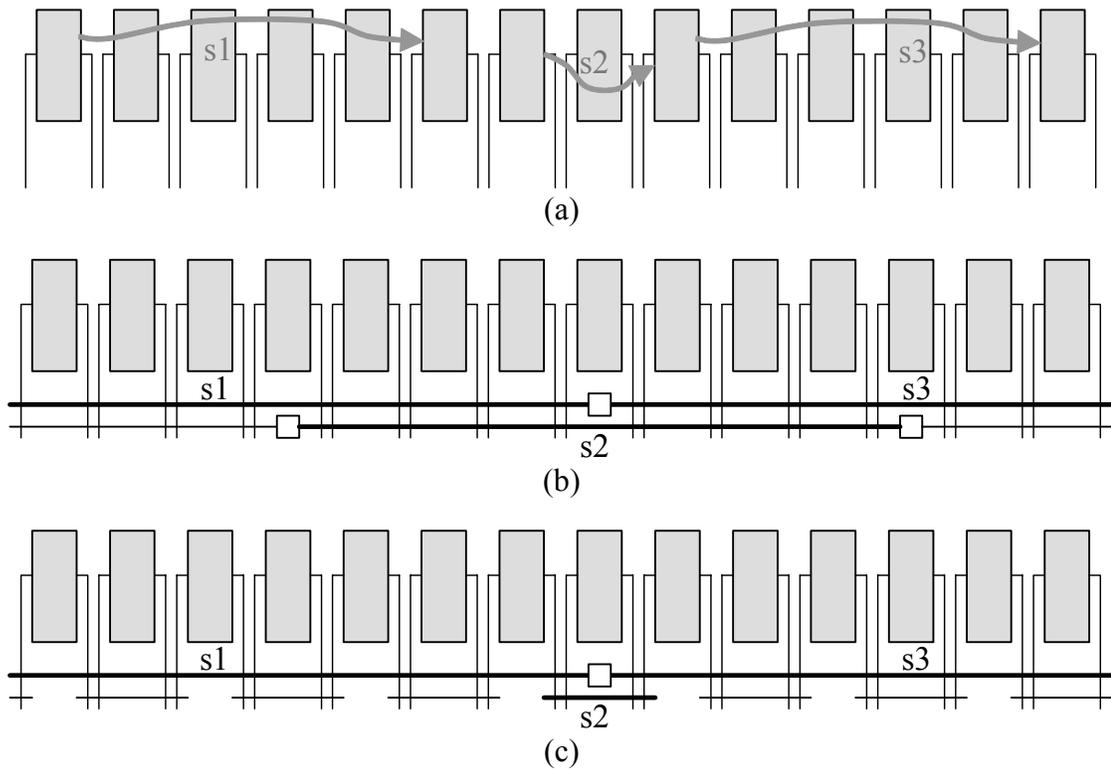


Figure 7.12: An example of AMO creating a more costly solution than necessary. (a) The routing generation problem, (b) the AMO solution, and (c) a less costly solution to the same problem.

This algorithm iteratively adds a small number of tracks to the overall routing architecture, until full routability can be achieved. Only one type of track is added per iteration. Within the loop, the algorithm repeatedly attempt to add tracks in the following order: length 2 local tracks, length 4 local tracks, length 16 distance tracks, and length 8 distance tracks. This order is based on an estimated relative cost of the various track types. Bus connectors consume area (priority is given to local tracks, followed by

distance tracks with fewer connectors), and short wires are faster than long ones (priority within local tracks is given to shorter lengths).

In the case of the local routing tracks, the algorithm attempts to add as many tracks as the length of the wires in the track (providing potentially the full range of offsets for that particular track type) per iteration. For distance routing tracks, which are considered to be much more expensive, only a single track can be added per iteration. Tracks are only kept if they result in a reduction in the unroutable cross-section. If any tracks are kept, all of the more “expensive” tracks are moved. So if a local track of length 4 is created, any tracks of length 8 or length 16 would be deleted. Once the “cheaper” track is added, the architecture may not need as many “expensive” tracks as was earlier computed. After the track counts are modified accordingly, execution begins again from the top of the loop, and the four track types are again considered in the same order as before.

At times, none of the track addition attempts will result in a reduction of the unroutable signal cross-section. In order to break this stalemate, the algorithm first looks at which of the attempts made would have reduced the count of unroutable signals the most. A single track of this type and wire length is created, and a new iteration begins. Note that there is a degree of favoritism in case of ties, where cheaper tracks are preferred over more expensive ones.

```

Add_Min_Loop() {
    Let T be the set of tracks (initially empty)
    Let S = the set of all signals

    Let nTracks = minimum number of feedback tracks required to get the
    maximum possible reduction in cross-section of unroutable signals
    Add nTracks feedback tracks to T

    Loop {
        route_signals()
        If all signals routed, break out of loop

        // try adding up to 2 length-2 local tracks
        nTracks = get_maxtracksbenefit(T, S, 2, LOCAL)
        If (nTracks > 2) nTracks = 2
        Add nTracks number of length-2 local tracks to T
        If (nTracks != 0) {
            Delete all local tracks from T where track length > 2
            Delete all distance tracks
            Jump to start of loop
        }

        // try adding up to 4 length-4 local tracks
        nTracks = get_maxtracksbenefit(T, S, 4, LOCAL)
        If (nTracks > 4) nTracks = 4
        Add nTracks number of length-4 local tracks to T
        If (nTracks != 0) {
            Delete all distance tracks
            Jump to start of loop
        }

        If adding one length-16 distance track reduces the cross-section
        of unroutable signals {
            Add one length-16 distance track to T
            Delete all distance tracks from T where track length < 16
            Jump to start of loop
        }

        If adding one length-8 distance track reduces the cross-section of
        unroutable signals {
            Add one length-8 distance track to T
            Jump to start of loop
        }

        // if algorithm gets here, none of the ones tried reduced the
        // cross-section of unroutable signals
        Let length, type = length and type of track from above tests that
        produced a solution with the fewest unroutable signals. If a tie,
        choose based on the order they were considered
        Add one track of that length and type to T
    }
}

```

Figure 7.13: The pseudocode for the Add Min Loop algorithm. The pseudocode for the `route_signals()` function appears in Figure 7.7, and the pseudocode for the `get_maxtracksbenefit()` function appears in Figure 7.11.

7.4 Results

The next few sections present comparative results of flexible routing architecture generation algorithms. First, the area results of the algorithms are compared to each other. Next, the resulting areas are also compared to those of the corresponding standard cell, FPGA, and RaPiD reference architectures. Finally, an initial flexibility comparison is made between the algorithms, with a more in-depth examination presented in Chapter 9.

7.4.1 Area

The same applications that were used for the cASIC comparisons of section 6.3 are used here to compare the flexible architectures. These applications are listed in Table 4.1. The standard cell (Table 4.2), FPGA (Table 4.3) and RaPiD (Table 4.4) reference architecture results from section 6.3 are also presented here for context. The areas of the flexible architectures are computed using the same methods as the cASIC area computations. Logic area is the sum of the areas of the manually-designed unit layouts; routing area is the sum of the areas of the layouts of the multiplexers, demultiplexers, and bus connectors. Again, wire cross-sections of greater than 24 can increase the height of the architecture beyond the height required by the logic units, and therefore increase the total area of the architecture.

Table 7.1: A table of the number of routing tracks created for each application by each routing generation algorithm. A lower bound is also given, and the factor of the bound is given for every case.

		Radar	OFDM	Camera	Speech	FIR	Matrix	Sort	Image	Average Factor
GH	Actual	17	34	24	25	18	24	21	21	1.86
	Bound	11	18	13	13	11	10	11	12	
	Factor	1.55	1.89	1.85	1.92	1.64	2.40	1.91	1.75	
AMO	Actual	18	28	25	24	17	21	21	17	1.55
	Bound	13	21	16	17	11	10	13	12	
	Factor	1.38	1.33	1.56	1.41	1.55	2.10	1.62	1.42	
AML	Actual	18	32	26	27	18	18	21	20	1.61
	Bound	13	21	16	17	11	10	13	12	
	Factor	1.38	1.52	1.63	1.59	1.64	1.80	1.62	1.67	

Table 7.1 lists the number of routing tracks created by each flexible routing generation algorithm, while Figure 7.14 charts the area results. A lower bound on the number of tracks is also given for each application set for each algorithm. This lower bound is based upon the signal cross-section of the placement. Since the Greedy Histogram (GH) method permits the physical units to move during placement, it results in a “better” placement that generally has a smaller signal cross section. On the other hand, the Add Max Once (AMO) and Add Min Loop (AML) algorithms have the same lower bound, as they use the same placement techniques. Each of the algorithms result in a number of tracks within a factor of 1.5 to 2 of the lower bound. However, in order to actually achieve a track count reaching the lower bound, a large number of bus connectors would likely be required in order to provide such a high degree of wire sharing between signals from different netlists. It is unlikely, especially as the number of dissimilar netlists in the application increases, that an efficient routing structure would result.

Figure 7.14 graphs the area of the generated flexible architectures for eight different domains. This figure indicates that while the areas of the architectures created by the three different algorithms are somewhat close, there are some notable differences. First, the Greedy Histogram algorithm has shown to have some unexpected results. This algorithm was intended to create architectures less flexible, but more optimized, than the two regular routing algorithms. In most cases, this is in fact true. However, there are also a number of applications for which the Greedy Histogram method produces architectures that are actually larger than those created by the other algorithms. One likely explanation for this behavior is that the Greedy Histogram algorithm does not consider the “big picture” when making decisions. For example, if the histogram indicates that the most common signal length is 11, and the second-most common signal length is 12, the algorithm will create a track of length 11, without considering that a length-12 track is also able to implement length-11 signals. This could lead to the creation of more tracks than necessary.

Generally, AMO results in an architecture with fewer tracks, but more bus connectors than AML (which emphasizes the use of local tracks when possible). However, in an effort to use fewer bus connectors, the AML algorithm may actually create more tracks than the AMO algorithm. These issues are reflected in the area results. For the Radar and Sort applications, both AMO and AML create architectures with the same number of tracks. However, the area of the AMO architectures is slightly higher due to a greater number of bus connectors. This is even more noticeable for the

Speech and Camera applications where the AML algorithm actually creates more tracks than the AMO algorithm, but still results in smaller architectures. In other cases, where the area of the AML algorithm is higher than that of the AMO algorithm, the AML algorithm created a significantly greater number of tracks, increasing the width of multiplexers and demultiplexers, and in the case of OFDM, also increasing the height of the architecture. A future generation algorithm that considers the area tradeoff between bus connectors and number of tracks could result in smaller architectures than AMO and AML in most if not all cases.

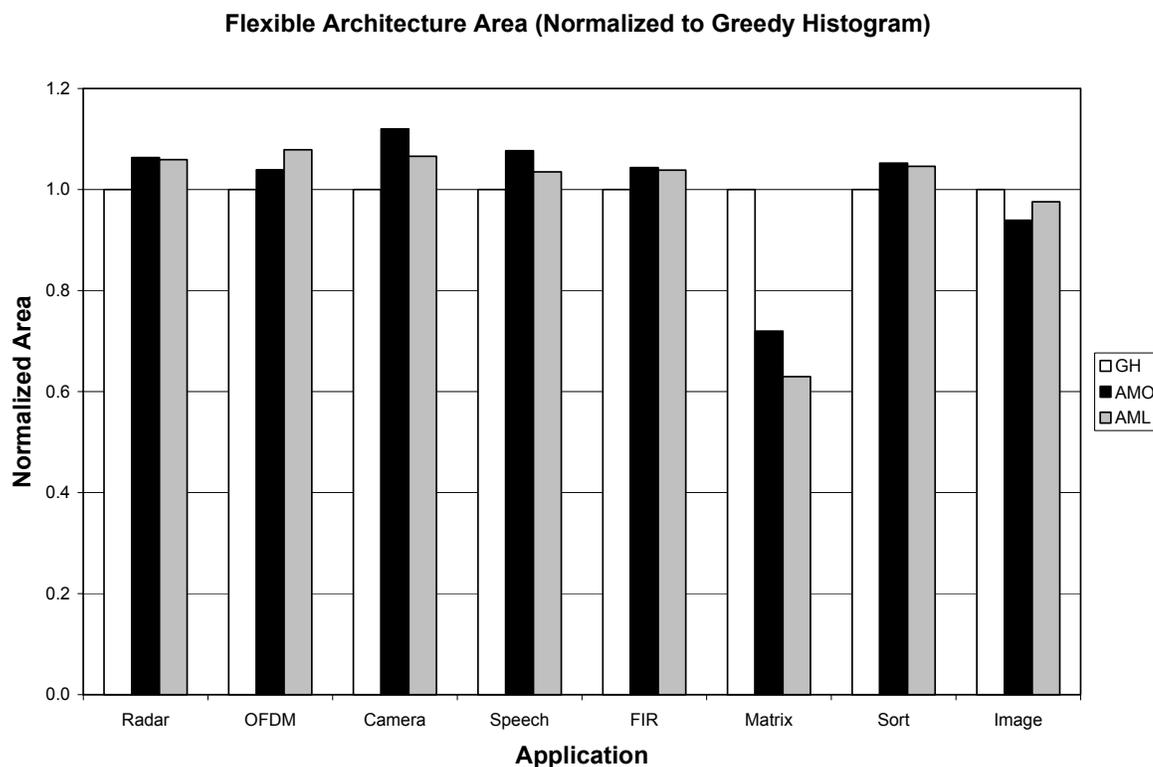


Figure 7.14: Comparative area results of the different flexible routing generation algorithms, Greedy Histogram (GH), Add Max Once (AMO), and Add Min Loop (AML). The areas have been normalized to the Greedy Histogram result.

Table 7.2: The areas, in mm², of the eight different applications from Table 4.1, as implemented with the three flexible routing architecture generation algorithms. The results from Table 4.2 though Table 4.4 are included for reference. A summary of these results appears in Table 7.3.

		Radar	OFDM	Camera	Speech	FIR	Matrix	Sort	Image
Std. Cell	Total	4.101	9.168	7.268	26.523	2.846	1.785	1.541	6.843
FPGA	Total	19.719	59.157	23.006	78.877	26.292	19.719	26.292	19.719
RaPiD	Logic	2.838	---	---	45.401	3.783	1.892	2.838	---
	Routing	2.158	---	---	34.536	2.878	1.439	2.158	---
	Total	4.996	---	---	79.937	6.661	3.331	4.996	---
Clique Overlap	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	0.086	0.456	0.297	0.261	0.262	0.140	0.294	0.216
	Total	1.520	4.574	2.475	13.010	2.004	1.264	1.487	1.833
GH	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	1.271	15.896	5.651	22.202	1.737	2.138	2.110	2.302
	Total	2.705	20.014	7.829	34.950	3.478	3.261	3.303	3.919
AMO	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	1.443	16.682	6.591	24.886	1.888	1.224	2.283	2.065
	Total	2.877	20.800	8.768	37.635	3.630	2.347	3.476	3.681
AML	Logic	1.433	4.118	2.178	12.748	1.742	1.124	1.193	1.617
	Routing	1.431	17.470	6.168	23.430	1.870	0.930	2.262	2.208
	Total	2.865	21.588	8.346	36.179	3.612	2.053	3.455	3.825

Table 7.3: A summary of area comparisons between the different implementation techniques. Area improvements are calculated over the heading implementation, then averaged across all applications.

Improvement Over Std Cells		Improvement Over RaPiD		Improvement Over FPGA		Improvement Over AMO	
Method	Area	Method	Area	Method	Area	Method	Area
FPGA	0.20	Std Cells	2.34	Std Cells	7.20	Std Cells	1.37
RaPiD	0.48	FPGA	0.38	RaPiD	4.01	FPGA	0.24
Clique (O)	2.16	Clique (O)	3.75	Clique (O)	12.30	RaPiD	0.60
GH	0.90	GH	1.72	GH	5.25	Clique (O)	2.61
AMO	0.91	AMO	1.71	AMO	5.37	GH	1.01
AML	0.92	AML	1.77	AML	5.53	AML	1.02

Table 7.2 lists the areas of the architectures created using the different flexible routing generation heuristics, with the corresponding standard cell, FPGA, and RaPiD areas listed for comparison. These results are summarized in Table 7.3. As expected, the customized flexible architectures are smaller than the corresponding FPGA

implementations—from a 5.3x to a 5.5x area improvement. These results highlight the area benefits of optimized reconfigurable computation and routing structures.

For the applications where a RaPiD implementation was possible, the flexible architectures ranged from 56%-58% of the required RaPiD area. When RaPiD was not able to implement an application, the flexible generation algorithms were still able to create an architecture. Again, this is a key feature of the Totem automatic architecture generation algorithms—both the logic and routing resource mix can be customized to the needs of the specification, and are not limited by a static design.

Surprisingly, the flexible architectures were also on average close in area to the corresponding standard cell layouts. One of the key benefits of reconfigurable architectures is that expensive hardware resources can be reused across netlists. While the added flexibility of segmented routing tracks does significantly increase the flexible architecture areas over those of cASIC designs, the results indicate this area increase does not overwhelm the area savings of hardware reuse.

7.4.2 Flexibility

In addition to area, the algorithms have also been compared on the basis of the flexibility of the architectures that they generate. The tests in this section involve examining architectures that were designed for one application, and attempting to implement a non-member netlist onto that architecture. The results for these tests are shown in Table 7.4. For each architecture created, all 26 of our netlists are placed and

routed onto the generated architectures. If a netlist failed placement and/or routing, it was attempted on a larger architecture, where the quantity of logic resources was increased by 10 or 20 percent, as indicated in the table. An architecture with a greater number of logic resources can sometimes allow for an improved placement, which in turn can result in an easier routing operation.

