©Copyright 2008
Kenneth Eguro

Supporting High-Performance Pipelined Computation in Commodity-Style FPGAs

Kenneth Eguro

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

Doctor of Philosophy

University of Washington

2008

Program Authorized to Offer Degree:
Electrical Engineering

University of Washington
Graduate School

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

Kenneth Eguro

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

Chair of the Supervisory Committee:

____

Scott Hauck

Reading Committee:

____

Scott Hauck

____

W. H. Carl Ebeling

____

Mani Soma

Date: ____________________________

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

Signature________________________________

Date____________________________________

University of Washington

Abstract

Supporting High-Performance Pipelined Computation in Commodity-Style FPGAs

Kenneth Eguro

Chair of the Supervisory Committee:
Professor Scott Hauck
Electrical Engineering

Although the popularity of Field Programmable Gate Arrays, or FPGAs, is a testament to their unique mixture of flexibility and ease of use, this adaptability can come at price. The programmable nature of FPGAs introduces significant inefficiencies that can limit the maximum clock frequency of mapped circuits. While there are multiple techniques developers apply to mitigate this performance penalty, these enhancements can generate an enormous number of additional registers. These heavily registered circuits have fundamentally different characteristics and create significant problems for many different aspects of FPGA application development. This dissertation investigates the concerns that arise for both FPGA physical design tools and the architectures themselves.

FPGA Development Tools: High quality compilation tools are necessary to create fast and efficient FPGA-based applications. However, heavily registered circuits can confuse existing packing, placement, retiming, and routing tools. This dissertation examines the roots of these problems and suggests new timing-driven and register-aware physical design techniques. These new approaches are shown to significantly improve achievable results, potentially doubling the speed of mapped circuits.

FPGA Architectures: Heavily registered applications can also overwhelm the register resources provided by classical FPGA architectures. While there have been previous research efforts to build FPGAs with better register support, most have suggested very specialized systems that depart significantly from conventional architectures and toolflows. This dissertation explores a different approach and investigates the practical advantages of making minimally invasive architectural changes to both FPGA logic blocks and interconnect resources. These architectural choices can affect the required area of implemented designs by a factor two.

This dissertation shows that netlists with a large number of registers can significantly change the problems presented to CAD tools and the demands placed on FPGA architectures. Failing to acknowledge these changes can be costly. That said, some problems are likely more pressing than others. Furthermore, although this dissertation identifies many of the aspects of an FPGA architecture that can dramatically affect the required area of deeply pipelined or C-slowed applications, this work merely scratches the surface and much more research is necessary to determine what future FPGAs should look like.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables vi

Chapter 1: Introduction 1

Chapter 2: Field Programmable Gate Arrays 3

- 2.1: Conventional FPGA Architectures 3
- 2.2: FPGAs, Microprocessors and Application-Specific Integrated Circuits 5

Chapter 3: Pipelining, Retiming and C-Slowing 8

Chapter 4: FPGA Development Tools....12

- 4.1: FPGA CAD Toolflow 12
- 4.2: Packing....12
- 4.3: Placement 14
- 4.4: Routing....18
- 4.5: Issues for Heavily-Registered Applications 20

Chapter 5: Enhancing Timing-Driven Placement....22

- 5.1: Background on VPR Timing-Driven Placement 22
- 5.2: Implications of Static Timing Analysis....24
- 5.3: Characteristics of Registered Applications....31
- 5.4: Registered Netlists & Placement Stability....32
- 5.5: Efficient and Stable Placement....41
- 5.6: Delay Imbalance and Optimality....45
- 5.7: Testing and Results 49
- 5.8: Conclusions and Future Research 54

Chapter 6: Register-Aware Placement 57

- 6.1: Feed-Forward Design Flow – Implications for Packing, Retiming and Placement....57
- 6.2: Previous Retiming-Aware Approaches 59
- 6.2.1: Previous Architectural Retiming Solutions 60
- 6.2.2: Previous CAD Retiming Solutions....62
- 6.3: Integrated Placement and Physical Synthesis....65
- 6.3.1: Packing and FF-Level Placement....66
- 6.3.2: Retiming 69
- 6.4: Testing and Results 72
- 6.5: Conclusions and Future Research 82

Chapter 7: Register-Aware Routing 86

- 7.1: Registers with Limited Connectivity....86

7.2: Pipelined Routing Problem 88
7.2.1: PipeRoute 89
7.2.2: QuickRoute 92
7.3: Timing-Driven Pipelined Routing 93
7.3.1: Determining Link Criticality 94
7.3.2: Assumed Criticality Searching 96
7.3.3: New Cost Formulation 99
7.4: Armada 101
7.5: Testing and Results 104
7.6: Conclusions and Future Research 109
Chapter 8: Register-Enhanced Architectures 114
8.1: Scaling of CLB Requirements and Performance 114
8.2: Previous Register-Rich FPGAs 118
8.3: New Potentials for Increasing Register Capabilities 119
8.3.1: Potential of Registered Switchboxes 119
8.3.2: Enhancing Logic Blocks with Additional Registers 126
8.3.2.1: Using LUTs as Shift Registers 127
8.3.2.2: Adding Independent Flip-Flops 131
8.4: Evaluation and Results 134
8.5: Conclusions and Future Research 147
Chapter 9: Conclusions and Future Research 153
Bibliography 161

LIST OF FIGURES

Figure Number Page

Figure 2.1: Conventional Island-Style FPGA 3
Figure 3.1: Clock Frequency and Latency Effects of Pipelining 8
Figure 3.2: Two Examples of Unbalanced Delay Between Pipelining Registers 9
Figure 3.3: Limitations of Retiming and Demonstration of C-Slowing 10
Figure 4.1: Packing Restrictions [2] 13
Figure 4.2: Simulated Annealing Windowing 17
Figure 4.3: Pseudo-Code for VPR Timing-Driven Placement 18
Figure 4.4: Pseudo-Code for PathFinder Routing 20
Figure 5.1: Static Timing Analysis 24
Figure 5.2: Effect of Stale Criticality Information 26
Figure 5.3: VPR Placement with Stale Criticality Information 26
Figure 5.4: VPR Placement λ and Criticality Exponent Tuning for Conventional MCNC Netlists 29
Figure 5.5: Discrepancy in VPR Placement for Conventional Combinational and Sequential MCNC Netlists, (λ = 0.3, Criticality Exponent = 12) 31
Figure 5.6: Timing Implications of Combinational Logic vs. Registers 32
Figure 5.7: Registered Netlists & Placement Oscillation 33
Figure 5.8: VPR Placement Convergence Problem with Depth = 1 MCNC Netlist 35
Figure 5.9: VPR Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists, Phase 1 37
Figure 5.10: VPR Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists, Phase 2 38
Figure 5.11: Similarity in VPR Placement for Depth = 1 Combinational and Sequential MCNC Netlists, (λ = 0.3, Criticality Exponent = 8) 41
Figure 5.12: Incremental Slack, Criticality Updating and Accuracy 42
Figure 5.13: Problems with Perfect Timing Information 43
Figure 5.14: Calculating Relative Change in Criticality 44
Figure 5.15: Example of Contraction and Imbalance 45
Figure 5.16: Total Delay Contraction as a Function of Criticality Exponent and Balance 48
Figure 5.17: Solving for Maximum $Delay_z$ Values 48
Figure 5.18: Incremental Criticality Update Placement λ and Criticality Exponent Tuning for Conventional MCNC Netlists 50
Figure 5.19: Incremental Criticality Update Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists 51
Figure 5.20: Comparison Between VPR and Incremental Criticality Update Placement 52
Figure 6.1: Packing Implications for Heavily Registered Netlists 58
Figure 6.2: Track-Graph, Universal and Registered Track-Graph Switchboxes 60

Figure 6.3: Non-Independently and Independently Accessible Flip-Flop Architectures 64
Figure 6.4: Probability of LUT and Flip-Flop Separation Versus Reunion 67
Figure 6.5: Pseudo-Code for Incorporating FF-Level Placement Moves 68
Figure 6.6: Incorporating New Registers Created By Retiming 70
Figure 6.7: Updating Timing Information for New Retiming Registers... 71
Figure 6.8: Pseudo-Code for Simulated Annealing-Based Retiming 72
Figure 6.9: Logic Blocks Reachable with 1 Wire Segment 80
Figure 6.10: Logic Duplication.. 85
Figure 7.1: Impact of Connectivity on Area and Number of Switchbox Registers 87
Figure 7.2: Failure of Dijkstra’s Algorithm for the N-Delay Routing Problem 88
Figure 7.3: Combined-Phase Breadth-First Search 89
Figure 7.4: Failure of Combined-Phase BFS and Need for 2 Combined-Phase BFS 90
Figure 7.5: Greedy Accumulation of Multiple-Latency Routes 90
Figure 7.6: PipeRoute and Self Intersection 90
Figure 7.7: QuickRoute and Self-Intersection 92
Figure 7.8: QuickRoute and Self Blocking.. 92
Figure 7.9: Pseudo-Code for QuickRoute.. 93
Figure 7.10: Timing Implications for Conventional Routing Versus Pipelined Routing 95
Figure 7.11: Multi-Terminal Criticality Problem 96
Figure 7.12: Assumed Criticality Searching.. 98
Figure 7.13: Congestion vs. Timing Concerns For Pipelined Routing 100
Figure 7.14: QuickRoute with Assumed Criticality Searching... 101
Figure 7.15: Re-initializing PQ for Multi-Terminal Nets 102
Figure 7.16: Pseudo-Code for Armada Timing-Driven Pipelined Routing 103
Figure 7.17: Illustration of a RaPiD Cell 105
Figure 8.1: Combinational MCNC Netlists Critical Path Delay... 115
Figure 8.2: Sequential MCNC Netlists Critical Path Delay.. 115
Figure 8.3: Combinational MCNC Netlists CLB Requirements 116
Figure 8.4: Sequential MCNC Netlists CLB Requirements 116
Figure 8.5: Effect of Pipelining and Netlist Topology on CLB Requirement 117
Figure 8.6: Significant Delay Numbers for an Island-Style FPGA Information taken from 65nm four 4-LUT, length-4 wire FPGA architecture 120
Figure 8.7: Significant Delay Numbers for an Island-Style FPGA with Registered Switchboxes 122
Figure 8.8: Delay Contribution of Best-Case Scenarios (1 Wire Segment) for Registered Switchboxes... 125
Figure 8.9: Conventional BLE (left) and LUT/16-bit Shift-Register BLE (right) 128
Figure 8.10: LUT/Two 8-bit Shift-Register BLE 129

Figure 8.11: LUT/Four 4-bit Shift-Register BLE..130
Figure 8.12: Registered Netlists and Effect of Architecture Register Density on Critical Path Delay........133
Figure 8.13: BLE with Two Independent Flip-Flops 133
Figure 8.14: Testing Phase I – Effect of Additional Independent Flip-Flops and Shift Registers on Depth = 1 Netlists 137
Figure 8.15: Testing Phase I – Effect of Additional Independent Flip-Flops and Shift Registers on Depth = 0.33 Netlists 137
Figure 8.16: Architecture Exploration on Depth = 1 Netlists, 1 LUT/CLB has Shift Register(s)...............141
Figure 8.17: Architecture Exploration on Depth = 0.33 Netlists, 1 LUT/CLB has Shift Register(s)..........141
Figure 8.18: Architecture Exploration on Depth = 1 Netlists, 2 LUTs/CLB have Shift Register(s)...........144
Figure 8.19: Architecture Exploration on Depth = 0.33 Netlists, 2 LUTs/CLB have Shift Register(s)......144
Figure 8.20: Architecture Exploration on Depth = 1 Netlists, 4 LUTs/CLB have Shift Register(s)...........146
Figure 8.21: Architecture Exploration on Depth = 0.33 Netlists, 4 LUTs/CLB have Shift Register(s)......146

LIST OF TABLES

Table Number Page

Table 4.1: Temperature Update Schedule ... 16
Table 5.1. Benefits of VPR Placement with Frequent Static Timing Analysis for Conventional MCNC Netlists (Default λ, Default Criticality Exponent) ... 27
Table 5.2: VPR Placement λ and Criticality Exponent Tuning for Conventional MCNC Netlists ... 30
Table 5.3. Instability of VPR Placement with Frequent Static Timing Analysis for Depth = 1 MCNC Netlists (Default λ, Default Criticality Exponent) ... 35
Table 5.4: VPR Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists, Phase 1 ... 39
Table 5.5: VPR Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists, Phase 2 ... 40
Table 5.6: Incremental Criticality Update Placement λ and Criticality Exponent Tuning for Conventional MCNC Netlists ... 50
Table 5.7: Incremental Criticality Update Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists ... 51
Table 5.8: Conventional MCNC Netlist Placement Comparison ... 53
Table 5.9: Depth = 1 MCNC Netlist Placement Comparison ... 53
Table 6.1: FF-Level Placement Activation Point Exploration (Clustered Architecture) ... 75
Table 6.2: FF-Level Placement Criticality Threshold Exploration (Clustered Architecture) ... 75
Table 6.3: FF-Level Placement Separation Probability Exploration (Clustered Architecture) ... 75
Table 6.4: FF-Level Placement Homing Probability Exploration (Clustered Architecture) ... 75
Table 6.5: FF-Level Placement Results (Clustered Architecture) ... 76
Table 6.6: Retiming Activation Point Exploration (Clustered Architecture) ... 77
Table 6.7: Simultaneous Retiming and Placement Results (Clustered Architecture) ... 77
Table 6.8: Comparison of FF-Level Placement and Retiming Placement (Unclustered Architecture) – Original Sequential MCNC Netlists ... 81
Table 6.9: Comparison of FF-Level Placement and Retiming Placement (Unclustered Architecture) – Depth = 1 MCNC Netlists ... 81
Table 6.10: Effect of Architecture on Leiserson/Saxe Retiming After Placement ... 81
Table 7.1. Normalized Results for Length-16 Long Wire Architecture ... 107
Table 7.2. Normalized Results for Length-8 Long Wire Architecture ... 107
Table 7.3. Normalized Results for Length-4 Long Wire Architecture ... 107
Table 7.4. Normalized Results for Armada, *k*=1, 2, 4 ... 109
Table 7.5. Normalized Results for Armada, *AC*=10, 8, 6, 4, 2 ... 109
Table 7.6: Capped Link Criticality of Connections in Figure 7.13 ... 111
Table 7.7: Effect of Critical Path Delay on Revised Cost Function ... 111
Table 8.1: Estimated Critical Path Delay of Conventional FPGA ... 124

Table 8.2: Estimated Critical Path Delay of Island-Style FPGA with Registered Switchboxes 124
Table 8.3: Architectures Used in Testing Phase I – Adding Independent Flip-Flops and 1 to 16-bit Shift Registers..135
Table 8.4: Architectures Used in Testing Phase II – Adding Independent Flip-Flops and Shift Registers, 1 Modified LUT/CLB 140
Table 8.5: Architectures Used in Testing Phase III – Adding Independent Flip-Flops and Shift Registers, 2 Modified LUTs/CLB..143
Table 8.6: Architectures Used in Testing Phase IV – Adding Independent Flip-Flops and Shift Registers, 4 Modified LUTs/CLB..145

ACKNOWLEDGEMENTS

More than anything having to do with transistors, wires or CAD tools, working on this dissertation has taught me to appreciate people. During my time in school, I have had the good fortune to work for and alongside some truly brilliant people. As I look back, I struggle to really comprehend the countless ways that they have made my life better.

First, I'd like to thank the professors that inspired me to do research while still an undergrad at Northwestern. As my academic advisor, Professor Prith Banerjee encouraged me to push my limits and helped me find my first research position. As an instructor and, later, a research advisor, Professor Majid Sarrafzadeh singlehandedly launched my interest in algorithms and solving hard problems in general. Although mind-bendingly difficult and more boot camp than classroom, his CAD course laid the foundation of my academic career.

Next, I'd like to thank the students in the lab that came before me. It was simply a privilege to work beside Mark Chang and Akshay Sharma. As colleagues, they made life as a grad student interesting and as friends, they make life as a grad student bearable. Along those lines, I'd like to thank all of my friends. I hesitate to single anyone out, but particularly Darrick Lew, Masa Mizuchi and Elizabeth Marsten helped me soldier on when I simply couldn't figure out how to put one foot in front of the other anymore.

Of course, grad school can be tough economically too. I'd like to thank the National Science Foundation for providing the majority of the financial support for my work. Not all students are fortunate to get research assistantships and I feel fortunate to have had the opportunity.

Lastly, I'd like to convey my deep appreciation and respect for my advisor, Professor Scott Hauck. For over a decade he has been a patient mentor, role model, and friend. Frankly, I don't think I could ever thank him enough for his help professionally and, more importantly, personally. Where could I possibly even start? He gave me my first shot at research as an undergrad, somehow managing to gently throw me into the deep end of the pool. Later, he quite literally took me in from the cold and gave me a second chance at grad school. Something that I certainly can never thank him enough for, through several spectacular failures over the years, he never lost confidence in me even when my confidence in myself had long eroded. There is absolutely no doubt in my mind that I would have left graduate school long ago without his guidance and support.

To everyone that I met me along the way - this dissertation is more the product of your help and encouragement than anything else. Thank you.

DEDICATIONS

For Mom and Dad. My blood, sweat and tears are your blood, sweat and tears.

Chapter 1: Introduction

Field Programmable Gate Arrays (FPGAs) are programmable semiconductor devices that can provide high performance computing with low engineering effort for a large variety of applications. This has proved to be a powerful combination, and FPGAs have grown into a multi-billion dollar market in the two decades since their introduction. FPGAs offer this fast and easy-to-use computation by providing a large array of relatively small programmable logic elements that can be connected to each other through a flexible communication network to form more complex calculations. Although there are some notable exceptions, the majority of FPGAs use SRAM to configure both the logical elements and the interconnect structure. This not only allows them to implement arbitrary computation, but also gives them the capability to be programmed and re-programmed to perform multiple different functions.

For many applications, the large programmable computational fabric FPGAs offer provides multiple advantages over both conventional microprocessors and *Application-Specific Integrated Circuits* (ASICs). Unfortunately, although the programmable nature of FPGAs represents the greatest advantage they hold over other technologies, it also contributes to one of the most serious disadvantages: a much lower achievable clock frequency. FPGA application developers often try to reduce the impact of this inherent performance overhead by breaking their computations into smaller, faster sections using registers. One issue this can cause is that adding registers to an application can fundamentally change its characteristics and the demands it places on the underlying system. As designers demand higher and higher throughput from their FPGA-based applications, the number of registers in their circuits will also rise. This further compounds the issues that these types of circuits can present. This proliferation of heavily registered applications raises concerns for at least two areas of FPGA research, the primary topics of this dissertation.

First, FPGA application developers rely on a large range of sophisticated *Computer Aided Design* (CAD) tools to map their computations to a physical device. The effectiveness of these development tools is extremely important to produce efficient, high performance implementations. However, circuits with a large number of registers present multiple problems that existing CAD algorithms do not address. These issues can cause poor circuit performance, instability within specific compilation tools, or even instability in the entire toolflow. Dealing with these concerns can allow developers to obtain much better results.

Second, from the perspective of the architectures themselves, adding registers to a circuit puts a larger burden on the flip-flop resources provided by the device. Increasing the number and accessibility of the registering resources can drastically improve an FPGA's support for heavily registered applications. Although there is a large body of academic work looking into improving these attributes, many of the proposed systems create serious problems for applications that do not have a large amount of registers. This makes general-purpose computing very difficult on these specialized devices and prevents them from benefiting from the same technology scaling and economies of scale that have been an essential part of the success of mainstream FPGAs.

This dissertation discusses the nature of heavily registered applications, introduces CAD tools to handle them, and lays the groundwork for a new generation of high-performance FPGA system. It is organized as follows:

- **Chapter 2: Field Programmable Gate Arrays** provides background on classic FPGA architectures and discusses some inherent design tradeoffs.
- **Chapter 3: Pipelining, Retiming and C-Slowing** describes how registers can be introduced into an application to improve circuit speed.
- **Chapter 4: FPGA Development Tools** offers details of traditional FPGA physical design techniques and discusses some of the problems that heavily registered circuits can pose.
- **Chapter 5: Enhanced Timing-Driven Placement** discusses a fundamental limitation of existing timing-driven placement algorithms and offers a new technique that dramatically improves critical path delay.
- **Chapter 6: Register-Aware Placement** illustrates some of the difficulties that the conventional toolflow encounters with heavily-registered circuits and presents two new techniques to improve performance.
- **Chapter 7: Register-Aware Routing** describes existing register-centric routing algorithms and presents a new timing-driven approach.
- **Chapter 8: Register-Enhanced Architectures** concentrates on the potential register resource limitations of conventional FPGA architectures and discusses several prospective improvements.
- **Chapter 9: Conclusions and Future Research** summarizes the contributions of this dissertation and suggests some potential topics that warrant further investigation.

Chapter 2: Field Programmable Gate Arrays

Although FPGAs have evolved considerably over the last two decades, the fundamental benefits, tradeoffs, and characteristics of the hardware remains largely the same. FPGAs offer a large sea of programmable logic blocks embedded in a flexible communication network and their unique computational fabric can offer multiple advantages over competing technologies. This chapter will outline the architectural components of a modern FPGA and compare FPGAs to other computational systems.