Naturally, the larger benchmarks tended to generate architectures more capable of implementing the other benchmarks. For example, the Camera and Image applications were able to implement far more netlists than the Matrix or Sort applications. The Sort application has the added difficulty that absolutely no multiplier units were required by the benchmarks used to create the architectures. Therefore, increasing the logic on a percentage scale does not introduce any multipliers. All benchmarks that fail to place and route onto this architecture require at least one multiplier.

Table 7.4: Initial flexibility study of the generated architectures. All 26 available netlists were tested on all generated architectures. When necessary, the logic in the architecture is increased on a % basis to attempt to fit the benchmark. The rows of this table indicate how many benchmarks are source netlists for the architectures (SRC), how many will P&R based on a percent increase in logic (0%, 10%, or 20% increase), and how many will fail altogether.

	Radar			Camera			OFDM			Image			DCT/FFT			FIR			Matrix			All Sort		
	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML	GH	AMO	AML
SRC	3	3	3	3	3	3	2	2	2	5	5	5	3	3	3	6	6	6	5	5	5	4	4	4
0	8	8	8	18	18	18	16	16	16	11	11	11	9	9	9	5	5	5	3	4	4	1	1	1
10	0	0	0	2	2	2	1	1	1	4	4	4	0	0	0	7	7	7	1	0	0	0	0	0
20	0	0	0	1	1	1	0	0	0	0	0	0	3	3	3	0	0	0	0	0	0	0	0	0
Fail	15	15	15	2	2	2	7	7	7	6	6	6	11	11	11	8	8	8	17	17	17	21	21	21

In most cases, when sufficient physical logic is present to implement a netlist, a successful routing can also be found. However, one of the FIR filter netlists will fail to

route on the Matrix Multiplier architecture generated by the Greedy Histogram method, even though that architecture has sufficient resources to implement the logic of the netlist. Increasing the logic by 10% allows the netlist instances to be placed onto the architectural components in a more routable fashion. Both Add Max Once and Add Min Loop create distributed routing structures, which may contribute to their ability to implement that particular FIR filter on their respective Matrix Multiplier architectures without an increase in logic.

7.5 Summary

This chapter described three different algorithms used to generate flexible Totem architectures. One, Greedy Histogram, was intended to be highly optimized to the specification set. However, this algorithm could be further refined. The two other algorithms, Add Max Once, and Add Min Loop, generate architectures with a high degree of regularity, which can lead to a greater ability to implement netlists different in structure than those in the architecture specification. While this flexibility was hinted at in the results of section 7.4.2, it will be more strongly demonstrated in Chapter 9. The AMO algorithm focuses on employing segmented distance tracks to reduce total track count, while AML instead aims to reduce the number of bus connectors at the expense of additional tracks. The area results reflect these goals, as AMO nearly always resulted in a lower track count, but AML on average resulted in a lower area. A future algorithm may incorporate both goals to achieve the lowest area of the flexible algorithms.

Much like the previous chapter, the area comparisons presented here demonstrate the area benefits of customization. The flexible Totem architectures were on average 5.3-5.5x smaller than the equivalent FPGA implementations. While FPGAs are important for situations in which one cannot reliably predict how the hardware will be used, they are comparatively inefficient for circumstances when some or all characteristics of the target applications are known. While the RaPiD project addresses this issue to some degree by employing coarse-grained computational units, it allows only a limited degree of application- or domain-customization by varying the number of cells.

An architecture generation tool such as presented here can be used to capture further optimizations, creating architectures nearly half the area required by RaPiD. Furthermore, the generation algorithms are not limited to one particular design, and are therefore able to create architectures large enough to implement applications which will not fit in the current RaPiD design. Finally, customized reconfigurable architectures, as demonstrated in this chapter, can sometimes even support applications in an area comparable to that required by standard cell implementations, due to the efficient layout of the coarse-grained logic structures, as well as the ability to reuse resources across the netlists.

Chapter 8

Track Placement

The design of an FPGA interconnect structure has usually been a hand-tuning process. A human designer, with the aid of benchmark suites and trial-and-error, develops an interconnect structure that attempts to balance flexibility with silicon efficiency. Often, such as in the previous chapter, the concentration is on picking the number and length of tracks – long tracks give global communication but with high silicon and delay costs, while short wires can be very efficient if signals travel a relatively short distance.

A topic that can sometimes be ignored is the placement of the breaks between wires in the routing tracks. If there is a symmetric interconnect, with N length- N wires, a break is simply placed at each position. However, for more irregular interconnects, it can be difficult to determine the best positioning of these breaks. Although the break positioning can have a significant impact on the routability of an interconnect structure, there has been little insight into how to quantify this effect, and optimize the break positioning.

While a manual solution may be feasible in many cases when examining only a single architecture, it is not always practical. This particular work is motivated by the automatic generation of FPGA architectures for systems-on-a-chip. Here a track placement may be performed a very large number of times within the inner loop of an architecture generator, such as presented in Chapter 7, and a fast but effective algorithm for automatic track placement becomes a necessity.

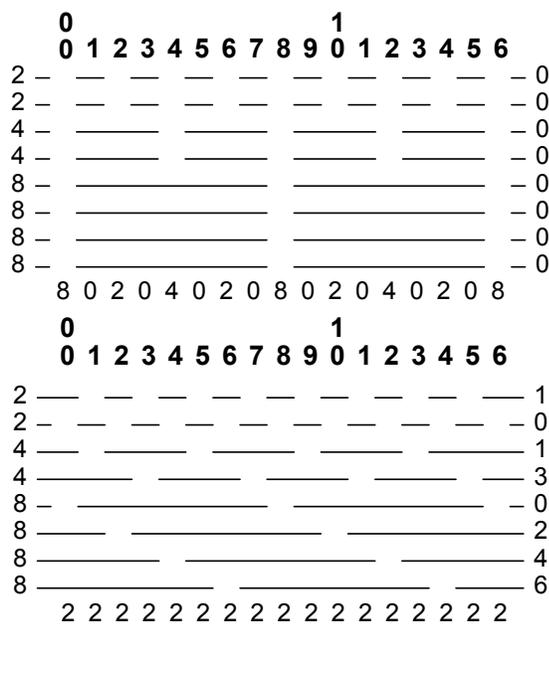


Figure 8.1: Two different track placements for the same type of tracks, (a) a very poor one and (b) a very good one. In each, the numbers on top indicate position in the architecture, and the numbers on the bottom indicate the number of breaks at the given position. Each track has its associated wire length at left, and offset at right.

Achieving the best track placement requires a careful positioning of the breaks on multiple, different-length, routing tracks. For example, in Figure 8.1b, the breaks between wires in the routing tracks are staggered. This helps to provide similar routing

options regardless of location in the array. If instead all breaks were lined up under a single unit, as in Figure 8.1a, a signal might become very difficult to route if its source was on one side of the breaks and at least one sink on the other. For large numbers of tracks of different wire lengths, it is challenging to find a solution where the breaks are evenly distributed through the array.

Determining the left-right shifting, or offset, of a track within an architecture is referred to as track placement. The goal, given a predetermined set of tracks with fixed length wires, is to pick an offset for each track in order to maximize the routability of the architecture. For simplicity, each track is restricted to contain wires of only one length, which is referred to as the track's S value, or track length. The actual determination of the quantity and S values of tracks is performed during the architecture generation from Chapter 7. Additional information specific to 2D FPGA routing architecture design can be found elsewhere [Betz00, Lemieux02, Lemieux03].

This chapter discusses the issue of track placement for reconfigurable architectures with segmented channel routing (Figure 8.2), such as RaPiD [Cronquist99a], Chimaera [Hauck97], Garp [Hauser97], and Triptych [Borriello95]. First, the track placement problem and a cost metric are defined. Next, several track placement algorithms, including one proven optimal for a subset of these problems, are introduced. Finally, the track placement algorithms are related to the flexible reconfigurable architecture generation of Chapter 7. Further details on the track placement problem can be found elsewhere [Compton02b].

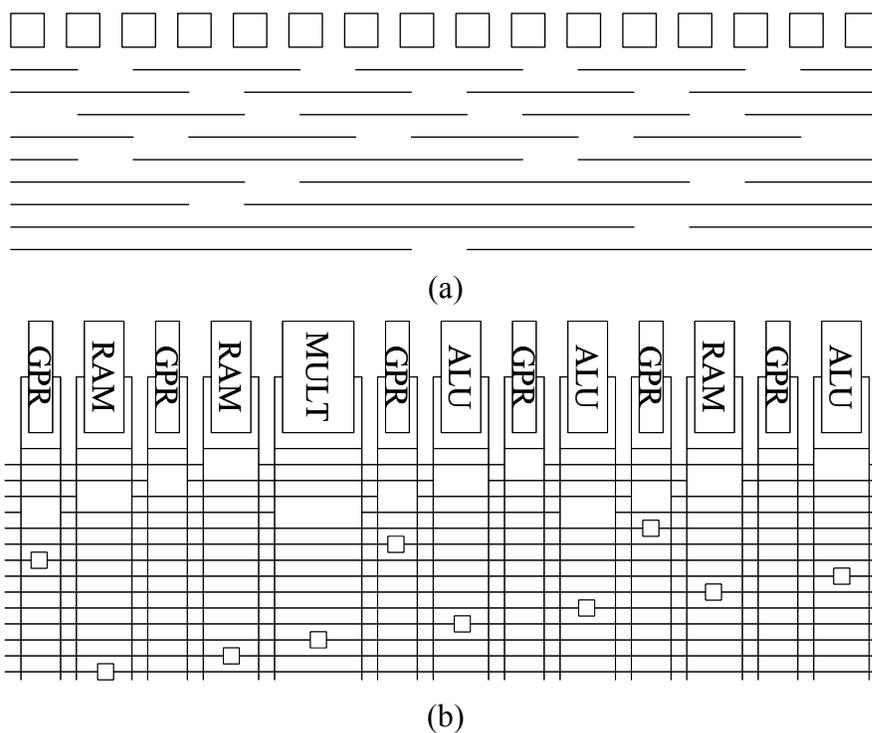


Figure 8.2: Two examples of reconfigurable architectures with segmented channels. At top, (a) Garp [Hauser97]. At bottom, (b) RaPiD [Cronquist99a].

8.1 Problem Description

Finding a good track placement is a complex task. Intuitively, tracks with the same S value should be spaced evenly, such as by placing the length-8 tracks from the problem featured in Figure 8.1 at offsets 0, 2, 4, and 6. However, a placement of tracks of one S value can affect the best placement for tracks of another S value. For example, Figure 8.1b shows that the length-4 tracks are placed at offsets 1 and 3 in order to avoid the break locations of the length 8 tracks. This effect is called “correlation” between the

affected tracks. Correlations occur between tracks when their S values contain at least one common prime factor. It is these correlations that make track placement difficult.

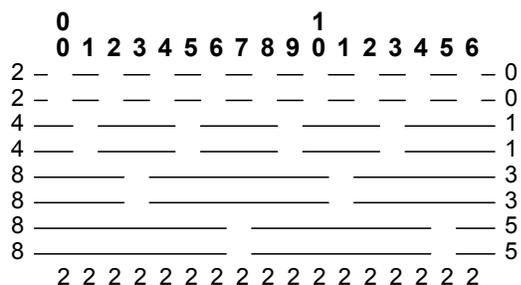


Figure 8.3: An alternate placement for the architectures in Figure 8.2 that maintains perfect smoothness of breaks, but is intuitively less routable than the placement in Figure 8.2b.

From the previous example, one might conclude that the “evenness” or “smoothness” of the break distribution could be a possible metric for measuring the quality of a track placement would be to compute. However, the smoothness metric fails to capture the idea of maintaining similar routing options for every location in the array. The placement in Figure 8.3 has the same smoothness of breaks as the solution in Figure 8.1b, but is not an equally good solution. In Figure 8.3, although there are two length-4 tracks, each logic unit position is at the same location along the length-4 wires in both tracks. On the other hand, the architecture in Figure 8.1b provides for two different locations along the length-4 wires at every logic unit position. In Figure 8.1b, a logic unit at position 8 can reach as far left as position 6 or as far right as position 11, depending on which length-4 track is used. The logic unit at position 8 in Figure 8.3 can only route as far left as position 6 and as far right as position 9 regardless of which length-4 track is

the problem. Examining a larger window would simply yield the same results. Within this window, the number of tracks at each position that can be used to route a signal of the given length towards the right is counted. The different possible signal lengths (L) are listed, and at each position, the number of tracks useable to route that signal length is given.

In Figure 8.4a, two different tracks can be used to route length-2 signals to the right at position 6, but at position 5 no tracks are useable. At right, exactly one track can be used to route length-2 signals from any position in our window. The minimum value at each length is found (representing a routing bottleneck for that signal length), and these minimums are summed across all L values to determine the diversity score of a track placement. The fewer and smaller the routing bottlenecks, the more routing choices are available throughout the architecture, and the higher the diversity score. An equation for the diversity score is given as Equation 8.1. In this equation, T represents the set of tracks to be placed, and O represents the set of offsets assigned to those tracks.

Equation 8.1: The diversity score for a particular track placement routing problem T and solution set of offsets O for the tracks in T can be calculated using the equation:

$$diversity_score(T, O) = \sum_{all\ L} \left(\min_{all\ positions} \left(\sum_{T_i \in T} uncut(T_i, O_i, L, position) \right) \right)$$

where T_i is an individual track and O_i is the offset for that track. As the equation indicates, the minimum is found for all positions, and the sum is calculated of all the minimums at each L value.

The *uncut()* function returns a binary result that is 0 if there is a break within the range [position, position + L) on track T_i that has been placed at offset O_i , and 1 otherwise.

A bound has been determined on the diversity score, as described in Theorem 8.1 (proven elsewhere [Compton02b]). Note that comparing diversity scores is only valid across different solutions to the same track placement problem. The method of determining the actual quantity and types of tracks was given in Chapter 7. The placement algorithms that will be presented are intended to be used within an inner loop of the flexible architecture generation algorithms.

Theorem 8.1: For all possible offset assignments O to track set T ,

$$diversity_score(T, O) \leq \sum_L floor \left(|T| - \sum_{T_i \in T} \min(1, L / S_i) \right)$$

The focus is on the worst-case (the regions with the fewest possible routes) instead of the average case, since the average number of possible routes for a signal is independent of track placement. Changing the offset of a track in a given placement only shifts which positions can use that track for the different signal lengths. If the length-2 track at offset 0 in Figure 8.2 were shifted to offset 1, the two length-1 signals that that track may implement within the window would originate at positions 6 and 8 instead of 5 and 7. The length-1 count of uncut signals becomes “0 4 2 4” at positions 5 through 8, respectively. Regardless of the amount of the shift, the same number of breaks remains within the window, and therefore the same number of signals of each length can be routed—only at different starting positions. Shifting the track does not affect the sum (and therefore the average) for each signal length. While the average cannot differentiate

between the placements in Figure 8.2 and Figure 8.3, the worst-case shows that the placement in Figure 8.2b is superior.

8.2 Track Placement Algorithms

A number of algorithms, of varying complexity, have been developed to solve the track placement problem based on the diversity score cost function. These track placement algorithms, both optimal and heuristic, are discussed in depth in the next few sections. The first algorithm, Brute Force, simply tests all possible track placements, and returns a solution with the highest diversity score. This algorithm, however, runs extremely slowly and is impractical for large problems, large architectural explorations, or automatic architecture generation. The next algorithm, Simple Spread, treats sets of tracks at each S value as separate problems, spacing tracks evenly within the sets. While this can provide a fast solution, it does not consider the potential correlations between tracks of different segment lengths. Power2 is another very simple algorithm which does consider inter-set effects, but is restricted to handle tracks with segment lengths that are powers of two.

The final two algorithms are more complex in operation than the other algorithms, yet require significantly less computation time than the Brute Force algorithm. Pseudocode for is given for these two algorithms. The Optimal Factor algorithm provably finds an optimal solution based on diversity score, provided the problem meets a number of key restrictions. The Relaxed Factor algorithm is an extension of the

Optimal Factor algorithm that can operate without restrictions. While the Relaxed algorithm is not guaranteed to provide an optimal solution in every case, it provides very high quality results in most cases.

8.2.1 Brute Force Algorithm

Using a brute-force approach, a solution with the highest possible diversity score is guaranteed to be found. However, as with many complex problems, examining the entire solution space is a very slow process. While useful for finding the best possible solution for a final design, this method is inappropriate for situations where a large number of different routing architectures (each with different types and or quantities of tracks) are being examined. The number of cases that the brute force approach must examine for a given problem is given in Equation 8.2.

Equation 8.2: The number of distinct cases that must be examined by the Brute Force algorithm in order to have complete coverage of the solution space. This is the product over all distinct track lengths in the problem of a multi-choose of track length and quantity:

$$\prod_s \binom{(S_i - 1 + Q_i)!}{(S_i - 1)! Q_i!}$$

where S is the set of all distinct track lengths in the problem, S_i is a particular track length, and Q_i is the number of tracks with that track length.

For example, a modest architecture with 8 length-12 tracks, 4 length-6 tracks and 2 length-4 tracks will require the examination of over *95 million distinct cases* using the brute force approach. Given that a track placement algorithm may operate within an inner loop of architecture generation, this approach is far too slow. Instead, it is used to

provide a bound on the diversity score. This bound is then used to experimentally verify the results of an optimal algorithm, and provide a frame of reference for the solution quality of the other algorithms.

8.2.2 Simple Spread Algorithm

As stated previously, the Simple Spread algorithm is a very straightforward heuristic for performing track placement. In this particular algorithm, tracks are grouped by segment length. Each track group is considered as a separate problem. Each group first has any “full sets” (where the number of tracks is equal to the segment length) placed with one track at each possible offset for that particular S value. Once the number of tracks in a set is less than the S value for the set, these remaining tracks are spaced evenly throughout the possible offsets 0 through S-1 using the equation $O_i = \text{floor}(S*i/|G|)$, where G is the set of remaining tracks of segment length S, i is the index $0 \dots |G|-1$ of a particular track in the set, and O_i is the calculated offset of said track.