2.1: Conventional FPGA Architectures

The most popular FPGA arrangement today is the *island-style* architecture. As seen in Figure 2.1, it is named for the characteristic that its computational resources are divided into small islands of logic blocks that are surrounded by a sea of interconnect wires and programmable communication resources.

Each logic block can generally implement any function of *N* inputs through the use of Look-Up Tables (LUTs). LUTs are simply small memories that use the inputs of the logic block to address a read-only memory. By filling the contents of a LUT with different values when configuring the device, the user can

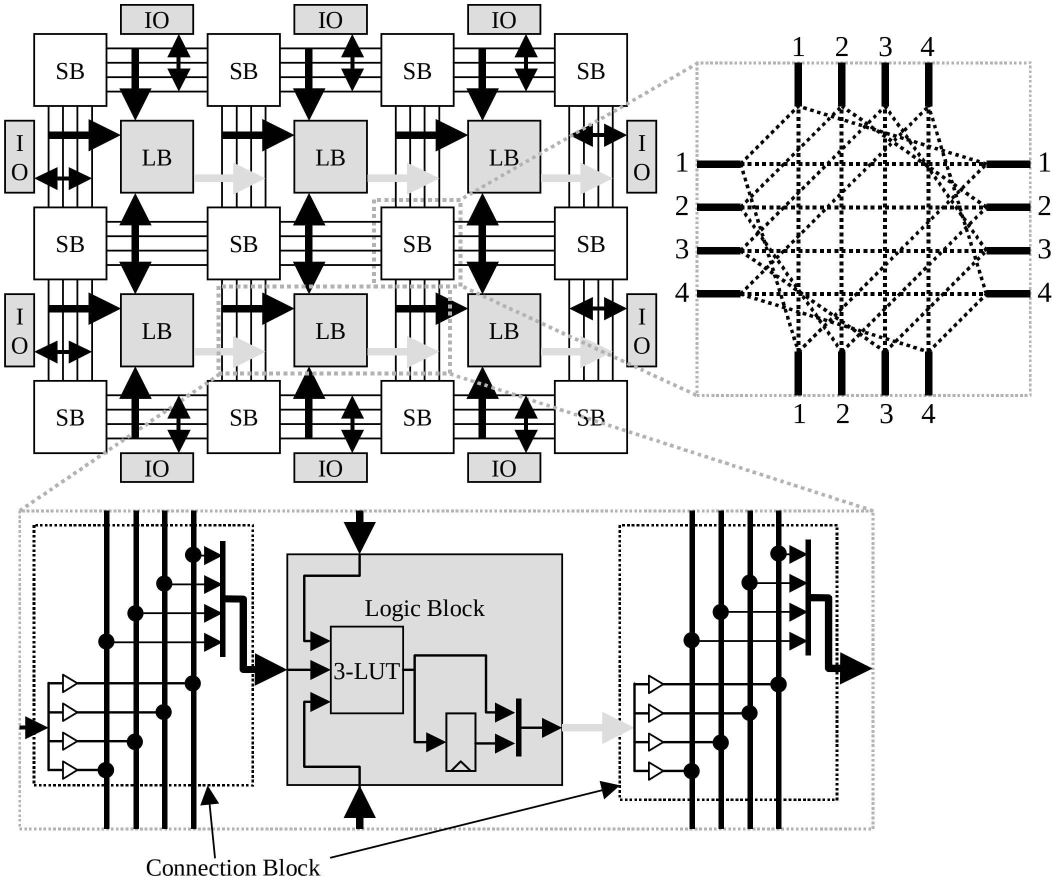


Figure 2.1: Conventional Island-Style FPGA

change the behavior to calculate any arbitrary function. FPGAs also offer the opportunity to implement sequential logic by providing an optional flip-flop on the output of its LUTs. This LUT/flip-flop pair is sometimes referred to as a *basic logic element* (BLE). To create a denser computation fabric modern FPGAs often cluster multiple BLEs into a single logic block.

The communication resources provided by island-style FPGAs can be separated into three main components: *channels, connection blocks* and *switchboxes*. Channels are simply groups of individual wires logically organized into bundles by their physical location. The architecture shown in Figure 2.1 has channels of width four since four independent wires surround each logic block. Connection blocks manage the movement of data in and out of channels by controlling which wires within a channel receive a logic block output or primary input, and which wires drive a logic block input or primary output. Switchboxes are responsible for connecting wires in different channels together. Although there are many different types of switchbox, Figure 2.1 shows one possibility. Here, each wire in a channel has the capability of connecting directly across the switchbox to make longer connections in the same direction or turning 90 degrees left and right. Similar to logic block configuration, connection blocks and switchboxes are built from programmable elements that make arbitrary communication possible.

The most sophisticated FPGAs today often also include specialized communication, logic, and memory features. In addition to the single-length interconnect wires shown in Figure 2.1, they generally have longer segments that span multiple logic blocks, even up to the entire chip's length. Although less flexible than unit-length wires, longer segments improve the speed of long distance communication since signals that use these resources need to traverse fewer programmable switch points. FPGAs may also include dedicated *carry chains*. These are specific logic and directional connections between blocks in the same row or column that can improve the speed of wide additions. These dedicated connections supplement the generic communication network and are considerably faster than sending signals out on to shared interconnect channels. By a similar token, FPGAs that offer multiple BLEs clustered within a single logic block often have an internal interconnect system within each logic block that allows BLEs to be cascaded together without using external wires.

The logic structures themselves also often have some specialized resources or unique operating modes. Modern Xilinx devices, for example, have the capability of exposing the memory bits within their LUTs so that they can be used as very small RAMs or shift registers [45]. FPGA architectures might also replace some of the LUT-based logic blocks altogether. Dedicated coarse-grain functional units such as large memories, fast multipliers or even simple microprocessors are common. These *hard cores* supplement the generic logic fabric by implementing functions that are very slow or expensive to implement using LUTs.

Although not all applications might make use of these sophisticated resources, they are used commonly enough that commercial FPGA companies often include this type of specialized feature to improve the performance of their devices for the bulk of customer applications. As will be discussed in more detail in Chapter 8, maintaining the general-purpose performance and efficiency of FPGAs is critical. This means that while changes can be made to an architecture, any modifications must have one of two characteristics. If the change is costly in terms of silicon area, as in the case of embedded multipliers or microprocessors, it must add a great deal of functionality or boost performance dramatically for a large number of end users. Alternatively, if a given architectural change is only useful for some applications, it must minimally affect the area and performance of the FPGA for applications that cannot use the new feature. The implications of this fundamental design decision is central to the discussion in Chapter 8.

2.2: FPGAs, Microprocessors and Application-Specific Integrated Circuits

The programmable computation fabric that FPGAs offer give them some clear advantages over both microprocessors and ASICs for many applications. Compared to traditional general-purpose microprocessors, FPGAs provide two capabilities: the ability to implement customized computation and the ability to execute many calculations in parallel.

Although both software running on a microprocessor and a circuit implemented on an FPGA allow a user to perform arbitrary computation, the degree of flexibility between the two platforms differs considerably. A program written for a microprocessor must be compiled down to a fixed set of instructions dictated by the processor's *instruction set*. On the other hand, the developer of an application on an FPGA has the capability to generate specialized pieces. For example, if a particular application does not use any floating-point computation, the transistors devoted to a floating-point unit on a modern processor will sit idle. However, since the instructions that a processor provides are fixed and many applications use floating point extensively, the processor must have dedicated hardware to support this. Conversely, an FPGA can be configured to implement one specific application, so all of the available resources can be devoted to the task at hand. Similarly, the individual instructions that a microprocessor supports are largely determined by legacy compatibility and what "anticipated" programs require. Thus, while common operations such as simple addition and multiplication will be implemented in the instruction set directly, less common operations will need to be broken down into a series of instructions that the processor does support. A good example of this is the bit-wise operations popular in encryption algorithms. Although very simple transformations, these functions require multiple instructions to accomplish on a modern processor. On the other hand, FPGAs have the capability to implement custom computations and can directly implement any necessary operations.

Furthermore, the performance of microprocessors is limited on many applications by their *sequential execution model*. Classically, instructions are fetched and executed one at a time, and even modern superscalar processors only have the capability of executing a small handful of instructions simultaneously. FPGAs, however, have the capability to exploit massive parallelism. For example, if a user has a list of N numbers to sum together, a microprocessor will fetch each one individually and keep a running tally. This will require on the order of N clock cycles to complete because the sequential execution model of the processor limits the amount of parallelism the system can implement. Conversely, the parallelism that can be exploited on an FPGA is only limited by the size of the device. If an adder tree with N leaf nodes can fit on a given FPGA, the computation can be performed in $\log N$ time.

FPGAs are also often used as an alternative to Application-Specific Integrated Circuits. As the name suggests, ASICs are custom-fabricated chips designed to perform a specific computation extremely quickly. Since they are specialized hardware devices, like FPGAs they are able to avoid the overhead and limited parallelism of microprocessor-based implementations. However, unlike FPGAs, they are not programmable circuits and generally cannot be repurposed for any other application. Since each new design must be developed and manufactured independently, new devices present an extremely high economic and intellectual hurdle. Not only must a design go through months of development and verification before fabrication can begin, even highly related devices will have completely unique sets of fabrication masks, packaging concerns and testing requirements.

FPGAs have a distinct advantage over ASICs because one chip can be used to produce a wide range of different applications. Once a single FPGA has been designed, manufactured and tested, applications mapped to that chip can be developed and debugged at a much more intuitive functional block level. Combined with the fact that fabrication and packaging costs are divided among all the designs that use that platform, using FPGAs results in a much faster time-to-market and smaller engineering cost. Companies such as Xilinx and Altera specialize in producing commodity FPGAs that provide a versatile and inexpensive pathway to producing hardware-based applications.

Although all of these factors seem to indicate that FPGAs are inherently superior to both microprocessors and ASICs, this heavily depends on the desired application. First, computations that do not benefit from custom operators or do not have inherent parallelism are generally far more efficiently implemented on conventional microprocessors. Obviously, if a computation cannot exploit any of the advantages that an FPGA has over a microprocessor, these devices become far less attractive. Furthermore, FPGAs are only economically advantageous compared to ASICs if the desired volume of chips is relatively low. In high volume, the initial engineering costs become less important since they are amortized over so many chips.

On the other hand, the overhead presented by the flexibility of an FPGA creates a larger overall die, increasing the per-unit manufacturing cost.

Furthermore, the universal nature of the logic elements, combined with the flexibility built into the communication network, means that all of the netlists mapped to an FPGA are merely “emulated” on the hardware and running through a level of indirection. For example, if we would like to add two numbers together on either a microprocessor or an ASIC, the physical adder that this computation is executed on can be built from dedicated transistors communicating via directly connected wires. This means that the entire operation can be carefully designed and optimized specifically for high performance. Conversely, an addition performed on an FPGA must be built from much more generic logical pieces that are connected through much slower, shared communication channels. Thus, although the underlying hardware is capable of performing a wider range of different functions, this flexibility limits the operational efficiency. The next chapter introduces some techniques that application developers can apply to their circuits to minimize the impact of this intrinsic performance penalty.

Chapter 3: Pipelining, Retiming and C-Slowing

Despite the potential economic and engineering advantages FPGAs hold, the generic programmable logic and interconnect offered by an FPGA can be far less efficient at implementing a specific computation than specialized, finely tuned wires and transistors. As discussed in [19], it can be expected that an application mapped to an FPGA will lag an ASIC counterpart by up to 40 times in terms of silicon area, 4.3 times in terms of critical path delay and 12 times in terms of dynamic power consumption. Although minimizing area and power consumption is certainly important, the technical specifications of many applications dictate a required throughput. That is, for the device to function correctly it must reach a specified data rate. Thus, this chapter will focus on three techniques that application developers can apply to a circuit that can improve the operational frequency: *pipelining*, *retiming* and *C-slowing*.

Pipelining is a very simple technique in which a datapath is separated into multiple stages. As shown in Figure 3.1, by breaking a function into smaller pieces we can decrease the longest path in the circuit. However, this increases the *latency*, or number of clock cycles between when data enters the circuit and when completed results are seen on the output. Although this increased latency makes it unsuitable for applications that are sensitive to this, the additional latency can often be offset with a higher clock rate if the computation is split into relatively equal parts.

For example, disregarding the interconnect delay for a moment and assuming an adder to have a delay of 10 units and the setup times of a register to be 1 unit, the unpipelined circuit on the left of Figure 3.1 will have a delay of 20 units and a latency of one clock cycle. The pipelined circuit on the right, however, will have a delay of 11 units and a latency of two clock cycles. Thus, although the pipelined circuit only requires 2 extra units of time to complete the first result (20 units of delay vs. 2 x 11 = 22), it will produce new results nearly twice as fast as the unpipelined circuit (20 units of delay versus 11).

However, to achieve this large performance benefit with small additional latency, pipelining requires that the individual stages be relatively balanced in terms of delay. Considering either of the pipelined circuits in Figure 3.2, for example, both have increased the latency of the netlist (21 units) without decreasing the critical path delay (still 20 or 21 units). This is where retiming can be applied.

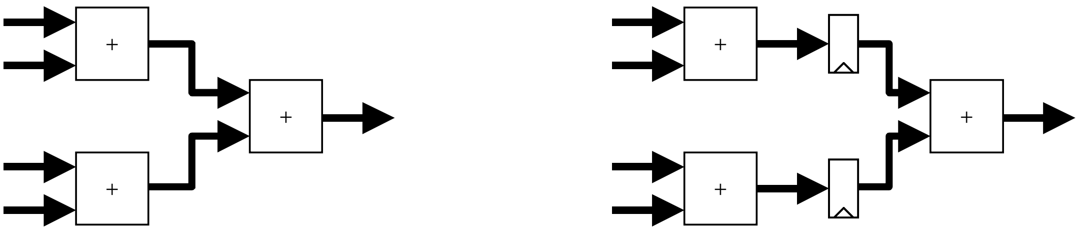


Figure 3.1: Clock Frequency and Latency Effects of Pipelining

Retiming is a technique used in conjunction with pipelining in which registers can be “pushed” or “pulled” through computational blocks to better balance the delay of different stages. First discussed in [21], this relies on the concept that registers can generally be migrated either from each of a block’s inputs to the block’s output or from a block’s output to each of the block’s inputs without changing the logical operation performed by the circuit. The circuit on the left of Figure 3.2 can be transformed into the optimal pipelined circuit in Figure 3.1 by simply combining the two registers on each of the adders’ inputs to a single register on the output. Similarly, the circuit on the right of Figure 3.2 can be improved by replacing the register on the adder’s output with a registers on each of the adder’s inputs.

The most famous method to implement retiming is the Leiserson/Saxe approach [22]. While the authors of this paper actually discuss multiple different formulations of their technique, all of them are iterative processes that operate on a netlist, given a specific target critical path delay. These Leiserson/Saxe techniques gradually push registers around the circuit and can determine whether or not the system can be retimed to reach the target delay given the current amount of registering in the system. By performing a binary search on the target critical path delay, a user can reach the provably maximum clock frequency for a given input netlist.

While retiming can help balance delay across multiple clock cycles, this is not to say that retiming can overcome all limitations. First, not all pipelined applications can be retimed because debugging, testing and proper initialization of the circuit can become much more difficult after retiming is performed. In addition, there is also a theoretical limit imposed by circuits with feedback. For example, consider a circuit that has a feedback loop, as in Figure 3.3. Although registers can be migrated forwards (Figure 3.3a and Figure 3.3b) or backwards (Figure 3.3c and Figure 3.3d) through the chain of adders, the number of registers on the loop itself cannot be changed. This limits the achievable clock frequency to at least four adder delays. The original authors of the work done on retiming [22] discussed the limitation of not being able to increase the number of registers on a loop and suggested an alternative: C-slowing. C-slowing adds additional registers onto feedback loops by duplicating all registers in the netlist *C* times. This increases the retiming capability by interleaving *C* completely separate computations. The original netlist in

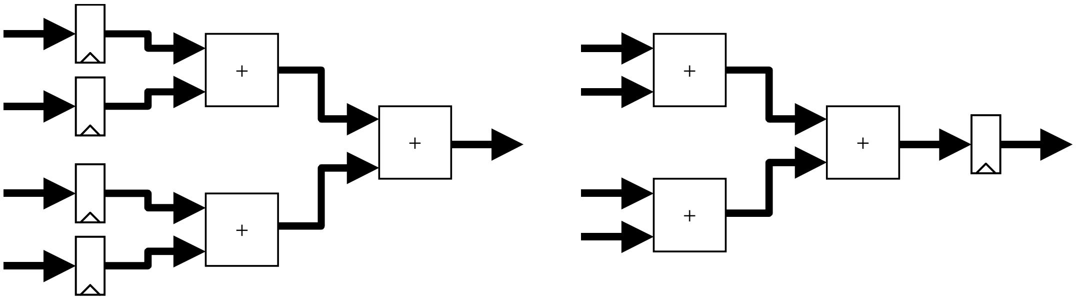


Figure 3.2: Two Examples of Unbalanced Delay Between Pipelining Registers

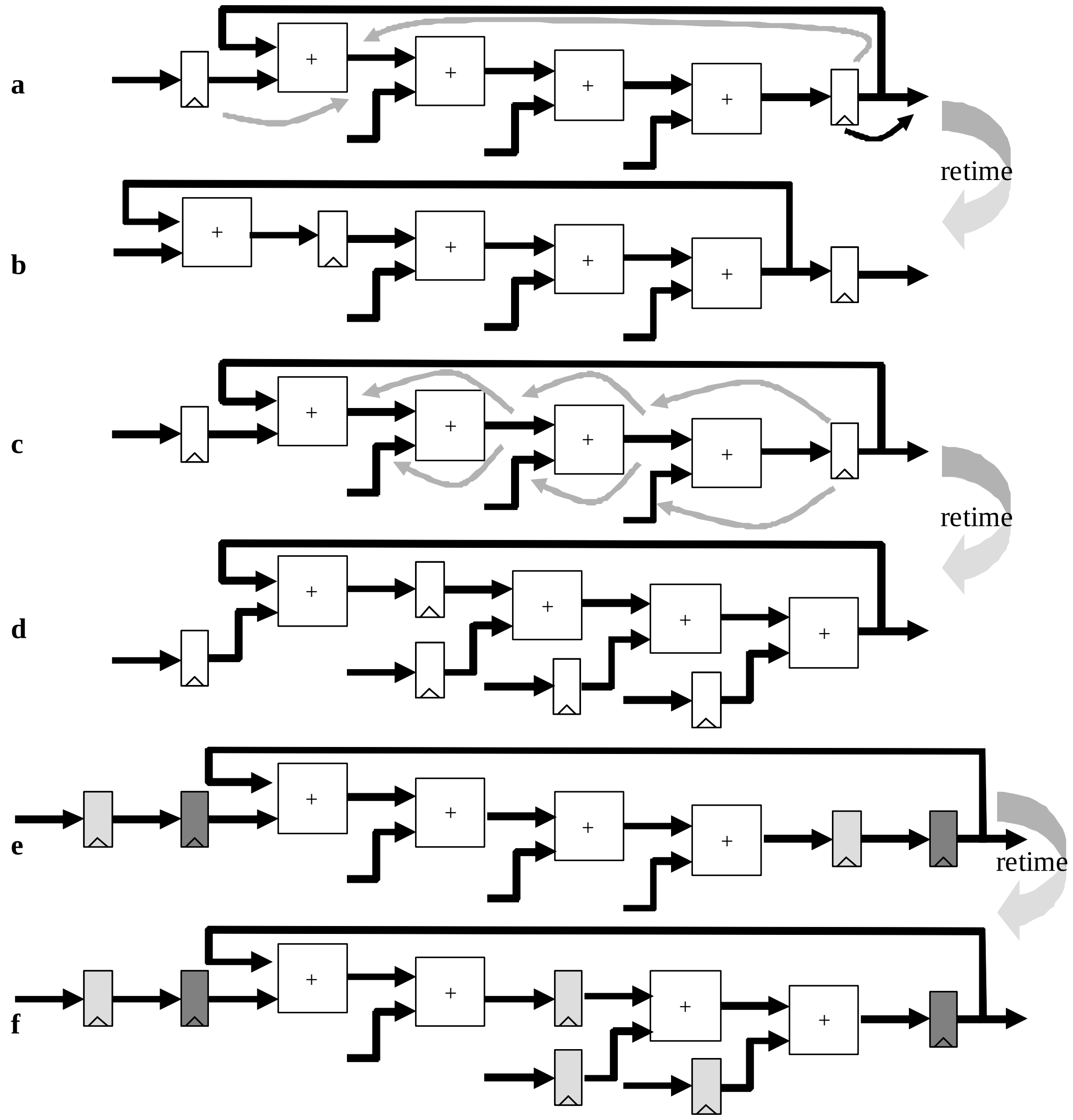


Figure 3.3: Limitations of Retiming and Demonstration of C-Slowing

Figure 3.3a or Figure 3.3c can be “2-slowed” to produce the netlist in Figure 3.3e. As seen in retimed netlist in Figure 3.3f, this approximately doubles the achievable clock frequency. Unfortunately, C-slowing can have limited use since it interleaves multiple independent, partially completed calculations. Thus, the nature of the application itself and its I/O protocol must be amenable to this kind of parallelization. Computations that require iterative computation on a single set of data may not be able to take advantage of C-slowing.