This algorithm, while simple in its operation and fast in execution, disregards the fact that a placement decision at one S value can affect the quality of a placement decision at another due to correlations. Figure 8.5 shows a small case in which the Simple Spread returns a track placement significantly worse than the Brute Force optimal placement. The diversity score for the Simple Spread solution is only 6, whereas the Brute Force optimal solution has a diversity score of 9. Looking at the number of breaks occurring at each offset (the bottom row of numbers on each of the diagrams), the

solution found by the Brute Force method is significantly smoother than the one found by Simple Spread, which leads directly to the difference in diversity scores. This equation is provably optimal for a single track length in isolation, so less-than-optimal results are a direct result of interactions between the different S values in a track routing problem, which demonstrates the importance of correlations.

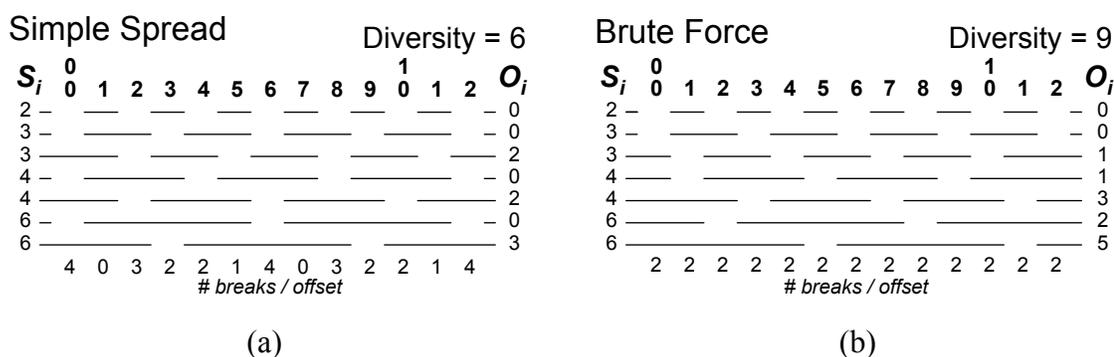


Figure 8.5: A track placement problem solved using (a) Simple Spread solution and a (b) Brute Force solution. For each diagram, the top row of numbers indicates the position, and the bottom row indicates the number of breaks occurring at the corresponding position. Each track is labeled on its left with its S value and on its right with its assigned offset. The diversity score of each solution is also given in the upper right-hand corner.

8.2.3 Power2 Algorithm

The Power2 algorithm, unlike Simple Spread, places more importance on compensating for the correlation between the S values of tracks than on the even spreading of tracks within a particular S value. This algorithm, however, is restricted to situations where all tracks have S values which are a power of 2. In this case, the offset for a particular track is calculated based on the offsets and S values of all previously placed tracks. While initially this may seem a complex task, it has been simplified such

that only the placement of the track immediately previous to the current track must be considered.

First, the tracks in a placement problem are grouped by S value, and these groups are sorted in increasing order. Offsets are then assigned in a particular constant pattern dependent upon the S value. The example of Figure 8.6 gives the patterns for S=2, S=4, and S=8, while Equation 8.3 explains how these patterns are determined. The tracks in the group with the smallest S are assigned using the pattern for that S value. Afterwards, the next S group, with its particular pattern, is placed, starting with the offset value that would have been next in the previous pattern. In Figure 8.6 after the length-4 track has been placed at offset 1, the length-8 tracks resume placement offset 3, the next offset in the length-4 sequence. The remainder of the length-8 tracks follow the length 8 pattern after that point. This algorithm attempts to maintain a somewhat even density of breaks throughout the array. This does mean that the breaks of a particular S group may not be evenly distributed, such as when the quantity of tracks in a group is not a power of 2, but the goal of the algorithm is to use the correlations between different S groups to compensate. This is the algorithm used by the regular track placement algorithms in Chapter 7.

Equation 8.3: The recursive calculation of the pattern of offsets for the Power2 track placement algorithm. The pattern for a given *length* is dependent on the pattern for the previous length *prev* (both restricted to be powers of 2, so if *length* is 16, *prev* is 8).

$$Pattern_{length}[x] = Pattern_{prev}[x \bmod 2] * 2 + \left\lceil \frac{x}{length/2} \right\rceil, \text{ for } x = 0 \dots length-1.$$

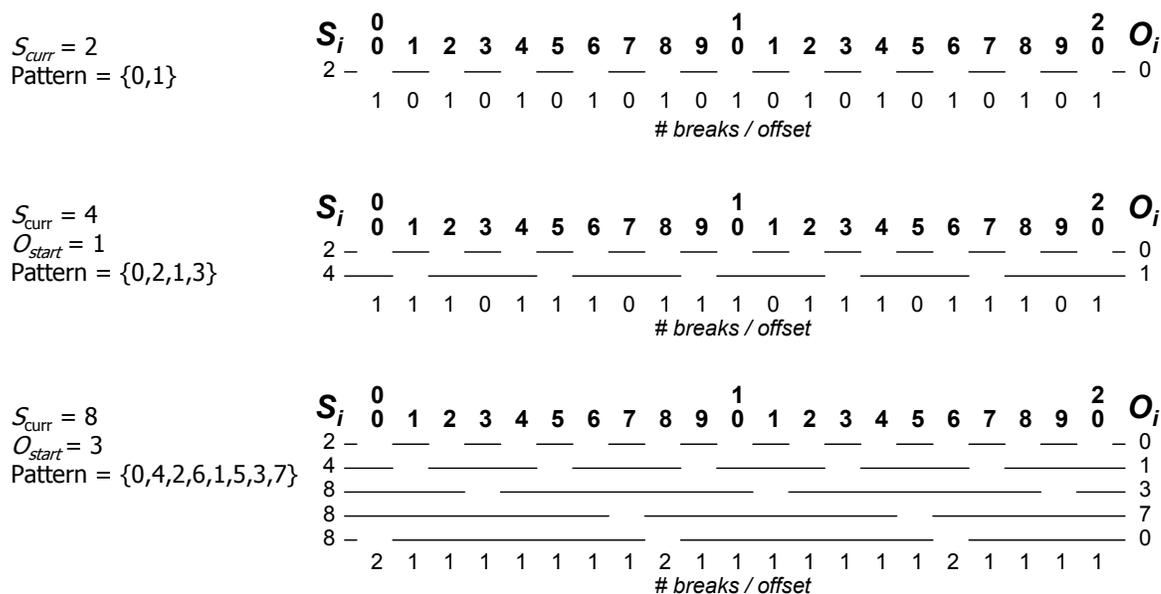


Figure 8.6: An example of the operation of Power2 at each of three S values for a case with one length-2 track, one length-4 track, and three length-8 tracks. The offset pattern at each S value is given, along with the offset O_{start} to use for the first track in that S group based on the placement of the previous tracks. For each diagram, the top row of numbers indicates the position, and the bottom row indicates the number of breaks occurring at the corresponding position. Each track is labeled on its left with its S value and on its right with its assigned offset.

8.2.4 Optimal Factor Algorithm

A fast algorithm has been developed that, with some restrictions, provably finds the optimal solution. The Optimal Factor algorithm is the culmination of many theorems and proofs. While many of the theorems will be presented here, their proofs can be found elsewhere [Compton02b]. This section presents various track placement situations, theorems regarding their solutions, and finally the pseudocode for the Optimal Factor Algorithm itself.

As stated previously, correlations across S values can occur when those S values share a common factor. Any optimal algorithm must therefore consider the correlations

between breaks not only within a particular S group, but also across S groups. The example of Figure 8.5 illustrates the importance of correlations, and the fact that it is transitive. Because the S=2 group and the S=6 group are correlated (sharing a common factor of 2), there is also a correlation between the S=2 group and the S=3 group, and consequently, between the S=3 group and the S=4 group. But if an additional track group of S=5 were to be introduced into the problem, it would not share a factor with any other track group, and would not introduce further correlations. Regardless of offset choice, the length-5 track would overlap with breaks from each of the other tracks at various positions in the architecture. Therefore, it can be placed independently of the other tracks. This feature is embodied in Theorem 8.2, which is used to divide a track placement problem into smaller independent (uncorrelated) problems when possible.

Theorem 8.2 If T can be split into two sets $G1 \subset T$ and $G2 = T - G1$, where the S values of all tracks in G1 are relatively prime with the S values of all tracks in G2, then $diversity_score(T,O) = diversity_score(G1,O1) + diversity_score(G2,O2)$, where O is the combination of sets O1 and O2. Thus, G1 and G2 may be solved independently.

The fact that correlation is based on common factors can be used to further simplify the track placement problem. Theorem 8.3 states that whenever one track's S value has a prime factor that is not shared with any other track in the problem (or a track has a higher quantity of a prime factor than any other track in the problem), the prime factor can be effectively removed. After all "extra" prime factors have been removed, the remaining prime factors are all shared by at least two tracks. This can make interactions between tracks more obvious. For example, if there are 2 length-6 tracks and one length-

18 track, a 3 can be removed from 18, and then in effect there are 3 length-6 tracks, which can be placed evenly using a later theorem (Theorem 8.6). Note that the S value of the length-18 track is not permanently changed, just the S value used during the course of track placement. After the offsets of tracks are determined, the original S values are restored if necessary.

Theorem 8.3: If the S_i of unplaced track T_i contains more of any prime factor than the S_j of each and every other track T_j ($i \neq j$), then for all solutions O , $diversity_score(T, O) = diversity_score(T', O)$, where T' is identical to T , except T'_i has $S'_i = S_i$ with the unshared prime factor removed. This rule can be applied recursively to eliminate all unshared prime factors.

For the rest of the optimal algorithm, the tracks are grouped according to S value (where the S value may have been factored using Theorem 8.3). These groups are then sorted in decreasing order, such that the largest S value is considered first. Any full sets are removed using Theorem 8.4, where a full set is defined as a set of N tracks all with $S_i = N$. In other words, when there are enough tracks to fill all potential offset positions $0 \dots N-1$. If the remainder of the track placement problem can be solved to meet the bound of Theorem 8.1, then an overall placement that also meets the bound places one track from the full set at each potential offset position for the S value of that set.

Theorem 8.4: Given a set $G \subset T$ of tracks, each with $S=|G|$. If there exists a solution O' for tracks $T'=T-G$ that meets the bound of Equation 8.1, then there is also solution O for tracks T that meets the bound (and thus is optimal), where each track in G is placed at a different offset $0 \dots |G|-1$.

Note that throughout this algorithm, as well as all of the others, only offsets in the range 0 through $|S_i|-1$ are considered for a track with $S=S_i$. While it is perfectly valid to place a track at an offset greater than S_i-1 , the fact that all wires within the track are of

length S_i causes a break to be located every S_i locations. Therefore, there will always be a break on track T_i within the range 0 to S_i-1 , which can be found as dictated in Theorem 8.5.

Theorem 8.5: If a track T_i has a break at position P , it will also have a break at position $P \% S_i$. Therefore, all possible offset locations for track T_i can be represented within the range 0 through S_i-1 (inclusive).

The optimal algorithm uses Theorem 8.4 and Theorem 8.6 to space the tracks of the largest S value (S_{\max}) evenly. Then the tracks with the next largest S value (S_{next}) must be placed evenly. In order to find the best offsets for the second (and successive) groups of tracks, the interaction between the breaks of the previously placed tracks and those of the new tracks must be considered. Therefore, the optimal algorithm examines the breaks of the previous tracks in terms of the next S value (S_{next}) in order to determine which offsets should be assigned to the next group of tracks to avoid bottlenecks. The previously placed tracks are converted to a set of temporary “placeholder” tracks with $S=S_{\text{next}}$, and with each real break modeled by a break in the same position from a placeholder track, as demonstrated by the example of Figure 8.7. This enables the algorithm to accurately determine the best offsets for the real tracks for the current S value. This process is repeated within a loop, where S_{next} now becomes S_{\max} (since the tracks of the old S_{\max} have already been placed), and a new S_{next} is found below the new S_{\max} .

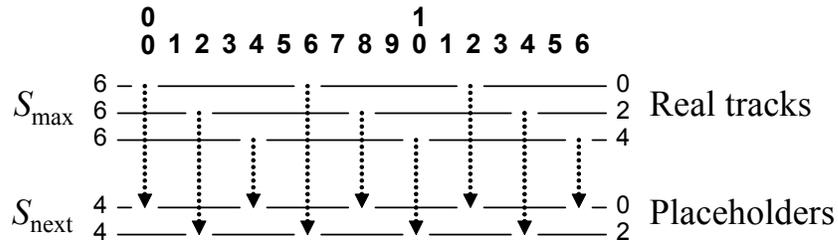


Figure 8.7: The breaks from tracks of length S_{max} are emulated by the breaks of placeholder tracks of length S_{next} for the next iteration.

There are a number of restrictions, however, in order to ensure that at each loop iteration, (1) the tracks with $S=S_{max}$ can always be evenly spaced, and (2) the breaks from the set of tracks with $S=S_{max}$ can be exactly represented by some integer number of tracks with $S=S_{next}$. These restrictions are outlined in the next two theorems.

Theorem 8.6: Let S_{max} be the maximum S value currently in the track placement problem, M be the set of all tracks with $S = S_{max}$, $N = |M|$, and S_{next} be the largest $S \neq S_{max}$. If $N > 1$, S_{max} is a proper multiple of N , and $S_{next} \leq S_{max} * (N-1) / N$, then any solution to T that meets the bound must evenly space out the N tracks with $S = S_{max}$. That is, for each track M_i , with a position O_i , there must be another track M_j with a break at $O_i + S_{max} / N$.

Theorem 8.7: Given a set of tracks G , all of segment length X , where X is evenly divisible by $|G|$, and a solution O with these tracks evenly distributed. There is another set of tracks G' , all of length $Y = |G'| * X / |G|$, with a solution O' where the number of breaks at each position is identical for solution O of G and solution O' of G' . If solution in which G has been replaced with G' meets its bound, the solution with the original G also meets its bound.

```

Optimal_Factor(T) {
  // Theorem 8.2: Run algorithm independently on relatively prime sets
  // of tracks
  If T can be split into two sets G1⊂T and G2 = T-G1, where the segment
  length of all elements of G1 are relatively prime with the segment
  length of all elements of G2 {
    Optimal_Factor(G1)
    Optimal_Factor(G2)
    Quit
  }

  // Theorem 8.3: Factor out unshared prime factors from each track's S
  // value
  Initialize Si for each track Ti to the segment length of that track.
  While the Si of any track Ti has more of any prime factor P than the
  Sis of all other tracks, individually, in T {
    Set Si = Si/P
  }

  While tracks exist without an assigned Oi {
    // Theorem 8.4: Place any full sets
    While there exists a set of G tracks, where Si = |G| {
      For each track in G, assign positions to those without a
      preassigned Oi, such that there is a track in G at all
      positions 0≤pos<|G|.
      Eliminate all tracks in G from further consideration.
    }
    If all tracks have their Oi assigned, end.

    // Theorem 8.6: Smax must be evenly divisible by |M|, and require
    // that Snext ≤ Smax*(|M|-1)/|M|
    Let Smax = the largest Si amongst unplaced tracks
    Let Snext = the 2nd largest Si amongst unplaced tracks
    Let M be the set of all tracks with Si = Smax
    If no track has an assigned position, assign M1's position O1 to 0.
    Assign all unassigned tracks in M to a position Oi, such that all
    tracks in M are at a position k*Smax/|M|, for all k 0≤k<|M|.
    If all tracks have their Oi assigned, end.

    // Theorem 8.7: Require Snext = c*Smax/|M| for some integer c≥1
    // Use placeholder tracks of S=Snext to model existing breaks
    // Snext must be evenly divisible by c to make Th. 6 work
    Add c placeholder tracks, where for j=0..c-1, Sj = Snext, and
    Oj = j*Smax/|M|
    Remove all tracks in M from further consideration
  }
}

```

Figure 8.8: The pseudocode for the Optimal Factor algorithm. This algorithm is optimal [Compton02b] provided all restrictions listed in the pseudocode are met. The comments list the theorems governing the operations.

Using the theorems presented, an optimal algorithm [Compton02b] can be constructed provided the restrictions in Theorem 8.6 and Theorem 8.7 are met. There is one additional restriction that is implied through the combination of the two theorems above. Because the tracks at a given S value must be evenly spread, the offsets assigned to the placeholder tracks added using Theorem 8.7 in the previous iteration must fall at offsets calculated using Theorem 8.6. This is accomplished by requiring that S_{next} also be evenly divisible by the number of placeholder tracks of that length added during the track conversion phase. Figure 8.8 gives the pseudocode for the Optimal Factor Algorithm. This pseudocode is commented to show the relevant theorems for each operation.

8.2.5 Relaxed Factor Algorithm

While the Optimal Factor Algorithm does generate track placements that are optimal in diversity score, there are significant restrictions controlling its use. Because not all architectures may meet the segment length and track quantity restrictions of the optimal algorithm, a relaxed version of the algorithm has also been developed. This relaxed algorithm operates without restrictions on track segment lengths or the quantity of tracks of each length, but may not be optimal.