Despite the restrictions associated with pipelining, retiming and C-slowing, developers often utilize these techniques whenever the netlists and application specifications allow. However, determining how to best apply these techniques given a specific circuit and a target architecture can be challenging. Although these concepts will be discussed in far more detail in the following chapters, these issues can be grouped into two basic types of problems.

First, the pipelining, retiming and C-slowing discussed up to this point has only considered the delay though the logic portion of a circuit. However, the delay accumulated in the communication network is a significant part in the overall delay of a system and the distribution of this interconnect delay can vary greatly from one net to another once it has been mapped to a physical architecture. Thus, the manner in which registers could be best distributed is highly dependant upon the arrangement of the rest of the system, but that can be difficult for application mapping tools to evaluate faithfully. This problem is discussed further in Chapters 4 – 7.

Second, as shown in Figure 3.3d and Figure 3.3f, pipelining, retiming and C-slowing can dramatically increase the amount of registers in a circuit. However, the number and availability of physical flip-flop locations offered by the classical FPGAs discussed in Chapter 2 is relatively limited. This is largely because FPGA applications have traditionally not required a large number of registers. That said, for the reasons outlined earlier, future applications will likely require a growing number of registers. Chapter 8 discusses several ways of efficiently increasing architectural support for heavily registered applications.

Chapter 4: FPGA Development Tools

Just as the quality of a software compiler plays a major role in determining the speed of code running on a microprocessor, FPGA CAD tools fundamentally affect the achievable performance of an application mapped to a reconfigurable fabric. This chapter will provide details regarding the traditional FPGA CAD toolflow and discuss some of the issues that heavily pipelined, retimed and C-slowed applications can present.

4.1: FPGA CAD Toolflow

The logic and communication resources that FPGAs offer obviously pose a different problem to both developers and development tools compared to programming for conventional microprocessors. Even so, despite significant differences in the underlying framework, the process of creating applications for modern FPGAs can be thought of much like developing software for a microprocessor. Applications generally begin with a *Hardware Description Language* (HDL) specification. Much like C or C++, this is a largely platform independent representation of the application that must be compiled to a specific FPGA. Compilation for an FPGA consists of five primary steps: *logic synthesis*, *technology mapping*, *packing*, *placement*, and *routing*.

Logic synthesis takes the high-level constructs in the HDL code and turns them into a netlist of basic gates such as NANDs, NORs and flip-flops. The technology mapping phase uses this generic gate representation and determines how these pieces could be efficiently translated to the hardware given the specific LUTs and fixed resources offered by the target FPGA. The packing tool then takes these mapped pieces and attempts to merge LUTs and flip-flops into groups of logic blocks. The placement tool then determines the physical location of each logic block in this packed netlist so as to minimize the amount of communication required. Finally, routing determines how the blocks in the placed netlist communicate with each other by assigning signals to specific wires. This routed netlist can then be turned into a configuration bitstream to program the FPGA.

While logic synthesis and technology mapping are essential parts of a modern FPGA compiler, this dissertation primarily focuses on the effect netlist and architectural characteristics have on packing, placement and routing. Thus, the discussion here will feature background on these three physical design phases.

4.2: Packing

The most popular academic FPGA packing tool today is VPack [26]. VPack uses a two-step approach in which flip-flops are first mated with appropriate LUTs to map to the fewest BLEs, and then these BLEs are

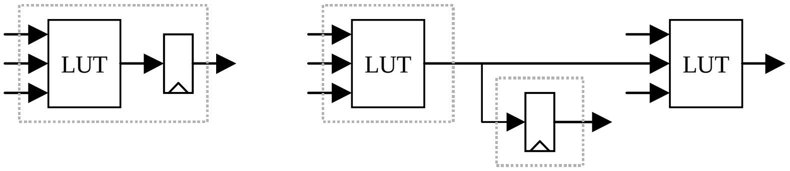


Figure 4.1: Packing Restrictions [2]

combined to form logic blocks if the architecture implements clustered logic. The first stage examines the way that each flip-flop is used to determine how to pack LUTs and flip-flops together. Some architectures may restrict the output of a BLE. In the case shown in Figure 2.1, a LUT/flip-flop pair has the capability to output either the raw LUT output or the registered LUT output, but not both. Thus, as seen on the left of Figure 4.1, if the rest of the netlist only uses the registered output of a LUT, the optional flip-flop attached to the host LUT can be used and they can be mapped to a single BLE. However, as seen on the right of Figure 4.1, if both the gated and non-gated output is needed the LUT and flip-flop must be mapped to separate BLEs.

The second portion of the packing process attempts to combine BLEs into the fewest number of clustered logic blocks, subject to the limitations of the architecture. Although the architecture might have multiple LUTs grouped within a single CLB, some FPGAs attempt to reduce the hardware needed to implement the connection blocks by offering fewer independent inputs than the maximum number that could be required by the cluster. For example, if an architecture is built from clusters of four 4-input BLEs, each logic block might only have twelve, not sixteen, inputs. FPGA architects do this because they realize that logic blocks do not necessarily require independent inputs for all BLEs. Multiple BLEs within a logic block may share common inputs, BLEs may be cascaded together and use communication resources internal to the logic block, or the function mapped to a LUT may use fewer than the maximum number of inputs.

VPack iteratively clusters BLEs with one of two techniques. It first simply selects an unassigned BLE to seed a cluster. Other BLEs are then added to the cluster to completely fill the logic block. Potential cluster-mates are ranked based on their "attraction" to the current cluster – how many inputs and outputs they share. VPack iteratively gathers BLEs with the highest attraction to the current cluster until the CLB is full. Occasionally, though, a cluster may run out of independent inputs before all BLEs are occupied. These situations are forwarded to a second technique. Here, clustering is repeated, but BLEs are added to the cluster based on minimizing the number of inputs.

VPack has also been extended with a timing-driven formulation, T-VPack. This tool is very similar to VPack, but attempts to consider critical path timing during the clustering process. Although it cannot necessarily estimate the delay encountered in the interconnect, T-VPack evaluates how likely it is that each BLE lies on the netlist's critical path based upon the maximum number of consecutive LUTs, or the logical depth, of the logic using the BLE. BLEs along paths that have multiple consecutive LUTs without registering are given special priority. Since communication between BLEs in the same cluster is generally very fast compared to utilizing external routing resources, the tool adjusts its attraction scheme to prefer BLEs that are more likely to be timing sensitive.

Packing is also very useful on architectures that do not limit the input or output connectivity of the LUTs and flip-flops within their CLBs. This is because combining multiple LUTs and registers into a single atomic unit via packing decreases the number of movable blocks. In turn, this dramatically simplifies the following placement process. For example, take a very small netlist consisting of 20 4-LUTs. If a placement tool is attempting to map this netlist to the minimum-sized architecture that consists of five 4-LUT CLBs, there are roughly 3.6×10^{12} different possible placements[1]. Obviously, searching such a large solution space is extremely difficult. However, if the LUTs in the netlist are first packed into five groups of four 4-LUTs, there are only 120 different possible placements[2]. Of course, this simplification of the placement problem means that the vast majority of the potential possible placements are never examined. While that is true, packing is a natural step for most netlists because the placement problem specifically tries to put interconnected blocks as close together as possible. Since the packing tool groups tightly coupled LUTs and registers together into the same CLB, it is likely that the placement tool will still be able to approach the optimal arrangement.

4.3: Placement

The most common algorithm used for FPGA placement is *simulated annealing*. The basic premise of simulated annealing likens the process of determining physical locations for all the logic blocks in a netlist to nature finding a low-energy atomic arrangement for the atoms in a crystal. The authors of [17] recount basic metallurgy: if an iron bar is thoroughly heated, then quickly cooled in water, the result is very brittle and prone to cracking. This is because the small, high-energy crystals that make up the bar contain large amounts of internal strain. A quick cooling process forces atoms into whatever arrangements they can manage before they freeze. However, if the metal is allowed to cool slowly in air, the result is much more

---

[1] This calculation assumes that the individual LUTs within each CLB of the array are functionally equivalent. Thus, there are 5 possible different CLB locations for the first LUT to go into. Since one LUT does not fill the first CLB location to capacity, there are still 5 possible CLB locations for the second LUT, etc. This makes the number of possible solutions ($5^{16}*4*3*2 \approx 3.6 \times 10^{12}$)

[2] There are 5 possible CLB locations in which to map the first packed CLB, 4 possible CLB locations to map the second packed CLB, etc. This makes the number of possible solutions ($5! = 120$).

flexible and resilient. This is because the slow cooling process allows the atoms to move around freely and arrange themselves into large, low-energy state crystals.

This phenomenon is mirrored in logic block placement in several ways. First, the random motion available to atoms while the metal hot is paralleled by iterative random swaps between logic blocks. Next, the energy state of an atomic arrangement is represented by a cost function that can determine the quality of a given placement. The most basic cost function used in FPGA CAD is the total rectilinear, or *Manhattan*, distance between connected logic blocks. Finally, a temperature is associated with each iteration of the process that allows the system to gradually move towards better and better solutions.

Placement begins with an arbitrary initial placement and a very high system temperature. Optimization is achieved by conditionally accepting or rejecting moves while slowly decreasing the temperature of the system. Swaps that provide a better placement are always allowed, while movements that provide a worse placement are probabilistically allowed depending upon the current system temperature and how much worse the movement would make the placement as a whole. In [3], the authors suggest that “bad” movements should be accepted with a probability shown in Equation 4.1.

$$\text{random number}\,[0,1] < e^{-\text{deltaCost}\,/\,\text{Temperature}} \tag{4.1}$$

Since the probability of accepting a move for the worse is directly related to the temperature and inversely related to the change in quality, we are likely to accept virtually all moves early in the annealing process and gradually tend towards only accepting changes for the better as placement continues. While performing changes that make the placement worse seems counter-productive, only accepting good moves is similar to the quenching of metal, which results in local minima and poor placements. As it turns out, permitting solutions that temporarily make the system worse actually encourages better overall placements. This is because the placement tool often needs to transition through “bad” solutions in order to make larger-scale improvements.

The fact that the probability of accepting a move for the worse is dependant upon temperature makes controlling the rate at which the system cools very important. In addition, determining the initial temperature and the total number of moves attempted is also critical. The work in [2] suggests a sophisticated scheme in which these factors are somewhat tied together. First, the initial temperature is determined by performing *N* random moves on the initial placement, where *N* is approximately 100 times the number of blocks in the incoming netlist. Since the initial placement is already arbitrary, these swaps are unconditionally accepted. However, the costs of these moves are recorded and the initial temperature of the annealing is set to 20 times the standard deviation. This insures that virtually all moves are accepted at the beginning of the annealing.

The subsequent placement is divided into *temperature iterations*. During each iteration, the number of moves attempted is based upon the size of the incoming netlist as calculated in Equation 4.2.

$$\text{Moves Per Temperature Iteration} = 10 * (\mathit{NumberBlocks})^{4/3} \tag{4.2}$$

At the end of each temperature iteration a new system temperature is calculated based upon the number of moves accepted during the previous iteration. This is shown in Equation 4.3 and Table 4.1.

$$\text{New Temperature} = \gamma * \text{Old Temperature} \tag{4.3}$$

After the system temperature is updated, the termination condition shown in Equation 4.4 is evaluated.

$$\text{Temperature} < 0.005 * \frac{\mathit{TotalCost}}{\mathit{NumberNets}} \tag{4.4}$$

This adaptive temperature schedule allows the annealer to operate for a short period of time at a high temperature to facilitate large-scale changes to the placement, and spend the bulk of its operation performing medium-scale improvements and small-scale refinements.

This type of relationship is often further reinforced with the addition of *movement windowing*. First suggested in [20], the annealing begins by allowing any logic block to swap with any other logic block in the array. However, as placement continues, it slowly decreases the range that a logic block can move in a single swap by only attempting to change places with a location within an imaginary frame surrounding that block. This window slowly shrinks over time until we only allow nearest-neighbors to exchange places. This can be seen in Figure 4.2.

This enhancement is particularly effective because it encourages the system to continue optimization through a larger portion of the annealing. Late in the annealing process we have largely determined most

Table 4.1: Temperature Update Schedule

Acceptance Rate > 0.96	$\gamma = 0.5$
0.8 < Acceptance Rate ≤ 0.96	$\gamma = 0.9$
0.15 < Acceptance Rate ≤ 0.8	$\gamma = 0.95$
Acceptance Rate ≤ 0.15	$\gamma = 0.8$

of the placement. Therefore, long distance moves are not liable to be accepted because they are unlikely to improve the placement but rather disturb the arrangement we have already carefully set up. On the other hand, shorter distance moves are both far more likely to improve the placement and, if they are a change for the worse, any degradation will also naturally be smaller. Thus, windowing prevents the annealing from stagnating during the later stages of the process by guiding the system towards shorter, more incremental changes. [13]

Similar to the cooling rate, the size of this movement window can also be determined in an adaptive manner. The work in [2] suggests updating the window size at the end of each temperature iteration with Equation 4.5. Obviously, this value is subsequently clamped between one and the maximum size of the array.

$$\text{New Window Size} = \text{Old Window Size} * (1 - 0.44 + \text{Acceptance Rate}) \qquad (4.5)$$

Since the interconnect represents such a large portion of the overall delay in FPGA designs, placement also plays a vital role in determining a netlist's critical path. Although discussed in more detail in Chapter 8, the authors of [3] incorporate both Manhattan distance and delay estimation into their simulated annealing cost function. When their placer is initialized, the system first performs a point-to-point routing between all logic blocks in the target architecture. This allows the system to fill a look-up matrix with the delay of the fastest connection between each pair of logic blocks. These values are then used during annealing to estimate the delay and timing criticality of every connection in the netlist for a given placement. This is shown in Equation 4.6.

$$\text{Timing\_Cost}(i, j) = Delay(i, j) * Criticality(i, j)^{Crticality\_Exponent} \qquad (4.6)$$

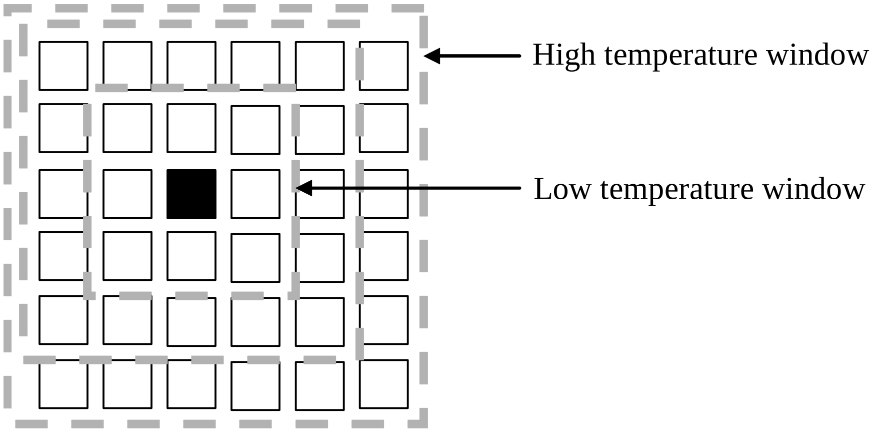


Figure 4.2: Simulated Annealing Windowing

```
Conventional VPR Placement
randomly place logic blocks onto architecture
determine initial temperature
while(!done)
    for I = 0 to numAnnealMovesPerTemp
        select random CLB
        swap CLB with random CLB in move window
        accept or reject move($\Delta$Cost, currTemp)
    end for
    update critical path delay
    update currTemp
    update range limit window
    evaluate exit criteria
end while
```

Figure 4.3: Pseudo-Code for VPR Timing-Driven Placement

In this equation *Timing_Cost(i, j)* represents the cost of the link between blocks *i* and *j*. The slower and the more timing-critical the link, the more expensive delay becomes. On top of this, by increasing the *criticality exponent*, the placer can further emphasize reducing delay on the most critical segments. In a similar manner to the way the movement window is adaptively changed, the criticality exponent is generally set to one at the beginning of the annealing process and slowly increased as the annealing continues.

This timing cost can then be combined with a more traditional Manhattan distance-based cost to evaluate the overall quality of the placement. This will encourage the placement tool to gather the most timing-critical blocks close together at the expense of lengthening less critical connections. Pseudo-code for the entire placement process is shown in Figure 4.3.

4.4: Routing

FPGA routing is generally handled with the *PathFinder* algorithm [28]. PathFinder is an iterative technique that allows signals to negotiate with each other for control over communication resources. The guiding principle behind this approach is that each signal “bids” on the routing resources that it wants. Over time, the “price” of popular resources goes up, encouraging signals that can use less scarce commodities to do so and leave more restricted resources for the signals that truly need them.

PathFinder begins by representing all of the logic and routing resources offered by the target architecture as a *directed graph* of *vertices* and *edges*. Each logic block and wire is converted to a vertex, while the programmable connections offered by the connection blocks and switchboxes are converted into directional edges linking these vertices. The placed netlist is then mapped to this abstract graph. This means that connecting two logic blocks in our netlist is simply a matter of finding a *path*, or series of connected vertices, between the nodes that represent the logic blocks in our graph. Since a given physical wire can only carry a single signal, the challenge PathFinder must solve is to connect all of the signals in our netlist such that no node is *congested*, or allocated to too many nets.

An essential part of PathFinder is *Dijkstra's algorithm* [8]. This is a fast and optimal technique that finds the lowest-cost path between two vertices in a directed graph. Dijkstra's begins by starting a wave of exploration at the source vertex. The neighbors of this node are then added to a list that is sorted by the total cost of the path to these nodes. The source node is marked as "visited" and the router selects a new vertex – the lowest cost node in the list. The unvisited neighbors of this node are then added to the sorted list and the process continues until we find the target vertex or empty the list of routing nodes. PathFinder also uses a slightly enhanced version of Dijkstra's algorithm to find multi-terminal routes by stopping and reinitializing the search each time a sink is found, considering the entire routing tree built thus far as the source.

The PathFinder algorithm uses this basic search while encouraging congestion resolution between different nets. It begins by initializing the cost associated with each vertex to a small *base cost*. All signals in the netlist can then be routed using the approach from above. At this point, PathFinder evaluates the use or *occupancy* of each vertex in the graph. If all of the nets have been connected and no vertices are congested, the routing is valid and the algorithm is complete. However, if any vertices are congested, the cost of these nodes is increase and another routing iteration is attempted. By gradually increasing the cost of overused vertices over time, the use of these nodes is slowly discouraged. This frees them to be used by other paths. The cost of a node during a given iteration is shown in Equation 4.7.

$$c_n = (b_n + h_n) * p_n \tag{4.7}$$

Here, b_n is the base cost of using the node, h_n is a term that reflects the historical congestion of the node, and p_n is a term that reflects the current congestion of the node.

Of course, for most applications it is extremely important to consider critical path timing. The authors of [28] also suggest a timing-driven formulation of PathFinder that uses a slightly modified cost function to improve performance. This allows timing-critical nets to follow fast, but possibly congested paths while encouraging non-critical nets to seek slower, lower congestion alternatives. This is shown in Equation 4.8.

$$C_n = A_{ij} d_n + (1 - A_{ij}) c_n \tag{4.8}$$

Here, A_{ij} represents the criticality of a source/sink pair as found during the last routing iteration, d_n is the delay of a node and c_n is the congestion-based cost function described above. Since A_{ij} falls between zero and one, a route along the critical path of the netlist (A_{ij}=1) only considers the delay of a node without considering its congestion cost. In this way, it will naturally seek the fastest possible path. However, a less

```
Timing-Driven PathFinder Routing
while(!all signals routed || congestion exists)
    for all nets N
        clear N.routing tree
        put source of N into N.routing tree
        sort sinks in decreasing order of criticality (for iteration #1, set all criticalities to 1.0)
        for all sinks of N
            for all nodes in architecture clear visited flag
            put all nodes in routing tree into priority queue PQ at cost C, previous node null
            while(PQ.head not sink[i] of N && PQ not empty)
                remove head of PQ H at cost C, previous node P
                if(H not visited)
                    mark H visited
                    set H.cost to C
                    set previous node of H to P
                    put unvisited neighbors of H into PQ at cost C + neighbor cost + edge cost, previous node H
                end if
            end while
            if(PQ is empty)
                net is unroutable, exit
            else if(PQ.head is sink[i] of N)
                mark sink found
                set previous node of sink to P
                set S to sink
                while (S not in routing tree of N)
                    add S to routing tree
                    set S to S.previous node
                end while
                clear PQ
                update cost of congested nodes
            end if
        end for
    end for
    update critical path delay and sink criticalities
end while
```

Figure 4.4: Pseudo-Code for PathFinder Routing

critical net will consider both delay and congestion. As A_{ij} approaches zero, the congestion cost will play a larger role in determining which path is taken. This formulation encourages less critical nets to find detours so that the most timing-sensitive links can use the fastest, most direct wires. Pseudo-code for the entire timing-driven routing process is shown in Figure 4.4.

4.5: Issues for Heavily-Registered Applications

Pipelining, retiming and C-slowing an application introduces additional registers into the netlist with the hope that this will increase the overall throughput of the system. However, as discussed in Chapter 3, since these new registers also increase the latency of the circuit these registers must be carefully positioned to evenly distribute delay. This makes the effectiveness of timing-driven CAD tools crucial to the system as a whole. However, the addition of a large number of registers into an application can fundamentally change its characteristics and, by extension, the optimization problem it presents to the CAD tools. This potentially creates two unique challenges.