The general framework of the algorithm remains basically the same as the optimal version, although the placement and track conversion phases differ in operation. Previously, restrictions were used on track length and quantity to ensure optimality. In the Relaxed Algorithm, a number of heuristics are instead used to intelligently spread the

tracks' breaks across the architecture. Figure 8.9 shows the changes made to the Optimal Algorithm's main `while` loop for the Relaxed Algorithm. Figure 8.11 and Figure 8.14 contain the replacement heuristic functions for the placement and track conversion. Note that there is an additional sub-function that will sometimes be needed for placement, shown in Figure 8.12.

```

// Place any full sets
While there exists a set G of tracks where  $S_i$  of every track  $G_i$  equals
|G| {
    For each track in G, set their  $O_i = i$  (their index in G)
    Remove all tracks in G from further consideration.
}

// perform some initializations of structures needed in placement and
// track conversion
Initialize tracks[] array to size  $S_{max}$  = largest  $S_i$  amongst all tracks in
T, and fill with 0's
Initialize breaks[] array to size  $K = \text{LCM of } S_i\text{'s of all tracks in T, and}$ 
fill with 0's.
// Iteratively assign offsets of unplaced tracks w/longest segment length
While tracks exist in T without an assigned  $O_i$  {
    Let  $S_{max}$  = the largest  $S_i$  amongst unplaced tracks
    Let  $S_{next}$  = the largest  $S_i$  amongst unplaced tracks such that  $S_{next} < S_{max}$ 
    Let M be the set of all unplaced tracks with  $S_i = S_{max}$ 
    Relaxed_Placement(M, tracks[], breaks[])
    If all tracks have their  $O_i$  assigned, end
    Remove all tracks in M from further consideration
    Convert_To_Snext( $S_{next}$ , tracks[], breaks[])
}

```

Figure 8.9: The Relaxed Algorithm code that replaces the main `while` loop in the Optimal Algorithm. Sub-functions for this pseudocode can be found in Figure 8.11 and Figure 8.14.

In the Relaxed Algorithm, full sets are removed in the same way they were in the Optimal Algorithm, except the operation is performed once instead of within the `while` loop. This is because while the Optimal Algorithm adds new placeholder tracks in the loop, potentially creating new full sets, the Relaxed Algorithm does not—the only full sets that can exist in the Relaxed Algorithm are present before the loop starts.

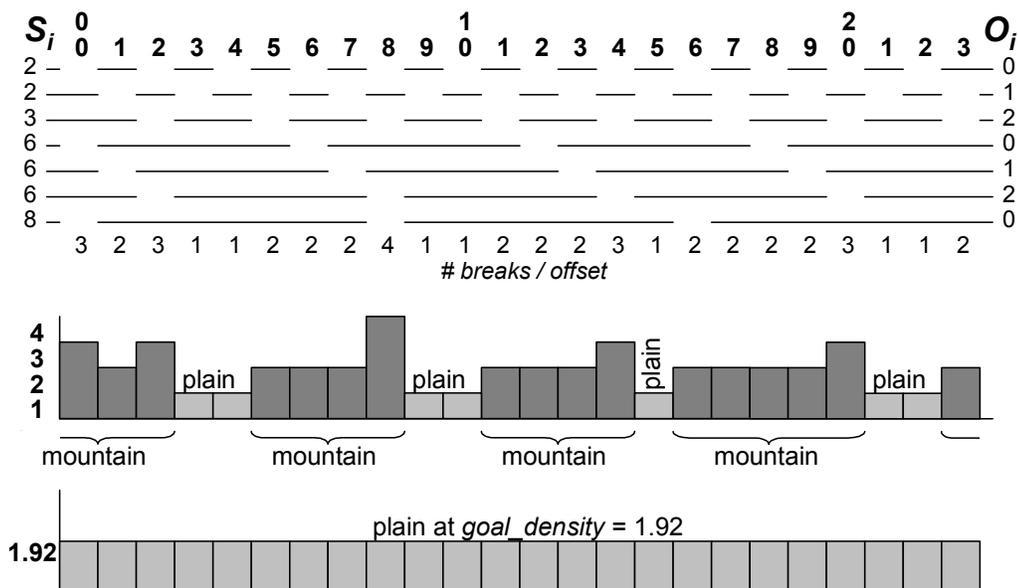


Figure 8.10: A track arrangement (top), corresponding topography (middle), and the ideal topography for these tracks (bottom). Note that this is an exaggerated example for illustrative purposes. A plain is a series of consecutive values at the lowest elevation, a mountain is a series of consecutive values that are higher than the minimum for the array. The topographies are circular, meaning that in the middle diagram, the value at index 23 is part of the same mountain as indices 0 through 2.

The bulk of the changes are in the general placement and placeholder track substitution phases. Again, heuristics are now used in place of restrictions on the problems that the algorithm can handle. The routing architecture can be considered in terms of a topography, where elevation at a given location is equivalent to the number of breaks at that position. Because “mountains” represent areas with many breaks, and therefore fewer potential routing paths, the algorithm attempts to avoid their creation. Instead, it focuses on placing tracks to evenly fill the “plains” in the topography, where the breaks from the additional tracks will have a low effect on the overall routability of

the resulting architecture. The top portion of Figure 8.10 shows the plains and mountains of a sample topography.

The pseudocode for the placement portion of the Relaxed Factor algorithm is shown in Figure 8.11, with a sub-function in Figure 8.12. This function uses the *tracks[]* array, which was first initialized in the main body of the algorithm to all 0's. This array is updated to always be the same size as the segment length of the track group currently under consideration (only one segment length is examined at a time). Each position in the array represents a different potential offset that can be chosen for that particular segment length, and essentially indicates the number of breaks that will be overlapped using the given potential offset.

```

Relaxed_Placement(M, tracks[], breaks[]) {
    Let unplaced be the number of tracks in M that are unplaced.
    While the number of i's where tracks[i] = min(tracks[]) is  $\leq$  unplaced
    {
        Place one track at each location with tracks[i] = min(tracks[])
        Update unplaced, tracks[], breaks[]
    }
    If all tracks[i] are the same, space all unplaced tracks in M evenly
    Else if unplaced tracks remain in M {
        Let U be the set of unplaced tracks in M
        Density_Based_Placement(U, unplaced, tracks[])
    }
}

```

Figure 8.11: The relaxed placement function. The function *Density_Based_Placement()* that is used here appears in Figure 8.12.

Using this information, the algorithm attempts to place tracks at the offsets that have the fewest breaks from the already-placed tracks. The *tracks* array is updated later in the track transformation function, which will be discussed shortly. For cases where all

the values in the *tracks[]* array are identical, the unplaced tracks of the current segment length are spaced evenly.

```

Density_Based_Placement(U, unplaced, tracks[], breaks[]) {
  Find the widest plain in tracks[]
  If tie, choose the one of these with the widest adjacent mountain
  If tie, choose randomly from the tied plains

  Let root_point be the location at the end of this plain run adjacent
  to the wider mountain
  Initialize blockage to the mountain adjacent to root_point
  Initialize next_plain to the plain adjacent to blockage (the plain
  that does not include root_point, unless blockage is the only
  mountain in the architecture)
  Initialize region to next_plain plus blockage

  While (1) {
    num_to_add = Calculate_Num_Tracks(region, S_max, unplaced, tracks[])
    Assign num_to_add unplaced tracks from U to next_plain, spaced
    evenly with respect to each other as well as the boundaries of
    next_plain
    Update tracks[], breaks[], and unplaced
    If at root_point or all are placed, end
    Set blockage to the mountain on the other side of next_plain from
    the current blockage
    Set next_plain to the plain on the other side of the new blockage
    from the current next_plain
    Update region to include new blockage and new next_plain
  }
}

Calculate_Num_Tracks(region, S_max, unplaced, tracks[]) {
  Let ideal =  $\frac{\text{size}(\text{region})}{S_{\max}} \left( \text{unplaced} + \sum_{j=0}^{S_{\max}-1} \text{tracks}[j] \right) - \sum_{i \in \text{region}} \text{tracks}[i]$ 
  Return either floor(ideal) or ceil(ideal) as num_to_add, whichever yields
  a smaller abs(potential_density(region, num_to_add) - goal_density)
}

```

Figure 8.12: The density based placement function and the function to calculate the number of tracks to add to a given region in the newest plain. A plain is defined as consecutive positions in *tracks[]* equal to $\min(\text{tracks}[])$. A mountain is defined as consecutive positions in *tracks[]* not equal to $\min(\text{tracks}[])$.

When there are fewer tracks than positions with the minimum number of breaks, the plains in the *tracks[]* array cannot be filled evenly. In this case, the sub-function in Figure 8.12 is used to perform density-based placement. Again, the goal is to distribute

the breaks as uniformly as possible throughout the architecture. The *tracks[]* array (which holds the number of breaks on placed tracks at each position) is treated as circular, with the last position next to position 0. The density-based placement looks at an ever-increasing region of this array and places unplaced tracks to bring the density of breaks in that region close to the *goal_density*. This *goal_density* represents a theoretical solution with a completely flat topography. Figure 8.10 illustrates a sample topography, with plains and mountains labeled, and the corresponding topography if the *goal_density* could be achieved.

The region of consideration will initially be a plain and an adjacent mountain. The number of breaks that should be added to the areas of lowest elevation (a plain) in this region is calculated. Any given new track can only add at most one break at one position in the *tracks[]* array, as the *tracks[]* array is always the same size as the current tracks' segment length. The calculated number of tracks is placed such that the breaks are spread evenly through the plain. The region is updated to include the next plain and mountain in the circular path around the *tracks[]* array. Again, the number of tracks to add to the bottom of the new plain is calculated in order to bring the density of the entire region closer to the ideal density. This pattern continues until the algorithm has traveled all the way around the *tracks[]* array back to the beginning position, and the entire *tracks[]* array has become the region.

The actual calculation of *num_to_add*, the number of tracks to add to the plain, is somewhat complex. Ideally, Equation 8.4 should be minimized, where

$potential_density(region, num_to_add)$ is the density of the current region if num_to_add tracks are added with breaks in that region, and $goal_density$ is the density if all tracks in the problem could be added such that the topography could remain perfectly flat throughout, as in the bottommost part of Figure 8.10. Equation 8.5 is the result of solving for the ideal number of tracks to be added [Compton02b], where $unplaced$ is the number of tracks that have not yet been placed, and $size(region)$ is the number of discrete locations in the current region.

$$\text{Equation 8.4: } \text{abs}(potential_density(region, num_to_add) - goal_density)$$

$$\text{Equation 8.5: } ideal = \frac{size(region)}{S_{max}} \left(unplaced + \sum tracks \right) - \sum_{i \in region} tracks[i]$$

The actual num_to_add is then either the floor() or ceiling() (whichever yields a lower result in the original equation to be minimized) of the possibly-fractional $ideal$, since fractional quantities of tracks cannot be placed. This fraction to integer conversion does introduce some error, which can accumulate as the algorithm progresses. Thus, the starting point is chosen such that the widest plain will be considered last, as it can tolerate the most error by amortizing it over its wider area.

Once all tracks with $S=S_{max}$ have been placed, the algorithm must prepare for the next iteration when the tracks with $S=S_{next}$ are placed. The effects of all previously placed tracks should be considered when placing tracks of length S_{next} . The contents of

the $tracks[]$ array are therefore in truth more complex than described previously. The $tracks[]$ array simulates the conversion of all previous track lengths to length S_{next} .

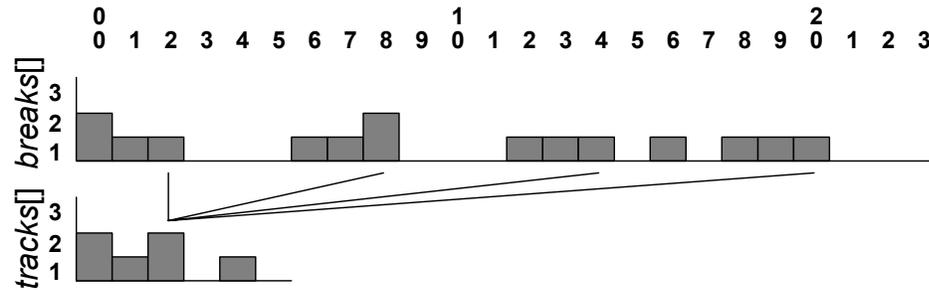


Figure 8.13: An example of a $breaks[]$ array for $K = 24$, and the corresponding $tracks[]$ array for $S_{next} = 6$. The lines between the two arrays indicate the locations in the $breaks[]$ array used to compute the value of $tracks[2]$.

The $tracks[]$ array provides a “global view” of the $breaks[]$ array within the potential offset values 0 to $S_{next}-1$, as illustrated by Figure 8.13. It basically records the peaks in the breaks array coinciding with each potential offset value that could be assigned to the S_{next} -length tracks. This translates all of the longer tracks into approximate numbers of S_{next} -length tracks at each offset from 0 to $S_{next}-1$ so that the effects of these tracks can be considered when placing tracks of length S_{next} . The pseudocode for converting all previously placed tracks to tracks of length S_{next} using the $tracks[]$ array is given in Figure 8.14.

```

Convert_to_Snext( $S_{next}$ ,  $tracks[]$ ,  $breaks[]$ ) {
  Replace current  $tracks[]$  array with  $tracks[]$  array of size  $S_{next}$ 
  For  $i = 0$  to  $i = S_{next}-1$ 
     $tracks[i] = \max(breaks[i+S_{next}*y])$  for all  $y$  such that  $i+S_{next}*y$ 
    falls within the bounds of the  $breaks[]$  array
  }

```

Figure 8.14: This function does not create actual placeholder tracks to represent tracks with $S > S_{next}$, but it does fill the $tracks[]$ array in such a way as to simulate all previously placed tracks being converted to segment length S_{next} .

8.3 Algorithm Comparison

A number of terms are used to describe the problem sets used in the algorithm testing. The value $numT$ refers to the total number of tracks in a particular track placement problem, $numS$ refers to the number of discrete S values in the problem, $maxS$ is the largest S value in the problem, and $maxTS$ is the maximum number of tracks at any one S value in the problem. The very large search space has also been reduced by only considering problems where the number of tracks at each S value is less than the S value itself, since Theorem 8.4 strongly implies that cases with S or more tracks of a particular S value will yield similar results by placing tracks from any full sets one per potential offset. Cases with only one track, or all track lengths less than 3, are trivial and thus ignored. The three terms, $numT$, $numS$, and $maxS$, along with the restrictions above, define the track placement problems tested.

The first test was to verify that the Optimal Factor yields a best possible diversity score in practice as well as theory. The results of this algorithm were compared to those of the Brute Force for all cases with $2 \leq numT \leq 8$, $1 \leq numS \leq 4$, and $3 \leq maxS \leq 9$, which represents 5236 different routing problems. Note that even with these restrictions on the problem set, the runtime of Brute Force is over a week of computation on multiple machines. In all cases where a solution could be found using Optimal Factor, the resulting diversity score was identical to the Brute Force result. Furthermore, Relaxed Factor was compared to Optimal Factor for this same range, and it was found that within

that range, Relaxed Factor produces optimal results for all cases that meet the Optimal Factor restrictions.

Next, the performances of Relaxed Factor and Simple Spread were compared to the results of the Brute Force method for the same search space as above to determine the quality of the heuristics. The results, normalized to Brute Force, are shown categorized by $numT$, $numS$, and $maxTS$ in Figure 8.15. In these graphs, Min is the worst performance of the algorithm at that data point, Max is the best, and Avg is the average performance. In this figure, the Brute Force result is a constant value of 1. Figure 8.15 left indicates that both heuristics achieve optimal results in some cases, with Relaxed Factor achieving results on average nearly optimal across the entire range. Simple Spread improves with the increasing number of tracks, and both algorithms degrade with an increase in the number of different S values, though Relaxed Factor to a lesser degree than Simple Spread.

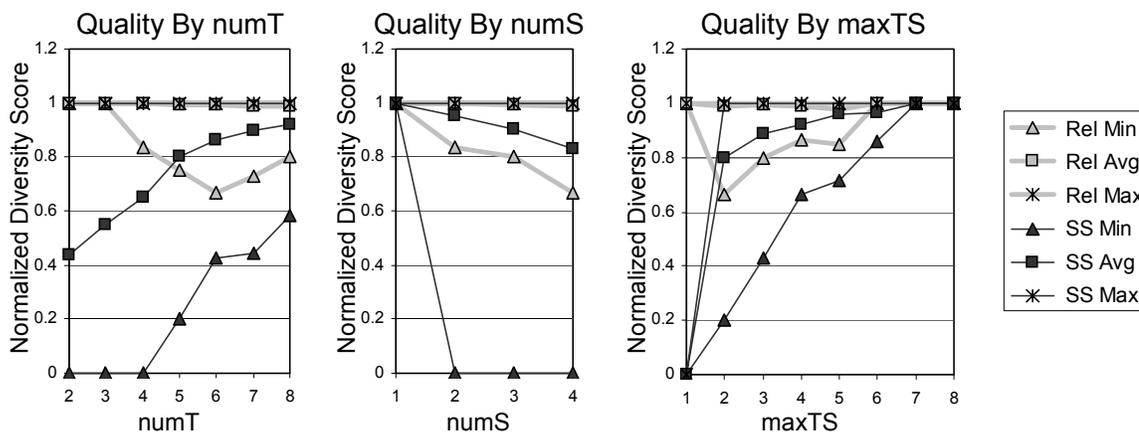


Figure 8.15: A comparison of Relaxed and Simple Spread to the Brute Force method, with respect to $numT$ (left), $numS$ (center), and $maxTS$ (right). The diversity scores for each case were first normalized to the Brute Force result, which is represented by a horizontal line at the value 1.

The upswing of both algorithms' minimums towards higher values of $numT$ may be an artifact of the restrictions used to limit the problem set – since at most 4 unique S values are allowed, when there are more than 4 tracks there are at least two tracks with the same S value. Figure 8.15 right shows that as the number of tracks per S value increases, the quality of all algorithms improves. The only exception is for the relaxed algorithm when $maxTS=1$; when there is only one track per S value Relaxed Factor is always optimal. All throughout these tests Relaxed Factor is superior to Simple Spread. This indicates the critical importance of correlation between S values in track placement. Figure 8.15 center also demonstrates that as $numS$ increases, so does the difficulty of finding a good solution, as was expected. Note that the results for both heuristics are optimal for the case when there is only one S value, as there are no correlations to contend with, and only an even spreading is required.