First, a large number of registers in a netlist can confuse existing timing-driven placement and routing algorithms. As will be discussed in Chapter 5 and Chapter 6, this is largely because the relative criticality of different parts of a circuit can change much more quickly in a heavily-registered circuit as the placement and routing is performed. This not only makes the timing information that the tools use to optimize the circuit much more difficult to keep up to date, the algorithms themselves are based upon iterative improvements that subtly rely on the fact that the criticality on individual links does not change very quickly. Thus, when it does change rapidly when attempting to process heavily-registered applications, these algorithms can produce degenerate solutions.

Second, a circuit with a large number of registers that need to be packed and retimed can exacerbate existing problems in the CAD toolflow. As will be discussed in Chapter 7, the traditional compilation process described above is highly compartmentalized and solely feed-forward. In some sense this causes problems already since design decisions that must be made by tools early in the flow, such as logic synthesis and technology mapping, dictate the netlist given to tools later in the flow, such as placement and routing. However, these early portions of the CAD process also have the least amount of information regarding the potential realities of the interconnect delay between logic blocks. Thus, the accuracy of the optimizations performed by these early tools is limited, even though they potentially have the largest impact on the quality of the final result. Packing circuits with a large number of registers can make this problem worse because traditional packing algorithms do not expect multiple registers on a LUT output. Thus, they can produce packed netlists that severely limit the options available to the placer and router. Retiming compounds these issues because it needs to restructure the netlist as it migrates registers through logical elements to balance delay. However, the point in the toolflow in which this is most convenient is prior to packing. Therefore, retiming is generally performed without considering the interconnect delay information only known after placement and routing.

Chapter 5: Enhancing Timing-Driven Placement

As discussed earlier, when pipelining, retiming and C-slowing are aggressively used they can insert a large number of registers into a netlist. However, these registers make the circuit larger and increase the latency of the system, so obviously application developers would like to maximize the potential performance benefits of these additional registers as much as possible. That said, while existing timing-driven placement tools have shown their advantages over purely wirelength-driven formulations [25], relatively little is known about the absolute performance of these types of algorithms. Furthermore, they have generally only been tested on classical, relatively lightly registered circuits.

This chapter will illustrate some potential shortcomings of the most popular timing-driven FPGA placement approach that can lead to instabilities in the simulated annealing placement itself. In addition this chapter will outline some of the different characteristics that heavily registered netlists have that can prevent existing timing-driven placement approaches from attaining the maximum potential of these circuits. This will lead to the introduction of a new technique for timing-driven placement that can significantly improve the performance of both lightly and heavily registered applications.

5.1: Background on VPR Timing-Driven Placement

VPR [3] is one of the most popular academic FPGA place and route tool suites. As the de facto standard, it has served as both a building platform and comparison target for countless other research efforts. VPR includes T-VPlace, a simulated annealing based timing-driven placement algorithm. T-VPlace considers both a net's wirelength and delay contribution during placement to achieve a good balance between overall netlist routability and critical path delay. During simulated annealing, it calculates the cost of a move using Equation 5.1.

$$\Delta \mathrm{C} = \lambda * \frac{\Delta \mathrm{Timing\_Cost}}{\mathrm{Previous\_Timing\_Cost}} + (1-\lambda) * \frac{\Delta Wiring\_Cost}{\mathrm{Previous\_Wiring\_Cost}} \tag{5.1}$$

In this way, VPR can emphasize maximum routability ($\lambda = 0.0$), minimum critical path delay ($\lambda = 1.0$) or, most likely, strike a balance between the two. While the *Wiring_Cost* is essentially just a summation of all nets' bounding boxes, calculating the *Timing_Cost* is a bit more complex.

Before placement on a given architecture is started, VPR builds a *distance vs. delay table* that estimates the shortest path delay between each logic block and I/O pad in the array and every other logic block and I/O pad in the array. VPR then uses this table throughout the annealing process to determine the source/sink delay of each connection in the netlist. This allows VPR to estimate the delay of each connection in the netlist for a given placement. Of course, due to routing congestion this estimate table cannot correctly reflect the real delay of every link of any placement. For example, if the annealing were stopped immediately and the immature placement sent to the router, the actual delay for any given connection as found by the router would likely be much larger than the shortest-path delay estimates used by the placement tool. However, it is generally assumed that the congestion in the final placement will be relatively low and that the most critical signals will be able to take their fastest preferred path during routing. Thus, these delay estimates offer the placement tool a relatively good idea regarding the timing implications of the placement as the annealing progresses.

Calculating the timing cost of the current placement begins by performing a *static timing analysis* on the initial random placement. As seen in Figure 5.1, static timing analysis uses the delay estimates from the distance vs. delay table and steps through the netlist from the inputs to the outputs in order to determine the critical path through the system. As seen in Figure 5.1b, this begins by setting the *arrival time* of all primary inputs and registers to be 0. Then, using the delay estimates of each connection, the maximum arrival time of all nodes is propagated throughout the netlist. This is seen in Figure 5.1c.

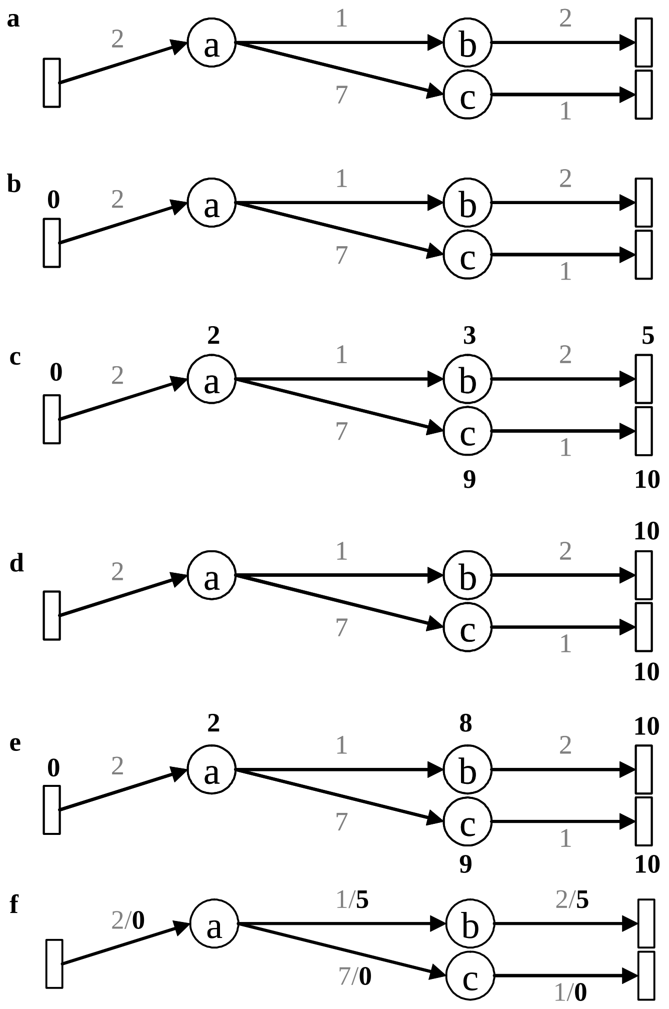
This process calculates D_{max}, the overall maximum critical path delay of the current placement. Based upon this information, the *timing slack* of each source/sink pair can also be calculated. This is performed by determining the *required time* of each node. As shown in Figure 5.1d, this begins by setting the required time of all primary outputs and registers to D_{max}. In a similar manner as before, the minimum required time for each node is propagated through the netlist. This is shown in Figure 5.1e. Finally, the timing slack for each connection can then be calculated. As shown in Figure 5.1f, this is the required time of the sink minus the arrival time of the source minus the delay of the connection itself.

The information from static timing analysis is then incorporated into the timing cost using Equations 5.2 and 5.3. As shown in Equation 5.2, first the relative criticality of each link in the netlist is calculated based upon D_{max} and the timing slack.

$$Criticality(i, j) = 1 - \frac{Slack(i, j)}{D_{\max}} \tag{5.2}$$

$$\text{Timing\_Cost}(i, j) = Delay(i, j) * Criticality(i, j)^{Crit\_Exp} \tag{5.3}$$

As shown in Equation 5.3, VPR then weights the impact of the delay between each source-sink pair based upon its criticality. That is, delay along a path that has lots of timing slack is relatively cheap, while delay anywhere along the critical path is expensive. An exponent is also sometimes included to further discourage high criticality links.



Delay of links from placement

Arrival time of primary inputs & registers set to 0

Propagation of arrival times forwards
AT_i = max(AT of source + link delay)

Required time of primary outputs & registers set to D_{max}

Propagation of required times backwards
RT_i = min(RT of source - link delay)

Slack(i, j) = RT of $sink_j$ - AT of $source_i$ - link delay

Figure 5.1: Static Timing Analysis

Finally, Equation 5.4 shows that the overall placement timing cost is calculated as the summation of the timing cost of each source/sink pair.

$$\text{Timing\_Cost} = \sum \text{Timing\_Cost}(i, j) \tag{5.4}$$

5.2: Implications of Static Timing Analysis

While the intent of VPR’s timing-driven formulation is indeed very important, the realities of practical implementations can interfere with its effectiveness. Focusing on Equations 5.2 and 5.3, VPR’s timing cost function is based upon the source/sink criticalities calculated during static timing analysis. Unfortunately, static timing analysis is far too computationally expensive to perform after each annealing move. Thus, by default VPR only performs a single timing analysis at the beginning of each temperature iteration. It then uses these criticalities to calculate the quality of subsequent moves until the next temperature iteration.

This means that VPR generally performs less than a few hundred timing analysis runs instead of potentially several millions.

Revisiting VPR's basic cost function, this optimization can be captured formally. In Equation 5.5, VPR calculates the criticality of each source/sink pair (*i*, *j*) at the beginning of temperature iteration *k*.

$$Criticality(i, j, k) = 1 - \frac{Slack(i, j, k)}{D_{\max}(k)} \qquad (5.5)$$

For any given placement within the *k*th temperature iteration, Equation 5.6 can be used to calculate the timing cost. This is simply the delay of the source/sink pair (*i*, *j*) at temperature iteration *k*, move number *l* multiplied by the criticality of the link as calculated at the beginning of the temperature iteration.