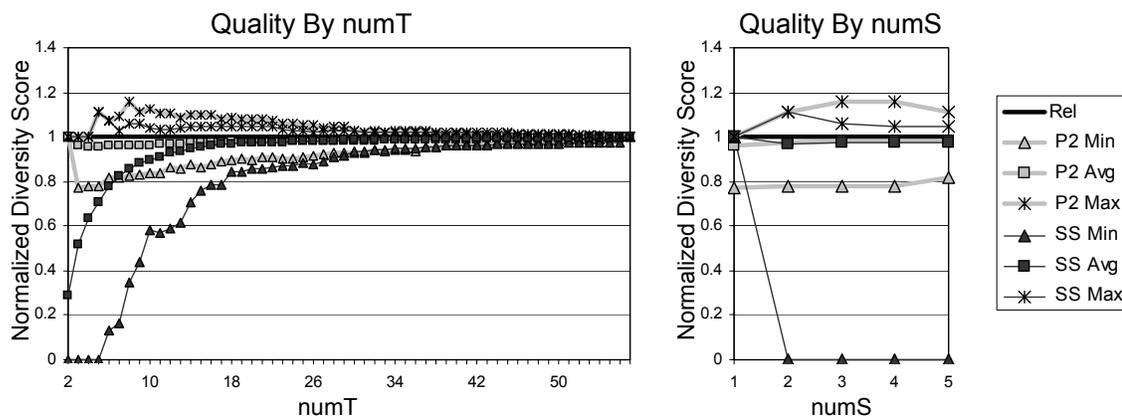


Figure 8.16: Relaxed Factor, Power2, and Simple Spread comparison for cases with only power-of-two S values. The results for Power2 and Simple Spread were normalized to the Relaxed Factor value.

The results of the Power2 algorithm were then compared to those of the other heuristics. For these tests, a different set of track placement problems were used, where all S values were a power of two, up to a $maxS$ of 32. The number of tracks in the tests were not restricted beyond the requirement that for each S value, the number of tracks must be less than that S value. These restrictions result in 32,762 different track placement problems. Figure 8.16 left graphs the minimum, average, and maximum diversity scores (normalized to the Relaxed Factor result) at each number of tracks ($numT$). The results of the algorithms become more similar as the number of tracks increases. The performance of the Power2 algorithm is similar to the Relaxed Factor algorithm, with Power2 performing better than Relaxed Factor in some cases, and worse in others. Simple Spread performs worse for these tests than the previous because by restricted all S values to powers of two, all tracks are correlated.

Figure 8.16 right graphs the same data as Figure 8.16 left, but instead graphed by $numS$. In general, performance is relatively consistent regardless of the number of S values, with the performance of Power2 again close to that of Relaxed Factor. The exception is when there is only one S value, where correlations do not matter. Unlike Relaxed Factor and Simple Spread, however, Power2 is not always optimal in this case because its focus is on properly handling correlations more than even spreading within one S value.

	OFDM	Camera	Radar	Image Processing	Sort	Matrix Multiply	FIR Filters
Simple Spread	27	23	20	23	23	11	20
Relaxed	24	19	13	15	13	10	11

Figure 8.17: The number of tracks in our target architecture required to successfully place and route all netlists in an application using the given track placement algorithm.

Next, the place and route tool [Compton02d] was used to test the correspondence between diversity score and routability of the architectures. These architectures are based on a tileable coarse-grained architecture similar in structure to RaPiD [Ebeling96, Cronquist99a]. This architecture has two length-2 local tracks, four length-4 local tracks, eight length-4 distance tracks, and eight length-8 distance tracks. Note that local tracks do not allow connections between wire segments to form longer wires. A test case was created for each of seven different multi-netlist applications using each track placement algorithm. By keeping the proportion of track types constant but varying the total quantity of tracks, the minimum number of tracks required to successfully place and route every netlist in the application onto this target architecture was determined for each track placement algorithm. Figure 8.17 lists the results of this experiment. Performing track placement using Relaxed Factor allowed netlists to be routed with on average 27% (and up to 45%) fewer tracks than Simple Spread.

8.4 Summary

As demonstrated in this chapter, the track placement problem involves fairly subtle choices, including balancing requirements between tracks of the same length, and

between tracks of different, but not relatively prime, lengths. The diversity score quality metric was introduced to measure the impact of track placement, and multiple algorithms were presented to solve the track placement problem. One of these algorithms is provably optimal for some situations, though it is complex and works for only a relatively restricted set of cases. A relaxed version of the optimal algorithm was also developed, which appears to be optimal in all cases meeting the restrictions of the optimal algorithm, and whose average appears near optimal (within 1.13%) overall. While the other simple heuristics presented here do have problems in some cases, they provide simple methods to obtain good quality results in many cases (particularly the Power2 algorithm, used by two algorithms from Chapter 7).

Track placement algorithms are critical in at least two notable situations. First, automatic track placement is used for automatic reconfigurable architecture generation. Second, even in hand designed architectures, the track placements achieved by the designers can be improved by carefully considering correlations between track lengths. Automatic track placement techniques can be applied by the FPGA designer to their work to potentially improve overall quality of the resulting architecture.

Chapter 9

Flexibility Testing

The design of a reconfigurable architecture differs from the design of a conventional ASIC in a key aspect: flexibility. The quality of an ASIC is generally measured in terms of power, performance, and area. However, with reconfigurable hardware, flexibility is equally important, given that the goal of these structures is to implement various circuits using a single set of hardware resources. In particular, measuring the flexibility of reconfigurable hardware is very important for automatically generated architectures intended for use in systems-on-a-chip. A designer may create many of these architectures, and wish to choose the one best for the purpose at hand on the basis of all the comparison metrics, including flexibility.

The flexibility metric has, however, been largely ignored by the FPGA and reconfigurable computing community. FPGA designs are frequently measured by “gate count” or logic block count. However, many feel that the first metric is primarily marketing propaganda and not an effective measure of the size or variety of the circuits the design can implement. The second metric, logic block count, is also undesirable, as it is difficult to compare values across architectures with different logic block structures.

One possible solution to measure flexibility might be to test the number of possible paths in an architecture, combined with the number of logic units. The more placement and routing options available, the more flexible the design. While this method might be effective for the general case when the structure of the implemented circuits is completely unknown, it is not ideal for testing domain-specific architectures. Domain-specific architectures attempt to omit logic resources and routing options that will not be needed by any of the netlists (circuits) within the domain. In these cases, additional logic and routing may not actually contribute to the flexibility (within the domain) of the architecture, but only the execution and configuration overhead. It is critical to consider the flexibility within the domain, rather than a more generic measurement.

Therefore, the flexibility testing methods presented here are based on the ability of architectures to implement circuits from the domain, beyond the specification used to create the architecture. A synthetic circuit generator is used to provide a large number of circuits for use in the flexibility comparisons. This chapter begins with a discussion of the techniques used to create the synthetic circuits. The relative flexibilities of the architecture generation methods from Chapter 7 are then compared. Finally, other important uses for the synthetic circuit generator are discussed.

9.1 Circuit Generator

Previous work in synthetic circuit generation generally operates at the gate level, or in the case of FPGAs, in terms of LUTs [Darnauer96, Hutton98, Wilton01, Hutton02].

However, the circuits used for the Totem architecture generation discussed in this dissertation are much more coarse-grained in nature, using word-width ALUs, multipliers, and other large units as the logical components. In many cases, the techniques from another circuit generator are used here [Hutton98, Hutton02], but are modified both to compensate for heterogeneity as well as for coarse granularity and the structure of RaPiD netlists. The operation of the circuit generator involves first measuring a few key characteristics of a “parent” netlist or netlists. Then the synthetic circuit is generated using these characteristics as guidelines. The next few sections describe the process of profiling netlists, followed by the techniques used to generate the synthetic circuits.

9.1.1 Circuit Profiling

Circuits are profiled in order to measure key defining characteristics. These characteristics are then used to generate new circuits with structures similar to the original parent circuit. RaPiD netlists currently form the source circuit material, and these netlists have a distinct high-level structure. RaPiD netlists are split into systolic *stages*, where each stage operates in parallel, and communicates only with adjacent stages. Inter-stage communication is limited, as is the number of inputs and outputs of the circuit. For the synthetic circuits to mimic RaPiD netlists, the stage and communication structures must be accounted for. Each netlist is converted to a directed graph, and measured for several characteristics, including:

- I/O requirements of the netlist
- Number of netlist stages
- Minimum/maximum connections between stages
- Minimum/maximum number of delay levels (logic levels) within the stages
- Number of logic nodes
- % of nodes of each type (ALU, multiplier, etc)
- Proportion of signals to nodes
- Proportion of signals that are back edges (sink is a lower delay level than the source)

Figure 9.1 illustrates a sample RaPiD netlist with 28 logic nodes, represented as a directed graph. Note that all multi-terminal signals are split into a set of 2-terminal signals for this profiling.

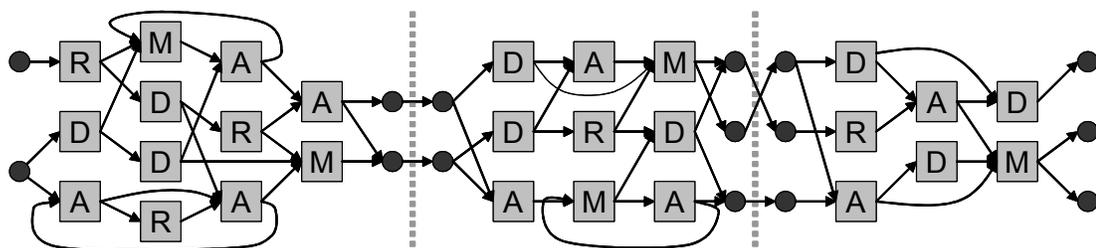


Figure 9.1: A directed graph of a sample RaPiD netlist to be profiled. This netlist has three stages (separated by the dashed line). The left stage has four delay levels, while the others each have three delay levels. Logic units are represented as squares and labeled as different logic types (RAM, ALU, Multiplier, Data register), and signals are represented as arrows. Back edges are signals with the sink at an earlier delay level than the source. I/Os of the circuit and between stages are represented by the dark circles.

9.1.2 Domain Profiling

Using a set of circuit profiles as generated in the previous section, a description of an application domain can be created. The minimum, maximum, mean, and standard deviation across the set of profiles are computed for each of the circuit characteristics. New profile information is generated by choosing values within the given range

according to a Gaussian distribution for each characteristic (using the mean and standard deviation to describe the function). The generated profile is then used to create a netlist circuit as described in the next section.

9.1.3 Circuit Creation

After the circuit characteristics are obtained, either through manual input or profiling of one or more circuits, the graph of the synthetic circuit can be created. The goal of this generator is to create a circuit that, when profiled, will have characteristics that approximate the circuit characteristics of the specification. First the general structure, or “skeleton” of the circuit graph is created. Next the logic nodes are created, and then the edges are created to connect the logic nodes together to form a directed graph. Figure 9.2 demonstrates these steps for an example synthetic circuit created from the profile of Figure 9.1. Finally, the completed graph describing the circuit is converted into an actual netlist.

Graph Skeleton

The circuit generator first creates a skeleton for the circuit before creating the circuit graph. The generated circuits need to have the general structure of RaPiD netlists, which is enforced by the skeleton. First the stages of the circuit are instantiated, where the number of stages is part of the input specification. The amount of connectivity between stages is also determined at this time. A random number of inter-stage connections is chosen between the minimum and the maximum stage I/O of the

specification (inclusive). The number of stage outputs of one stage is forced to match the number of inputs of the next stage. Next, the number of delay levels for each stage is randomly chosen between the minimum and maximum number of delay levels in the specification (inclusive). A sample skeleton created from the profile from Figure 9.1 is given in Figure 9.2a. At this point, the remainder of the circuit can be created.

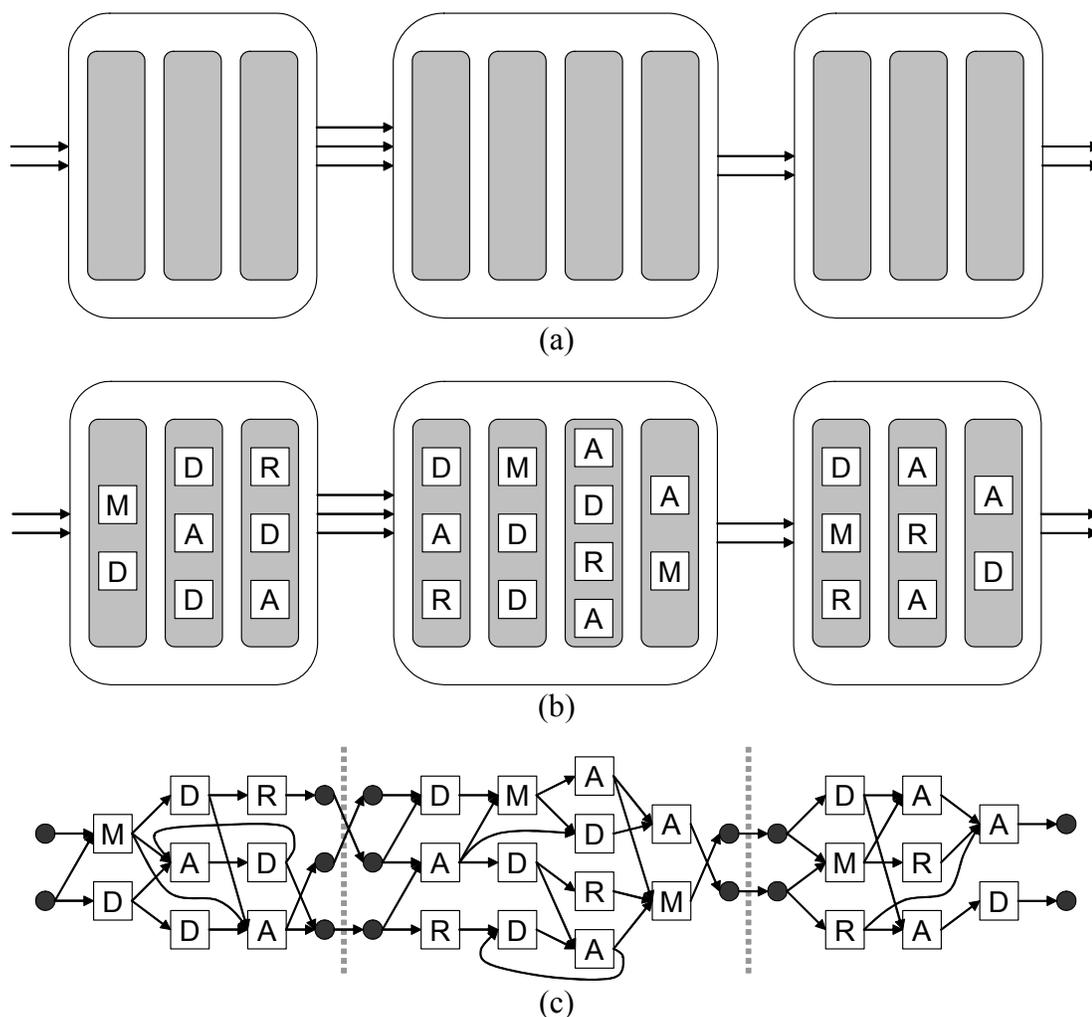


Figure 9.2: Steps in the creation of an example synthetic circuit graph. (a) The skeleton, including the number of stages (white bubbles) and delay levels within the stages (shaded bubbles). The connectivity between stages is also indicated by the arrows. (b) Logic nodes are then added to the skeleton. (c) Finally, the edges are added to connect the logic nodes.

Graph Nodes

The structure of the skeleton determines the minimum number of nodes that must be created in order to create a circuit based on that skeleton. None of the stages are permitted to be empty, and each delay level within each stage must have at least one node. If the number of nodes in the specification is lower than the minimum number allowed by the skeleton, the number of nodes is increased to meet the skeletal minimum. The logic nodes are then created, approximately in the same proportions by type (i.e., ALU, multiplier, etc) as presented in the specification. Each stage is then assigned enough random nodes to meet its minimum number of nodes. The remaining nodes (if any) are assigned to random stages. Each node can only be assigned to a single stage.

Once the nodes are assigned to stages, they must also be assigned to delay levels within the stages, where each node may only be assigned to a single delay level within its assigned stage. Only one of the delay levels within each stage requires special consideration. The number of nodes at the last delay level is determined by the specification's average proportion of the last level nodes to the number of stage outputs, bounded at the upper end by the number of outputs. This helps to control the amount of inter-stage communication. This proportion is used with the number of outputs for each stage to compute the number of nodes belonging to the last delay level. Each remaining delay level in a stage is assigned a random (unassigned) node from that stage in order to ensure each delay level has at least one node. The remaining nodes are assigned to a

random delay level in the stage, not including the last level. Figure 9.2b shows the example skeleton after the logic nodes have been added.

Graph Edges

At this point, the logical structure of the circuit is complete, and the edges of the graph must be added. The edges are created in two phases: first the connections within each stage, then the connections between stages. There are several steps involved in the creation of the edges within the stages. First, the delay level assignments must be enforced. For each node, a signal is created that sinks at that node. The signal's source node is randomly chosen from the immediately previous delay level. Next, the number of inputs to the nodes is enforced. The number of required inputs depends on the type of node—a data register only requires a single input, but an ALU requires two. Signals are created to meet this requirement. These signals sink at the given node, and source from a random node in the stage. Output constraints are then fulfilled, where each node is required to have at least one output. For any nodes without an output, a signal is created that sources from that node, and sinks at a random node in the stage. Finally, the rest of the signals (quantity determined by the parameter of the ratio of signals to nodes) are created with random sources and destinations.

For the above steps, apart from the enforcement of delay levels, the likelihood of a signal being a back edge is determined by the parameter of percent of signals that are back edges. If a signal is determined to be a forward edge, its source or sink must be

chosen such that the source is from a lower delay level than the sink. If the signal is determined to be a back edge, the source or sink must be chosen such that the source is from an equal or higher delay level than the sink, the sink is an ancestor of the source, and at least one register must separate them.

The second phase of edge generation is connecting the stages. When creating the routing within the stage, temporary stage I/Os are created for each stage (the quantity of which is determined during skeleton creation), and the nodes within the stage are permitted to connect to these I/Os. The stages are then connected by creating connections between the I/Os of adjoining stages. The outputs of each stage are paired randomly with inputs from the next stage, and a single signal is created to connect the pair of I/Os. Figure 9.2c shows the example circuit graph after the edges have been added.

Final Circuit

At this point, a directed graph has been created to describe the synthetic circuit. This graph then must be converted into the actual circuit format. The input signals at each node are assigned random input ports on that node, after ensuring at least one signal per port. The output signals are assigned random output ports on the node, but in this case, not every port is required to have an assigned signal. Finally, the explicit stage structure becomes implicit, as the connections between nodes and the stage I/Os are propagated to the inputs of the next stage, as determined by the inter-stage connections.

9.2 Synthetic Circuit Validation

The synthetic circuits are intended to mimic the structure of actual circuits, without duplicating those circuits exactly. To test that this is the case, ten synthetic circuits were created for each actual RaPiD netlist available, and the synthetic circuits were profiled with the same techniques used to profile the RaPiD netlists. The profiled values were normalized to the characteristics of the original real circuits, and averaged across the ten synthetic circuits. A chart comparing the normalized characteristics of the generated circuits is given in Table 9.1. The data in this chart indicates that in general, the synthetic circuits have very similar characteristics to the original circuits. Additionally, most standard deviations listed are above zero, which indicates that there is some variety in the characteristic values, as desired.

A few characteristics are less accurately mirrored in the synthetic circuits than others. Among these are the number of instances per delay level, the inter-stage communication (stage I/O), percent back edges, and fanin/fanout of individual nodes. Future revisions of the synthetic circuit generator should aim to improve the accuracy of generation to better follow the specification of these characteristics. On the whole, however, the synthetic circuit generator meets the goals of generating architectures close, but not identical to, a given set of characteristics (in this case, profiled directly from real netlists).

Table 9.1: A comparison of characteristics of generated synthetic circuits to those of the original circuits. All characteristics were normalized to the original netlists. The average values and standard deviations across ten synthetic circuits are given for each parent circuit. Blank entries for % back edges indicate that no back edges were created in those synthetic circuits.

Characteristic	1d_dct40		color_interp		decnsr		fft16_2nd		fft32		fft64		firsm		firsm2		firsm3	
	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev
# Inputs	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
# Outputs	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
# Instances	1.01	0.03	1.54	0.10	1.00	0.00	1.02	0.01	1.02	0.02	1.01	0.00	1.00	0.00	1.00	0.00	1.00	0.00
# Signals	0.97	0.05	1.70	0.10	1.20	0.00	0.95	0.03	0.94	0.03	1.02	0.02	1.06	0.02	1.06	0.02	1.06	0.03
Sig:Inst Ratio	0.96	0.03	1.10	0.03	1.20	0.00	0.93	0.03	0.91	0.02	1.01	0.02	1.06	0.02	1.06	0.02	1.06	0.03
# Stages	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
% Backedges	0.31	0.50					0.50	0.29	0.55	0.36	0.67	0.17						
Stage I/O	0.50	0.00	1.36	0.15	1.00	0.00	1.03	0.03	1.04	0.03	1.02	0.03	0.39	0.04	0.39	0.04	0.39	0.03
# Delay Levels	1.09	0.12	2.07	0.10	1.00	0.00	0.89	0.09	0.82	0.09	1.09	0.11	0.87	0.03	0.87	0.03	0.94	0.02
Inst / Stage	1.01	0.03	1.54	0.10	1.00	0.00	1.02	0.01	1.02	0.02	1.01	0.00	1.00	0.00	1.00	0.00	1.00	0.00
Inst / Delay Level	0.79	0.09	0.76	0.06	1.00	0.00	1.44	0.15	1.58	0.17	1.07	0.11	0.73	0.04	0.73	0.04	0.68	0.02
Avg Node Fanin	1.14	0.08	1.01	0.02	0.62	0.04	1.00	0.02	1.05	0.04	0.69	0.02	0.88	0.02	0.88	0.02	0.86	0.02
Avg Node Fanout	1.39	0.09	1.06	0.02	0.70	0.04	1.00	0.02	1.05	0.04	0.70	0.02	0.99	0.02	0.99	0.02	0.98	0.02
% registers	1.02	0.05	1.00	0.01	1.00	0.00	1.03	0.02	1.04	0.03	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
% multipliers	0.99	0.03	1.00	0.01	1.00	0.00	0.98	0.01	0.98	0.02	0.99	0.00	1.00	0.00	1.00	0.00	1.00	0.00
% RAMs	1.00	0.00	1.02	0.03	1.00	0.00	0.98	0.01	0.98	0.02	0.99	0.00	1.00	0.00	1.00	0.00	1.00	0.00
% ALUs	0.99	0.03	1.00	0.00	1.00	0.00	0.98	0.01	0.98	0.02	0.99	0.00	1.00	0.00	1.00	0.00	1.00	0.00

Characteristic	firsyeven		firtm_1st		firtm_2nd		img_filt		limited		limited2		log32		matmult		matmult4	
	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev
# Inputs	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
# Outputs	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
# Instances	1.00	0.00	1.03	0.02	1.05	0.04	1.00	0.00	1.04	0.03	1.01	0.04	1.00	0.00	1.07	0.04	1.05	0.02
# Signals	1.04	0.01	0.69	0.04	0.79	0.04	0.99	0.02	0.83	0.06	1.04	0.06	1.01	0.01	0.76	0.05	0.74	0.04
Sig:Inst Ratio	1.04	0.01	0.67	0.03	0.76	0.04	0.99	0.02	0.80	0.04	1.03	0.04	1.01	0.01	0.71	0.04	0.70	0.03
# Stages	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
% Backedges			0.58	0.43	0.49	0.24			0.35	0.23	0.34	0.24			0.70	0.39	0.64	0.26
Stage I/O	0.36	0.02	0.81	0.05	0.57	0.12	0.85	0.10	1.15	0.09	0.74	0.11	0.32	0.02	0.54	0.10	0.55	0.09
# Delay Levels	1.01	0.04	0.96	0.06	0.95	0.07	1.17	0.05	0.96	0.03	1.01	0.04	0.96	0.01	0.88	0.06	0.93	0.05
Inst / Stage	1.00	0.00	1.03	0.02	1.05	0.04	1.00	0.00	1.04	0.03	1.01	0.04	1.00	0.00	1.07	0.04	1.05	0.02
Inst / Delay Level	0.62	0.03	1.00	0.05	0.90	0.11	0.91	0.06	1.14	0.05	0.91	0.07	1.16	0.01	0.98	0.10	0.91	0.05
Avg Node Fanin	0.89	0.01	0.70	0.02	0.73	0.02	1.01	0.03	0.73	0.01	0.63	0.02	1.26	0.01	0.74	0.02	0.71	0.02
Avg Node Fanout	0.96	0.01	0.74	0.02	0.77	0.02	1.05	0.03	0.86	0.02	0.70	0.02	1.51	0.01	0.78	0.03	0.74	0.02
% registers	1.00	0.00	1.04	0.04	1.07	0.05	1.00	0.00	1.06	0.05	1.00	0.01	1.00	0.00	1.29	0.17	1.11	0.04
% multipliers	1.00	0.00	0.97	0.02	0.95	0.04	1.00	0.00	0.97	0.03	0.99	0.04	1.00	0.00	0.94	0.04	0.95	0.02
% RAMs	1.00	0.00	0.97	0.02	0.95	0.04	1.00	0.00	0.97	0.03	1.00	0.00	1.00	0.00	0.94	0.04	0.95	0.02
% ALUs	1.00	0.00	0.97	0.02	0.95	0.04	1.00	0.00	0.97	0.03	0.99	0.04	1.00	0.00	0.94	0.04	0.95	0.02

Characteristic	matmult_bit		med_filt		psd		sort_g		sort_rb		sort_2d_g		sort_2d_rb		sync	
	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev	Avg	Dev
# Inputs	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
# Outputs	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
# Instances	1.07	0.04	1.00	0.00	1.08	0.03	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
# Signals	0.76	0.05	1.78	0.02	1.79	0.05	1.01	0.02	1.14	0.02	1.01	0.03	1.11	0.02	1.05	0.01
Sig:Inst Ratio	0.71	0.04	1.78	0.02	1.66	0.04	1.01	0.02	1.14	0.02	1.01	0.03	1.11	0.02	1.05	0.01
# Stages	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
% Backedges	0.70	0.39			0.98	0.45										
Stage I/O	0.54	0.10	0.90	0.11	1.00	0.00	0.77	0.08	0.71	0.09	0.71	0.08	0.70	0.10	0.35	0.01
# Delay Levels	0.88	0.06	1.44	0.04	1.11	0.13	1.06	0.00	1.05	0.00	1.06	0.00	1.07	0.00	1.04	0.01
Inst / Stage	1.07	0.04	1.00	0.00	1.08	0.03	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
Inst / Delay Level	0.98	0.10	0.66	0.04	1.04	0.12	0.83	0.04	0.81	0.04	0.79	0.04	0.78	0.05	0.64	0.01
Avg Node Fanin	0.74	0.02	0.77	0.01	0.99	0.03	0.76	0.02	0.76	0.01	0.73	0.02	0.72	0.02	0.91	0.01
Avg Node Fanout	0.78	0.03	0.77	0.01	1.41	0.04	0.76	0.02	0.76	0.01	0.73	0.02	0.72	0.02	0.99	0.01
% registers	1.29	0.17	1.00	0.00	1.34	0.12	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
% multipliers	0.94	0.04	1.00	0.00	0.93	0.02	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
% RAMs	0.94	0.04	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
% ALUs	0.94	0.04	1.00	0.00	0.93	0.02	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00

9.3 Testing Flexibility

A few different techniques have been examined to measure the flexibility of the routing architecture generation algorithms from Chapter 7. First, flexibility measurement using circuits generated from single profiles will be discussed, and the generation algorithms will be compared using architectures generated from these circuits. Next, a few issues involved in flexibility testing with domain-based synthetic circuits will be examined. A recommended method to test flexibility using these domain circuits is presented, followed by an analysis of the routing generation algorithms using this technique.

9.3.1 Single Circuit Flexibility

First the flexibility of the different architecture generation methods was tested for single circuit generation. Ten circuits were created for each RaPiD netlist, and architectures were created using the three techniques of Chapter 7 given a single generated circuit as input. The generated circuits were forced to have the exact same logic resources as the parent circuit so that only the flexibility of the routing structures is tested. An attempt was then made to place and route (using the tool described in section 4.2.3) the parent circuit onto each of the corresponding architectures generated from the synthetic circuits. This is repeated for each parent circuit, for a total of 780 test cases—three methods used for each of ten circuits generated for each of twenty-six parent netlists.

Table 9.2: A table listing the % likelihood that the original parent netlist can be placed and routed onto an architecture created from a synthetic circuit based on the parent characteristics. Results are given for each architecture generation method. The *Orig* column indicates if the netlist can be placed and routed onto an architecture created directly from that netlist.

	GH		AMO		AML	
	Orig	%	Orig	%	Orig	%
1d_dct40	Y	40	Y	80	N	60
color_interp	Y	30	Y	100	Y	100
decnsr	N	10	Y	90	N	30
fft16_2nd	Y	10	Y	100	Y	100
fft32	Y	10	Y	100	Y	100
fft64	Y	10	Y	100	Y	100
firms	Y	10	Y	100	Y	100
firms2	Y	10	Y	100	Y	100
firms3	Y	0	Y	100	Y	100
firmsyeven	Y	0	Y	100	Y	100
firms_1st	Y	0	Y	100	Y	90
firms_2nd	Y	0	Y	80	Y	90
img_filt	Y	20	Y	100	Y	100
limited	Y	0	Y	100	Y	100
limited2	Y	30	Y	60	Y	60
log32	Y	90	Y	100	Y	100
matmult	Y	0	Y	90	Y	100
matmult4	N	0	Y	100	Y	100
matmult_bit	Y	0	Y	100	Y	100
med_filt	Y	40	Y	100	Y	90
psd	Y	100	Y	100	Y	100
sort_g	Y	10	Y	100	Y	100
sort_rb	Y	40	Y	100	Y	100
sort_2d_g	Y	10	Y	100	Y	90
sort_2d_rb	Y	10	Y	100	Y	100
sync	Y	0	Y	40	Y	10
AVERAGE		18.5		93.8		89.2

The results, presented in Table 9.2, highlight the flexibility differences of the architecture generation methods: Greedy Histogram (GH), Add Max Once (AMO), and Add Min Loop (AML). Over 80% of the time, GH with synthetic circuits does not result in an architecture that is sufficiently flexible to implement the original circuit. In fact, in two cases, the original netlist cannot be placed and routed onto architectures created using GH directly from that netlist, as indicated by the *Orig* column in of Table 9.2.

AML similarly fails in two cases. However, in all cases, the original netlist can be successfully placed and routed onto an architecture created from that netlist using AMO.

On the other hand, both the Add Max Once and Add Min Loop generation methods allow a significant percentage of the parent circuits to be implemented on architectures created from synthetic netlists. Using the Add Max Once method, there is a 94% success rate, while the Add Min Loop method results in an 89% success rate, indicating the Add Max Once technique results in architectures which are inherently more flexible than either of the two other methods. This is not unexpected, as Add Max Once tends to introduce more distance routing tracks than Add Max Loop, and the segmentation points in the distance tracks provide more routing options to the place and route tool.

9.3.2 Domain Flexibility

In order to test the flexibility of the different routing generation algorithms for a given domain, first that domain must be profiled, as described in section 9.1.2. The applications used for testing throughout this work are each composed of two to six netlists. Therefore, the domain architectures were created from five synthetic netlists, as this number falls within but at the upper end of this range. The number of netlists used was chosen to be within the range to represent a realistic number of netlists. However, a value near the upper end of the range was chosen under the assumption that the more synthetic netlists are used to create an architecture, the wider the variety of circuit

structures the architecture will be able to handle, and the more likely it will be able to implement the original netlists.

Generating an architecture from a range of profiles creates architectures less like any particular original netlist than if the architecture was created from a single profile. The different original netlists whose profiles define the domain may vary considerably in their structure or resource requirements. Plus, the synthetic circuits are generated randomly (with a Gaussian distribution) from the ranges of characteristic values. Therefore, it is difficult to guarantee that the exact logic mix required by the original netlists will be represented by a set of synthetic netlists using domain circuit generation. For example, a domain may be specified by two netlists, netlist A and netlist B. Netlist A has 100 units, where 90% are ALUs and 10% are multipliers. Netlist B has 100 units, where 90% are multipliers and 10% are ALUs. A circuit is generated from the range specified by the profiles of the two netlists. This circuit contains 100 units, 50% of which are ALUs and 50% of which are multipliers. If a domain architecture is created from this one netlist alone, neither of the two original netlists could be implemented due to logic constraints.

To examine the issue of logic requirements, the eight application domains were profiled to give a range of characteristics. Next, five synthetic netlists were created from each of these range profiles, with the number of logic nodes increased by 0, 10, 20, and 30 percent beyond the number chosen by the domain circuit generation. For each domain (and each level of logic increase), the synthetic netlists were examined to determine if

they would create an architecture with sufficient logic resources to implement the original netlists of the domain. This process was repeated ten times for each domain.

Table 9.3 lists the success rate as a percentage across all ten trials for each domain, for each of the four logic sizes. The results demonstrate that even by adding 30% more units to each synthetic netlist, it is by no means guaranteed that there will be sufficient logic of the correct type to implement the original netlists. Choosing to add no additional units results in less than a 60% success rate. Therefore, an improved method is required to reliably specify logic requirements for a domain.

However, this particular test does lead to an interesting result. In the synthetic circuit generation, the routing complexity is given as a proportion to the number of logic units. Therefore, increasing the number of logic units for a synthetic circuit by a percentage also increases the number of signals within the circuit, and thus the number of wires in the final architecture. In fact, testing the routability of the parent netlists onto the GH architectures with 10% extra logic indicated that this increase was quite significant. Only one netlist-architecture combination out of 310 cases fails to route when enough logic exists to implement the netlist. A further test indicated that all of the scenarios with sufficient logic for placement route successfully when either the AMO or AML architecture generation methods are used. These results demonstrate that a percentage-based increase of the logic resources of synthetic circuits is not a good technique to use when attempting to differentiate between the flexibilities of the generated architectures.

Table 9.3: A table indicating the percentage of architectures having enough logic to implement the given original netlists. Results are given for the cases when no additional logic is added to the synthetic circuit, as well as when 10, 20, and 30 percent additional logic units are added.

Application	Netlists	0	10	20	30
Radar	decnsr	100	100	100	100
	fft16_2nd	0	0	10	10
	psd	100	100	100	100
OFDM	sync	20	40	40	40
	fft64	60	60	60	80
Camera	color_interp	100	100	100	100
	img_filt	80	90	90	90
	med_filt	100	100	100	100
Speech	log32	0	0	10	10
	fft32	100	100	100	100
	1d_dct40	100	100	100	100
FIR	firms	70	70	90	90
	firms2	70	70	90	90
	firms3	70	70	90	90
	firsyeven	0	0	0	0
	firtm_1st	100	100	100	100
	firtm_2nd	100	100	100	100
Matrix	matmult	10	20	60	60
	matmult4	30	30	70	80
	matmult_bit	10	20	60	60
	limited	10	30	50	60
	limited2	100	100	100	100
Sort	sort_g	20	50	90	100
	sort_rb	0	80	90	100
	sort_2d_g	90	100	100	100
	sort_2d_rb	100	100	100	100
Image	med_filt	0	0	0	0
	matmult	80	80	80	90
	firtm_2nd	80	90	100	100
	fft16_2nd	30	50	60	80
	1d_dct40	100	100	100	100
AVERAGE		59.0	66.1	75.5	78.4

Table 9.4: Success rates, in percentages, of routing original netlists onto architectures created by each of the three flexible routing generation algorithms from a set of synthetic benchmarks created from a domain profile. The minimum logic requirements of the domain were imposed on the architectures to assure a successful placement.

Application	Netlists	GH	AMO	AML
Radar	decnsr	100.0	100.0	100.0
	fft16_2nd	10.0	100.0	100.0
	psd	100.0	100.0	100.0
OFDM	sync	100.0	100.0	100.0
	fft64	60.0	100.0	90.0
Camera	color_interp	100.0	100.0	100.0
	img_filt	100.0	100.0	100.0
	med_filt	100.0	100.0	100.0
Speech	log32	80.0	100.0	100.0
	fft32	100.0	100.0	100.0
	1d_dct40	100.0	100.0	100.0
FIR	firms	100.0	100.0	100.0
	firms2	100.0	100.0	100.0
	firms3	100.0	100.0	100.0
	firmsyeven	50.0	100.0	100.0
	firtm_1st	100.0	100.0	100.0
	firtm_2nd	100.0	100.0	100.0
Matrix	matmult	90.0	100.0	100.0
	matmult4	90.0	100.0	100.0
	matmult_bit	90.0	100.0	100.0
	limited	100.0	100.0	100.0
	limited2	100.0	100.0	100.0
Sort	sort_g	100.0	100.0	100.0
	sort_rb	100.0	100.0	100.0
	sort_2d_g	100.0	100.0	100.0
	sort_2d_rb	100.0	100.0	100.0
Image	med_filt	70.0	90.0	90.0
	matmult	100.0	100.0	100.0
	firtm_2nd	100.0	100.0	100.0
	fft16_2nd	100.0	100.0	100.0
	1d_dct40	100.0	100.0	100.0
AVERAGE		91.6	99.7	99.4

The next domain tests allow for the specification of a minimum logic set. In this flow, the domains are profiled as before, but this time the minimum logic requirements are also profiled. Again five synthetic circuits are generated for each domain (with no additional logic). Architectures are created using the synthetic circuits for each of the domain. If the synthetic circuits do not provide sufficient logic resources of the

necessary types (based on the domain minimums), the needed logic units are added directly to the architecture without affecting the netlists. The architectures are therefore guaranteed to have sufficient logic for the original netlists of their domain. The original netlists of the domain are then placed and routed onto the architectures. This process was repeated ten times for each domain, and the results are given in Table 9.4.

This set of domain tests differentiate between the flexibility of the three different architecture generation methods. GH has the lowest flexibility, successfully routing 91.6% of netlists onto the architecture created for their domain. AML had a mid-level flexibility, with a success rate of 99.4%. AMO has the highest flexibility, with 99.7% of the netlists successfully routed. These results mirror the predicted flexibility of the three algorithms. GH is inherently less flexible, as it attempts to customize the track-based routing architecture as much as possible to the specification netlists, which in this case were synthetic. AMO has the highest flexibility, as it emphasizes the use of distance (segmented) routing tracks, which inherently permit a wider variety of choices to the routing algorithm. AML creates a regular routing architecture, more generic than the routing created by GH. Its flexibility is quite similar to that of AMO, the other algorithm that creates regular routing architectures. On the other hand, AML attempts to use less area than AMO by using local (non-segmented) routing whenever possible, and so its flexibility is slightly lower.

9.4 Other Uses

Synthetic circuit generators might also be used to compare the flexibility of existing architectures for domain-specific applications. The current flexibility testing involves generating architectures from synthetic circuits, and testing using the real circuits. This would not be possible in this case, as the architecture is already fixed. One possible method would be to select a set of netlists representative of the domain, and generate a large number of synthetic circuits based upon the profiled domain information. Next, the designer would attempt to place and route these synthetic circuits onto the architectures. The relative flexibility of each architecture within the application domain could then be measured by the percentage of the netlists which can successfully be implemented.

There are also other situations in which synthetic netlists and domain generation can be useful for automatic reconfigurable architecture generation. In many cases, the target domain may not be completely specified—either the final netlists are not all complete, or the devices will be expected to implement future netlists beyond the specification. If some characteristics of these netlists are known in advance, or at least a few of the domain's netlists are available, synthetic circuits can be created to approximate the unknown circuits for the architecture generation.

9.5 Summary

The flexible routing generation techniques of Chapter 7 provide solutions at various points in the area/flexibility solution space. While the methods can easily be compared on the basis of area, there was not an established method to evaluate the flexibility of the generated architectures. This chapter therefore discussed a method to perform this flexibility comparison through the use of synthetic circuits. Both the single-circuit flexibility testing and the enforced minimum domain testing were able to clearly differentiate between the architectures generated using the three routing generation methods. The results given for this comparison match predictions from Chapter 7 based on the goals of each of the algorithms. Greedy Histogram algorithm produces the least flexible (most specialized) architectures, and Add Max Once technique produces the most flexible architectures. Add Min Loop produces architectures with flexibility close to, but lower than, Add Max Once, trading some flexibility for area savings.

There are other potential uses for the synthetic circuit generation techniques described here. A synthetic circuit can be used to approximate a circuit whose design has not yet been completed, allowing for a greater degree of parallelism in the design of the reconfigurable hardware and the designs which will use it. Furthermore, using the domain circuit generation, a more full description of a domain can be created to help ensure an architecture is sufficiently flexible for future netlists created after the SoC fabrication. Most importantly, though, the flexibility measurement techniques presented

in this chapter will allow SoC designers to more thoroughly and intelligently evaluate different custom architecture generation techniques to find the best solution for the task at hand.

Chapter 10

Conclusions

Because of its flexibility and ability to run applications in hardware instead of software, reconfigurable hardware is well-suited for use on Systems-on-a-Chip (SoCs). Structures such as pre-existing FPGA tiles could be used for this purpose. However, commercial FPGAs are very generic, and miss many optimization opportunities for coarse-grained computations such as DSP. Coarse-grained reconfigurable architectures have been designed to improve efficiency, but they still target a broad spectrum of computations. As demonstrated by the area comparisons to RaPiD, further customization can yield significant area improvements.

However, custom fabricated SoCs allow for the possibility of customized reconfigurable logic. By creating specialized reconfigurable architectures for the targeted application domain, an architecture can be created that possesses the ASIC benefits of custom computational and routing resources, while still leveraging the assets of reconfigurable computing. These custom reconfigurable architectures can then be embedded into SoCs, yielding highly efficient computing solutions.

Unfortunately, the cost in design time and effort involved in the manual creation of a new reconfigurable architecture for each type of SoC would be prohibitive. The Totem Project seeks to solve this problem by automating the process of custom reconfigurable architecture creation to quickly and easily provide flexible and powerful acceleration circuits for SoCs.

10.1 Contributions

This work focused on one aspect of the Totem Project: high-level architecture generation. The architecture generator reads a set of application netlists as input, and outputs an architecture designed specifically for those netlists. A number of algorithms were presented which created architectures with varying trade-offs in terms of area and flexibility.

Algorithms were presented in Chapter 6 to generate reconfigurable architectures close in style to an ASIC, harnessing the specialization benefits of ASICs, yet providing the hardware re-use of a reconfigurable architecture. These techniques yielded architectures on average up to 12.3x smaller than the equivalent FPGA solution, and 2.2x smaller than standard cells for a given application. In Chapter 7, algorithms were examined that create architectures with a greater degree of flexibility, capable of implementing netlists beyond the specification set. Area improvements of up to 5.5x over an FPGA implementation were achieved.

The design of flexible routing architectures highlighted several additional key issues that must be considered for custom reconfigurable architecture design. The first is *track placement*, which is the arrangement of the routing resources with a channel. Chapter 8 defined the track placement problem and a metric to measure track placement quality. It also discussed several algorithms that perform automatic track placement, and compared the relative track placement qualities achieved by these algorithms. While a provably optimal algorithm was given, this algorithm is highly restricted. A relaxed version of the algorithm was also presented that achieved track placement qualities within 1.13% of the optimal brute-force solution.

Finally, a method to compare the relative flexibilities of the generated architectures was proposed. Currently, the only metric available to compare the ability of FPGA structures to implement circuits is “effective gates”. However, this metric refers to logic capacity, not necessarily the flexibility of the routing network. Also, this metric has been widely criticized, and is even less appropriate for coarse-grained or domain-specific architectures. Chapter 9 discussed the use of a synthetic circuit generator to test flexibility. The relative flexibilities of the three flexible architecture generation algorithms from Chapter 7 were then measured. As expected from the goals of the individual generation algorithms, these results indicate that Greedy Histogram is the least flexible of the three, Add Max Once is the most flexible, and Add Min Loop has a flexibility comparable, but slightly lower than, Add Max Once.

This work provides a robust framework to create and evaluate automatically generated customized reconfigurable architectures on the basis of both area and flexibility. Because these architectures are generated automatically, several different architectures designs can be created in a small fraction of the time required for a single manual design. SoC designers then have the ability to explore and compare architecture designs at various points of the solution space in order to find the best Totem architecture for each SoC.

10.2 Future Work

Currently, the pipelining of the netlists is removed prior to architecture generation to simplify the problem. However, the types of coarse-grained computational domains that Totem targets is likely to include highly pipelined netlists. It is possible sufficient pipelining resources, such as an optional pipeline register at each segmentation point, are available to route the full netlists onto the generated architectures. However, the effects of pipelining on the architecture generation should be studied.

While the techniques presented here for automatic reconfigurable architecture generation provide significant improvements over other solutions, several improvements can be made to the algorithms involved. An improved clique partitioning weight function for the cASIC generation of Chapter 6 which uses a combination of common ports and a common span to merge similar signals into physical wires could be examined. Furthermore, segmentation points could be employed to potentially reduce the wire cross-

section within the architecture and provide a higher degree of routing resource sharing between netlists.

The algorithms used for the flexible architecture generation in Chapter 7 can also be refined, with the Greedy Histogram method altered to consider situations where signals can be implemented using wires of varying lengths. Another regular routing algorithm could be researched which attempts to harness the benefits of both the Add Max Once and the Add Min Loop algorithms, emphasizing the cost of both the number of segmentation points and the number of routing tracks.

Finally, the circuit generator used by the flexibility tester discussed in Chapter 9, that creates synthetic circuits of appropriate similarity to the original source netlists, is a first effort at coarse-grained circuit generation. A few characteristics of the synthetic circuits deviate significantly from the source signal characteristics, and future versions of the synthetic circuit generator should attempt to rectify this issue.

References

- [Abnous96] A. Abnous, J. Rabaey, "Ultra-Low-Power Domain-Specific Multimedia Processors", *Proceedings of the IEEE VLSI Signal Processing Workshop*, October 1996.
- [Abnous98] A. Abnous, K. Seno, Y. Ichikawa, M. Wan, J. Rabaey, "Evaluation of a Low-Power Reconfigurable DSP Architecture", *Proceedings of the Reconfigurable Architectures Workshop*, 1998.
- [Actel01] Actel Corporation, *VariCoreTM Embedded Programmable Gate Array Core (EPGATM) 0.18 μ m Family*. Actel Corporation, Sunnyvale, CA, 2001.
- [Actel02] Actel Corporation, *ProASICTM 500K Family*. Actel Corporation, Sunnyvale, CA, 2002.
- [Atmel02] Atmel Corporation, *AT94K Series FPSLIC*. Atmel Corporation, San Jose, CA, 2002.
- [Aggarwal94] A. Aggarwal, D. Lewis, "Routing Architectures for Hierarchical Field Programmable Gate Arrays", *Proceedings of the IEEE International Conference on Computer Design*, pp. 475-478, 1994.
- [Altera98] Altera Corporation, *Data Book*. Altera Corporation, San Jose, CA, 1998.
- [Altera01] Altera Corporation, *Excalibur: Embedded Processor Programmable Solutions*. Altera Corporation, San Jose, CA, 2002.
- [Altera03a] Altera Corporation, *Stratix Device Handbook, Volume 1*. Altera Corporation, San Jose, CA, 2003.
- [Altera03b] Altera Corporation, *Cyclone: The Lowest-Cost FPGA Ever*. Altera Corporation, San Jose, CA, 2003.
- [Annapolis98] Annapolis Microsystems, Inc., *Wildfire Reference Manual*. Annapolis Microsystems, Inc, Annapolis, MD, 1998.
- [Arnold92] J. M. Arnold, D. A. Buell, E. G. Davis, "Splash 2", *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 316-324, 1992.
- [Betz97] V. Betz, J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", *International Workshop on Field Programmable Logic and Applications*, pp. 213-222, 1997.
- [Betz99] V. Betz, J. Rose, "FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density", *ACM/SIGDA International Symposium on FPGAs*, pp. 59-68, 1999.

- [Betz00] V. Betz, J. Rose, "Automatic Generation of FPGA Routing Architectures from High-Level Descriptions", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 175 – 184, 2000.
- [Borriello95] G. Borriello, C. Ebeling, S. Hauck, S. Burns, "The Triptych FPGA Architecture", *IEEE Transactions on VLSI Systems*, Vol. 3, No. 4, pp. 491-501, December 1995.
- [Brown92a] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic, *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, Boston, MA, 1992.
- [Brown92b] S. Brown, J. Rose, Z. G. Vranesic, "A Detailed Router for Field-Programmable Gate Arrays", *IEEE Transactions on Computer-Aided Design*, Vol. 11, No. 5, pp. 620-628, 1992.
- [Budiu99] M. Budiu, S. C. Goldstein, "Fast Compilation for Pipelined Reconfigurable Fabrics", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1999.
- [Buell96] D. Buell, J. M. Arnold, W. J. Kleinfelder, *SPLASH 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, Los Alamitos, CA, 1996.
- [Cadambi98] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, D. E. Thomas, "Managing Pipeline-Reconfigurable FPGAs", *ACM/SIGDA International Symposium on FPGAs*, pp. 55-64, 1998.
- [Callahan98] T. J. Callahan, P. Chong, A. DeHon, J. Wawrzynek, "Fast Module Mapping and Placement for Datapaths in FPGAs", *ACM/SIGDA International Symposium on FPGAs*, pp. 123-132, 1998.
- [Chameleon00] Chameleon Systems, Inc., *CS2000 Advance Product Specification*. Chameleon Systems, Inc., San Jose, CA, 2000.
- [Chan97] P. K. Chan, M. D. F. Schlag, "Acceleration of an FPGA Router", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 175-181, 1997.
- [Chang99] M. L. Chang, S. Hauck, "Adaptive Computing in NASA Multi-Spectral Image Processing", *Military and Aerospace Applications of Programmable Devices and Technologies International Conference*, 1999.
- [Chipworks02] Chipworks, Inc., "Xilinx XC2V1000 Die Size And Photograph", Chipworks, Inc., Ottawa, Canada, 2002.
- [Chow99] P. Chow, S. O. Seo, J. Rose, K. Chung, G. Páez-Monzon, I. Rahardja, "The Design of an SRAM-Based Field-Programmable Gate Array—Part I: Architecture", *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 7, No. 2, pp. 191-197, 1999.
- [Compton00] K. Compton, J. Cooley, S. Knol, S. Hauck, "Configuration Relocation and Defragmentation for FPGAs", *Northwestern University Technical Report*, Available online at <http://www.ece.nwu.edu/~kati/publications.html>, 2000.

- [Compton01] K. Compton, S. Hauck, "Totem: Custom Reconfigurable Array Generation", *IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.
- [Compton02a] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", *ACM Computing Surveys*, Vol. 34, No. 2. pp. 171-210, June 2002.
- [Compton02b] K. Compton, S. Hauck, "Track Placement: Orchestrating Routing Structures to Maximize Routability", *University of Washington Technical Report UWEETR-2002-0013*, 2002.
- [Compton02c] K. Compton, Z. Li, J. Cooley, S. Knol, "Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing", *IEEE Transactions on VLSI*, Vol. 10, No. 3., pp. 209-220, June 2002.
- [Compton02d] K. Compton, A. Sharma, S. Phillips, S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", *International Conference on Field-Programmable Logic and Applications*, pp. 59-68, 2002.
- [Compton03] K. Compton, S. Hauck, "Track Placement: Orchestrating Routing Structures to Maximize Routability", *International Conference on Field-Programmable Logic and Applications*, 2003.
- [Cong98] J. Cong, S. Xu, "Technology Mapping for FPGAs with Embedded Memory Blocks", *ACM/SIGDA International Symposium on FPGAs*, pp. 179-188, 1998.
- [Cronquist98] D. C. Cronquist, P. Franklin, S.G. Berg, C. Ebeling, "Specifying and Compiling Applications for RaPiD", *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.
- [Cronquist99a] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Twentieth Anniversary Conference on Advanced Re-search in VLSI*, 1999.
- [Cronquist99b] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Presentation at Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999.
- [Cypress03] Cypress MicroSystems, *PSoCTM: Configurable Mixed-Signal Array with On-board Controller*. Cypress MicroSystems, Lynnwood, WA, 2003.
- [Dandalis01] A. Dandalis, V. K. Prasanna, "Configuration Compression for FPGA-based Embedded Systems", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 173-182, 2001.
- [Darnauer96] J. Darnauer, W.W.-M. Dai, "A Method for Generating Random Circuits and its Application to Routability Measurement", *ACM Symposium on Field Programmable Gate Arrays*, 1996.

- [DeHon99] A. DeHon, "Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100% LUT utilization)", *ACM/SIGDA International Symposium on FPGAs*, pp. 69-78, 1999.
- [Deshpande99] D. Deshpande, A. K. Somani, A. Tyagi, "Configuration Caching Vs Data Caching for Striped FPGAs", *ACM/SIGDA International Symposium on FPGAs*, pp. 206-214, 1999.
- [Diessel97] O. Diessel, H. ElGindy, "Run-Time Compaction of FPGA Designs", *Lecture Notes in Computer Science 1304—Field-Programmable Logic and Applications*. W. Luk, P.Y.K. Cheung, M. Glesner, Eds. Springer-Verlag, Berlin, Germany, pp. 131-140, 1997.
- [Dollas98] A. Dollas, E. Sotiriades, A. Emmanouelides, "Architecture and Design of GE1, a FCCM for Golomb Ruler Derivation", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 48-56, 1998.
- [Dorndorf94] U. Dorndorf, E. Pesch, "Fast Clustering Algorithms", *ORSA Journal on Computing*, Vol. 6, No. 2, pp. 141-152, 1994.
- [eASIC03] eASIC Corporation, "eASICore[®] Product Features", Available Online at: http://www.easic.com/technology/product_feature.html, 2003.
- [Ebeling96] C. Ebeling, D. C. Cronquist, P. Franklin, "RaPiD – Reconfigurable Pipelined Datapath.", *Lecture Notes in Computer Science 1142—Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, R.W. Hartenstein, M. Glesner, Eds. Springer-Verlag, Berlin, Germany, pp. 126-135, 1996.
- [Elbirt00] A. J. Elbirt, C. Paar, "An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher", *ACM/SIGDA International Symposium on FPGAs*, pp. 33-40, 2000.
- [Elixent02] Elixent, *DFA1000 RISC Accelerator*. Elixent, Bristol, England, 2002.
- [Emmert99] J. M. Emmert, D. Bhatia, "A Methodology for Fast FPGA Floorplanning", *ACM/SIGDA International Symposium on FPGAs*, pp. 47-56, 1999.
- [Estrin63] G. Estrin, B. Bussel, R. Turn, J. Bibb, "Parallel Processing in a Restructurable Computer System", *IEEE Transactions on Electronic Computers*, pp. 747-755, 1963.
- [Fry02] T. W. Fry, S. Hauck, "Hyperspectral Image Compression on Reconfigurable Platforms", *Earth Science Technology Conference*, 2002.
- [Gehring96] S. Gehring, S. Ludwig, "The Trianus System and Its Application to Custom Computing", *Lecture Notes in Computer Science 1142—Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*. R.W. Hartenstein, M. Glesner, Eds. Springer-Verlag, Berlin, Germany, pp. 176-184, 1996.

- [Goldstein00] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler", *IEEE Computer*, Vol. 33, No. 4, pp. 70-77, 2000.
- [Graham96] P. Graham, B. Nelson, "Genetic Algorithms In Software and In Hardware--A Performance Analysis of Workstations and Custom Computing Machine Implementations", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 216-225, 1996.
- [Hashimoto71] A. Hashimoto, J. Stevens, "Wire Routing by Optimizing Channel Assignment within Large Apertures", *Proceedings of the 8th ACM Design Automation Workshop*, pp. 115-169, 1971.
- [Hauck96] S. Hauck, A. Agarwal, "Software Technologies for Reconfigurable Systems", *Northwestern University, Dept. of ECE Technical Report*. Available online at: <http://www.ee.washington.edu/faculty/hauck/publications.html>, 1996.
- [Hauck97] S. Hauck, T. W. Fry, M. M. Hosler, J. P. Kao, "The Chimaera Reconfigurable Functional Unit", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 87-96, 1997.
- [Hauck98a] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems", *Proceedings of the IEEE*, Vol. 86, No. 4, pp. 615-638, 1998.
- [Hauck98b] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors", *ACM/SIGDA International Symposium on FPGAs*, pp. 65-74, 1998.
- [Hauck98c] S. Hauck, Z. Li, E. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 138-146, 1998.
- [Hauck99] S. Hauck, W. D. Wilson, "Runlength Compression Techniques for FPGA Configurations", *Northwestern University, Dept. of ECE Technical Report*, Available online at: <http://www.ee.washington.edu/faculty/hauck/publications.html>, 1999.
- [Hauser97] J. R. Hauser, J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 12-21, 1997.
- [Hauser00] J. R. Hauser, *Augmenting a Processor with Reconfigurable Hardware*. PhD Dissertation. University of California at Berkeley, 2000.
- [Haynes98] S. D. Haynes, P. Y. K. Cheung, "A Reconfigurable Multiplier Array For Video Image Processing Tasks, Suitable for Embedding in an FPGA Structure", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 226-234, 1998.

- [Heile99] F. Heile, A. Leaver, "Hybrid Product Term and LUT Based Architectures Using Embedded Memory Blocks", *ACM/SIGDA International Symposium on FPGAs*, pp. 13-16, 1999.
- [Huang00] W. J. Huang, N. Saxena, E. J. McCluskey, "A Reliable LZ Data Compressor on Reconfigurable Coprocessors", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 249-258, 2000.
- [Huang01] Z. Huang, S. Malik, "Managing Dynamic Reconfiguration Overhead in Systems-on-a-Chip Design Using Reconfigurable Datapaths and Optimized Interconnection Networks", *Conference of Design Automation and Test in Europe (DATE)*, 2001.
- [Huelsbergen00] L. Huelsbergen, "A Representation for Dynamic Graphs in Reconfigurable Hardware and its Application to Fundamental Graph Algorithms", *ACM/SIGDA International Symposium on FPGAs*, pp. 105-115, 2000.
- [Hutton98] M. Hutton, J. Rose, J. Grossman, and D. Corneil, "Characterization and Parameterized Generation of Synthetic Combinational Benchmark Circuits", *IEEE Transactions on CAD*, Vol. 17, No. 10, pp. 985-996, October 1998.
- [Hutton02] M. Hutton, J. Rose and D. Corneil, "Automatic Generation of Synthetic Sequential Benchmark Circuits", *IEEE Transactions on CAD*, Vol. 21, No. 8, pp. 928-940, August 2002.
- [Kastrup99] B. Kastrup, A. Bink, J. Hoogerbrugger, "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 92-101, 1999.
- [Kim00a] H. J. Kim, W. H. Mangione-Smith, "Factoring Large Numbers with Programmable Hardware", *ACM/SIGDA International Symposium on FPGAs*, pp. 41-48, 2000.
- [Kim00b] H. S. Kim, A. K. Somani, A. Tyagi, "A Reconfigurable Multi-function Computing Cache Architecture", *ACM/SIGDA International Symposium on FPGAs*, pp. 85-94, 2000.
- [Krupnova97] H. Krupnova, C. Rabedaoro, G. Saucier, "Synthesis and Floorplanning For Large Hierarchical FPGAs", *ACM/SIGDA International Symposium on FPGAs*, pp. 105-111, 1997.
- [Lai97] Y. T. Lai, P. T. Wang, "Hierarchical Interconnection Structures for Field Programmable Gate Arrays", *IEEE Transactions on VLSI Systems*, Vol. 5, No. 2, pp. 186-196, 1997.
- [Lattice03] Lattice Semiconductor Corporation, *ORCA[®] Series 4 FPGAs: Data Sheet*. Lattice Semiconductor Corporation, Hillsboro, OR, 2003.
- [Laufer99] R. Laufer, R. R. Taylor, H. Schmit, "PCI-PipeRench and the SwordAPI: A System for Stream-based Reconfigurable Computing", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 200-208, 1999.

- [Lemieux02] G. Lemieux, D. Lewis, "Analytical Framework for Switch Block Design", *International Conference on Field-Programmable Logic and Applications*, pp. 122 - 131, 2002.
- [Lemieux03] G. Lemieux, D. Lewis, *Design of Interconnection Networks for Programmable Logic*. Kluwer Academic Publishers, Boston, MA, 2003.
- [LeopardLogic03] LeopardLogic, Inc., <http://www.leopardlogic.com>, 2003.
- [Leung00] K. H. Leung, K. W. Ma, W. K. Wong, P. H. W. Leong, "FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 68-76, 2000.
- [Li99] Z. Li, S. Hauck "Don't Care Discovery for FPGA Configuration Compression", *ACM/SIGDA International Symposium on FPGAs*, pp. 91-98, 1999.
- [Li00] Z. Li, K. Compton, S. Hauck, "Configuration Caching for FPGAs", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 22-36, 2000.
- [Li02] Z. Li, *Configuration Management Techniques for Reconfigurable Computing*, Ph.D. Thesis, Northwestern University, Dept. of ECE, 2002.
- [Lucent98] Lucent Technologies, Inc., *FPGA Data Book*. Lucent Technologies, Inc., Allentown, PA, 1998.
- [Luk97] W. Luk, N. Shirazi, P. Y. K. Cheung, "Compilation Tools for Run-Time Reconfigurable Designs", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 56-65, 1997.
- [M2000-02] M2000, *Press Release – May 15, 2002*. M2000, Bièvres, France, 2002.
- [Mangione-Smith97] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prasanna, H. A. E. Spaanenburg, "Seeking Solutions in Configurable Computing", *IEEE Computer*, Vol. 30, No. 12, pp. 38-43, 1997.
- [Mangione-Smith99] W. H. Mangione-Smith, "ATR from UCLA", *Personal Communications*, 1999.
- [Marshall99] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, B. Hutchings, "A Reconfigurable Arithmetic Array for Multimedia Applications", *ACM/SIGDA International Symposium on FPGAs*, pp. 135-143, 1999.
- [McMurchie95] L. McMurchie, C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *ACM Third International Symposium on Field-Programmable Gate Arrays*, pp. 111-117, 1995.
- [Miyamori98] T. Miyamori, K. Olukoton, "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 2-11, 1998.

- [Moritz98] C. A. Moritz, D. Yeung, A. Agarwal, "Exploring Optimal Cost Performance Designs for Raw Microprocessors", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 12-27, 1998.
- [Phillips02] S. Phillips, S. Hauck, "Automatic Layout of Domain-Specific Reconfigurable Subsystems for System-on-a-Chip", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2002.
- [QuickLogic02] QuickLogic Corporation, *QuickMIPS™ QL90xM: Multi-application System-on-a-Chip*. QuickLogic Corporation, Sunnyvale, CA, 2002.
- [QuickSilver03] QuickSilver Technology, Inc. <http://www.quicksilverttech.com>, 2003.
- [Quickturn99a] Quickturn, A Cadence Company, *System Realizer™*, Available online at <http://www.quickturn.com/products/systemrealizer.htm>. Quickturn, A Cadence Company, San Jose, CA, 1999.
- [Quickturn99b] Quickturn, A Cadence Company, *Mercury™ Design Verification System Technology Backgrounder*. Available online at http://www.quickturn.com/products/mercury_backgrounder.htm. Quickturn, A Cadence Company, San Jose, CA, 1999.
- [Razdan94] R. Razdan, M. D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", *International Symposium on Microarchitecture*, pp. 172-180, 1994.
- [Rencher97] M. Rencher, B. L. Hutchings, "Automated Target Recognition on SPLASH2", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 192-200, 1997.
- [Rose93] J. Rose, A. El Gamal, A. Sangiovanni-Vincentelli, "Architecture of Field-Programmable Gate Arrays", *Proceedings of the IEEE*, Vol. 81, No. 7, 1013-1029, 1993.
- [Sheir99] D. R. Shier, "Matchings", *Handbook of Discrete and Combinatorial Mathematics*, Kenneth H. Rosen, Editor-in-Chief. CRC Press LLC, Boca Raton, FL. 1999. pp. 641-652.
- [Rupp98] C. R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. M. Arnold, M. Gokhale, "The NAPA Adaptive Processing Architecture", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 28-37, 1998.
- [Sankar99] Y. Sankar, J. Rose, "Trading Quality for Compile Time: Ultra-Fast Placement for FPGAs", *ACM/SIGDA International Symposium on FPGAs*, pp. 157-166, 1999.
- [Scalera98] S. M. Scalera, J. R. Vazquez, "The Design and Implementation of a Context Switching FPGA", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 78-85, 1998.

- [Scott01] M. Scott, "The RaPiD Cell Structure", *Personal Communications*, 2001.
- [Sechen88] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*. Kluwer Academic Publishers, Boston, MA, 1988.
- [Senouci98] S. A. Senouci, A. Amoura, H. Krupnova, G. Saucier, "Timing Driven Floorplanning on Programmable Hierarchical Targets", *ACM/SIGDA International Symposium on FPGAs*, pp. 85-92, 1998.
- [Shahookar91] K. Shahookar, P. Mazumder, "VLSI Cell Placement Techniques", *ACM Computing Surveys*, Vol. 23, No. 2, pp. 145-220, 1991.
- [Sharma02] A. Sharma, *Development of a Place and Route Tool for the RaPiD Architecture*. Master's Project, University of Washington, December 2002.
- [Sharma03] A. Sharma, C. Ebeling, S. Hauck, "PipeRoute: A Pipelining-Aware Router for FPGAs", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, pp. 68-77, 2003.
- [Shi97] J. Shi, D. Bhatia, "Performance Driven Floorplanning for FPGA Based Designs", *ACM/SIGDA International Symposium on FPGAs*, pp. 112-118, 1997.
- [Shirazi98] N. Shirazi, W. Luk, P. Y. K. Cheung, "Automating Production of Run-Time Reconfigurable Designs", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 147-156, 1998.
- [SiliconHive03] Silicon Hive, *Silicon Hive Technology Primer*. Phillips Electronics NV, The Netherlands, 2003.
- [Sotiriades00] E. Sotiriades, A. Dollas, P. Athanas, "Hardware-Software Codesign and Parallel Implementation of a Golomb Ruler Derivation Engine", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 227-235, 2000.
- [Stone96] G. O. Stone, *A Comparison of ASIC Implementation Alternatives*, M.S. Thesis, Northwestern University, Dept. of ECE, October, 1996.
- [Swartz98] J. S. Swartz, V. Betz, J. Rose, "A Fast Routability-Driven Router for FPGAs", *ACM/SIGDA International Symposium on FPGAs*, pp. 140-149, 1998.
- [Takahara98] A. Takahara, T. Miyazaki, T. Murooka, M. Katayama, K. Hayashi, A. Tsutsui, T. Ichimori, K. Fukami, "More Wires and Fewer LUTs: A Design Methodology for FPGAs", *ACM/SIGDA International Symposium on FPGAs*, pp. 12-19, 1998.
- [Trimberger97] S. Trimberger, K. Duong, B. Conn, "Architecture Issues and Solutions for a High-Capacity FPGA", *ACM/SIGDA International Symposium on FPGAs*, pp. 3-9, 1997.
- [Triscend02] Triscend Corporation, *Triscend A7S Configurable System-on-Chip Platform: Product Description*. Triscend Corporation, Mountain View, CA, 2002.

- [Triscend03] Triscend Corporation, *Triscend E5 Customizable Microcontroller Platform: Product Description*. Triscend Corporation, Mountain View, CA, 2002.
- [Tsu99] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, A. DeHon, "HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array", *ACM/SIGDA International Symposium on FPGAs*, pp. 125-134, 1999.
- [Vuillemin96] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", *IEEE Transactions on VLSI Systems*, Vol. 4, No. 1, pp. 56-69, 1996.
- [Weinhardt99] M. Weinhardt, W. Luk, "Pipeline Vectorization for Reconfigurable Systems", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 52-62, 1999.
- [Wilton98] S. J. E. Wilton, "SMAP: Heterogeneous Technology Mapping for Area Reduction in FPGAs with Embedded Memory Arrays", *ACM/SIGDA International Symposium on FPGAs*, pp. 171-178, 1998.
- [Wilton01] S. Wilton, J. Rose, Z. Vranesic, "Structural Analysis and Generation of Synthetic Digital Circuits with Memory", *IEEE Transactions on VLSI*, Vol. 9, No. 1, pp. 223-226, February 2001.
- [Wirthlin95] M. J. Wirthlin, B. L. Hutchings, "A Dynamic Instruction Set Computer", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99-107, 1995.
- [Wirthlin96] M. J. Wirthlin, B. L. Hutchings, "Sequencing Run-Time Reconfigured Hardware with Software", *ACM/SIGDA International Symposium on FPGAs*, pp. 122-128, 1996.
- [Wittig96] R. D. Wittig, P. Chow, "OneChip: An FPGA Processor With Reconfigurable Logic", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 126-135, 1996.
- [Wood97] R. G. Wood, R. A. Rutenbar, "FPGA Routing and Routability Estimation Via Boolean Satisfiability", *ACM/SIGDA International Symposium on FPGAs*, pp. 119-125, 1997.
- [Wu97] Y. L. Wu, M. Marek-Sadowska, "Routing for Array-Type FPGA's", *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 16, No. 5, pp. 506-518, 1997.
- [Xilinx94] Xilinx, Inc., *The Programmable Logic Data Book*. Xilinx, Inc., San Jose, CA, 1994
- [Xilinx96] Xilinx, Inc., *XC6200: Advance Product Specification*. Xilinx, Inc., San Jose, CA, 1996.
- [Xilinx01] Xilinx, Inc., *Virtex™ 2.5 V Field Programmable Gate Arrays: Product Specification*. Xilinx, Inc., San Jose, CA, 1999.

- [Xilinx02] Xilinx, Inc., *Virtex™-II Platform FPGAs: Detailed Description*. Xilinx, Inc., San Jose, CA, 2002.
- [Xilinx03a] Xilinx, Inc., *Virtex-II Pro™ Platform FPGAs: Advance Product Specification*. Xilinx, Inc., San Jose, CA, 2003.
- [Xilinx03b] Xilinx, Inc., *PicoBlaze 8-Bit Microcontroller for Virtex-E and Spartan-II/III Devices*. Xilinx, Inc., San Jose, CA, 2003.
- [Xilinx03c] Xilinx, Inc., *PicoBlaze 8-Bit Microcontroller for Virtex-II Series Devices*. Xilinx, Inc., San Jose, CA, 2003.
- [Xilinx03d] Xilinx, Inc., *MicroBlaze Processor Reference Guide*. Xilinx, Inc., San Jose, CA, 2003.
- [Zhong98] P. Zhong, M. Martinosi, P. Ashar, S. Malik, “Accelerating Boolean Satisfiability with Configurable Hardware”, *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 186-195, 1998.