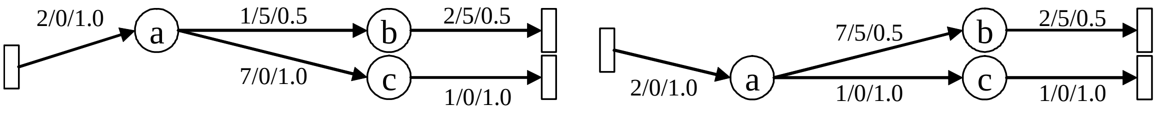
$$\text{Timing\_Cost}(i, j, k, l) = Delay(i, j, k, l) * Criticality(i, j, k)^{Crit\_Exp} \qquad (5.6)$$

This makes the incremental timing cost as shown in Equation 5.7 simply the change in delay between of move (*l*-1) and move *l* multiplied by the criticality of the link at the beginning of the temperature iteration.

$$\Delta\text{TC}(i, j, k, l) = \left[Delay(i, j, k, l) - Delay(i, j, k, l-1)\right] * Criticality(i, j, k)^{Crit\_Exp} \qquad (5.7)$$

Unfortunately, while performing static timing analysis only once per temperature iteration does make placement orders of magnitude faster, since the placement algorithm does not update the criticality nor critical path delay within a temperature iteration, the timing information that the annealer has slowly gets less and less accurate. This can lead to less than satisfying final results. At the beginning of the annealing the placement tool calculates the critical path delay. This value is then used to calculate the slack and criticality of each source/sink pair. The problem occurs because, as the annealing begin to move blocks around, a gap forms between the real criticalities of the current placement and the values used to calculate the timing cost. Since a single temperature iteration might attempt tens of thousands to hundreds of thousands of moves, the optimizations attempted towards the end of a temperature iteration can actually be self-defeating.

Figure 5.2 illustrates this problem. Here, the placer believes that the timing of the system will improve if it moves block *a* to reduce the delay on the critical path (*a*, *c*). However, this particular move accomplishes this by adding delay to the previously non-critical path (*a*, *b*). While this change actually increases the



Timing Cost = (2+7+1)*1.0 + (1+2)*0.5 = 11.5

Timing Cost = (2+1+1)*1.0 + (7+2)*0.5 = 8.5

Figure 5.2: Effect of Stale Criticality Information
Notation: delay / slack / criticality

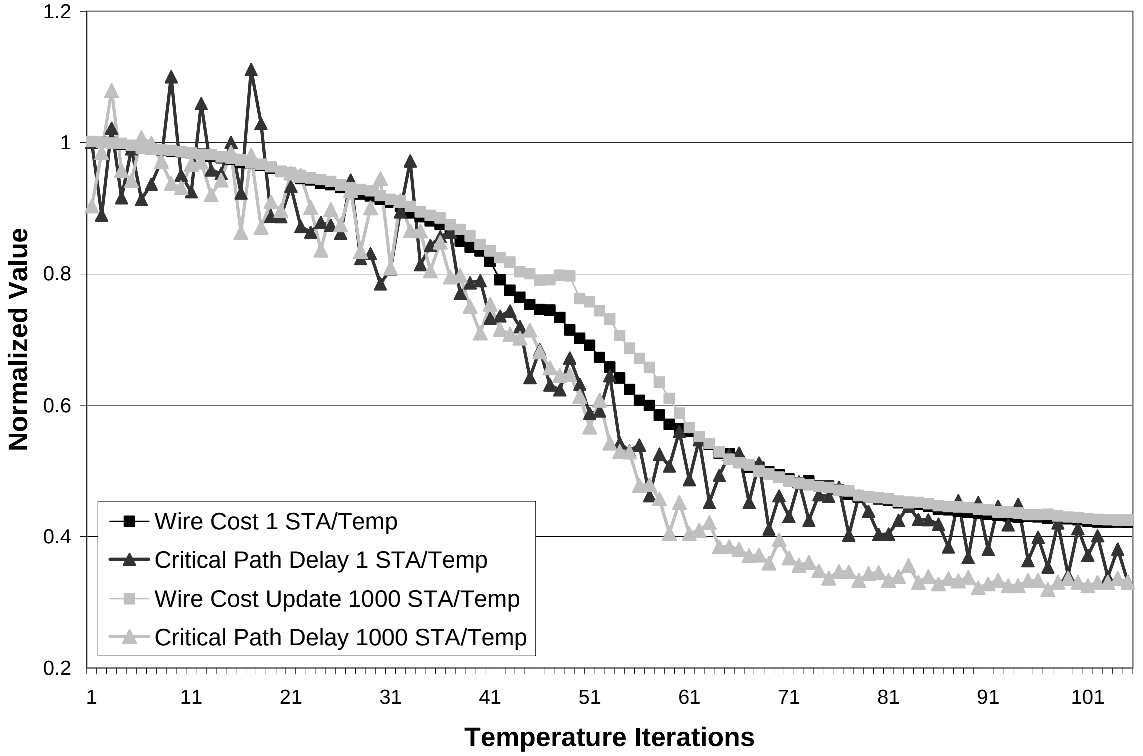


Figure 5.3: VPR Placement with Stale Criticality Information

critical path delay of the circuit from 10 to 11, the placement tool is unaware that this is a poor choice because, following Equation 5.7, the timing cost goes down from 11.5 to 8.5. Unfortunately, this timing cost is inaccurate because it only looks at the changes in delay on connections, without considering the impact that this has on link criticality.

Assuming for the moment that algorithmic runtime can be ignored, the advantages of more up-to-date criticality information is easily demonstrated. Figure 5.3 shows two placement runs of a benchmark included with the VPR toolsuite, *ex5p*. These placement runs were performed on the single 4-LUT, single flip-flop *4lut_sanitized* architecture, also included with VPR. Shown in black is the wirelength and estimated critical path delay calculated at the end of each temperature iteration when one static timing analysis (STA) is performed per temperature iteration. Shown in gray are the results when 1000 static

Table 5.1. Benefits of VPR Placement with Frequent Static Timing Analysis for Conventional MCNC Netlists (Default λ, Default Criticality Exponent)

Static Timing Analysis/Temp	Normalized Wire Cost	Normalized Routed CPD
1	1.000	1.000
10	1.029	0.904
100	**1.030**	**0.857**
1000	1.031	0.864
10000	1.036	0.869

timing analysis runs are performed per temperature iteration. For a point of reference, in the case of *ex5p* this equates to roughly one static timing analysis for every 100 simulated annealing move attempts. Clearly, while the wirelength costs for both placement runs, denoted in squares, are very similar and smoothly decreasing, the critical path delay for the placement performed with the default settings, denoted in black triangles, fluctuates considerably. This is particularly concerning since this oscillation persists even as the placer nears the end of the annealing process. These oscillations represent a 20-30% swing in critical path delay, with no apparent guarantee whether the placement will end with a faster or slower circuit. This oscillation is likely due to the fact that, with stale criticality information, the placement tool may not notice when it is increasing the critical path delay of the system. On the other hand, the placement performed with frequent static timing analysis shows a much more stably decreasing critical path delay.

This behavior can be demonstrated on the full suite of netlists provided by VPR, 22 of the largest MCNC benchmarks (11 combinational and 11 sequential circuits). Additional information regarding these benchmarks can be found in Appendix A. Table 5.1 shows the results when the amount of static timing analysis is increased during placement. Reported are the normalized geometric mean final placement wirelength and post-routing critical path delay. Testing was performed on the *4lut_sanitized* architecture using a commonly used methodology [1]: minimum sized square arrays with 1.2x the minimum channel width. Stated more plainly, these netlists were mapped to the smallest square array they could fit on and routing was performed in two stages. The first phase of routing searched in a binary fashion to find the minimum channel width architecture that the netlist would route successfully using the timing-driven PathFinder-based router built into VPR. The second routing run used to produce the reported data increased this channel width by 20% to provide a slightly lower-stress routing problem. This increase in channel width is commonly performed to provide slightly more realistic results that better evaluate the quality of the placement tool. This is done for two reasons. First, modern FPGA architectures generally have a very large number of communication channels to increase their flexibility. Thus, designs are typically placed onto systems with very low congestion. Second, this slightly relaxed routing problem avoids the potentially very poor solutions that routers can produce on heavily congested systems. In this type of situation, much of the subtle differences in the quality between different placements are lost because the routed results include so many unpredictably circuitous paths.

These results were obtained with the *A** optimizations [39] option turned off. Although beyond the scope of this discussion, A* is meant to improve routing runtime, without impacting quality. This option was not used because the aggressive implementation built into VPR increased the unpredictably of the routing.

As seen in Table 5.1, simply increasing the amount of static timing analysis resulted in a relatively clear benefit to the average critical path delay. This advantage also seems to get larger given 1 to 100 static timing analysis runs per temperature, peaking at a 0.857x speedup. Updating more frequently than that did not seem to have measurable additional benefit in this testing. That said, while this performance benefit is nice, there is the matter of placement runtime. Although CPU runtime is notoriously difficult to accurately measure, in preliminary testing, placement with 100 static timing analysis runs per temperature iteration took 20x longer to produce than default placement. This is because the time required to perform static timing analysis quickly begins to eclipse the runtime of the other necessary calculations associated with placement.

Aside from the issue of runtime, this performance benefit also seems to come with a small average wire cost penalty. Thus, it is possible that these placements are unfairly taking advantage of the wider communication channels used in this testing process to improve delay. However, as seen in Equation 5.1, VPR has a parameter that can change the emphasis placed on wire cost versus critical path delay. This is the λ term. In addition, as seen in Equation 5.6, VPR also has a parameter that changes the progressive penalty placed on the highest criticality nets. This is the criticality exponent. While in some sense the default parameters suggested by such a rigorously tested toolsuite such as VPR are an interesting starting point, increasing the frequency of static timing analysis by such a large amount does change some of the basic assumptions likely made during the authors' tuning process. Thus, recalibrating the λ and criticality exponent terms seems reasonable.

The testing process was repeated, this time both lowering the λ term to increase the emphasis placed on the wire cost and increasing the criticality exponent to place more pressure on high criticality nets. The results of this testing can be seen in Figure 5.4, with more details in Table 5.2. The default parameters used by VPR are (λ=0.5, crit. exponent = 8). Therefore, the default values can be seen in Figure 5.4 indicated by the black line marked with black circles – the progressive points from the top left to the bottom right denoting 1 to 10,000 static timing analysis runs per temperature iteration. During this testing λ was swept between 0.5 and 0.3 while the criticality exponent was swept between 8 and 12. Based upon the results of this testing, VPR seems to obtain the best placements with the parameters (λ=0.3, crit. exponent = 12) and 10, 000 static timing analysis runs per temperature iteration. Unlike the results obtained with the default parameters, these placements have a lower average wire cost (0.977x) despite their better critical path delay (0.873x). However, this benefit comes with an even larger algorithmic complexity problem since it is obtained with dramatically more static timing analysis. Although annealing with such a large amount of static timing analysis is impractical in most situations, this does provide a point of reference to show what is possible with more accurate timing information.

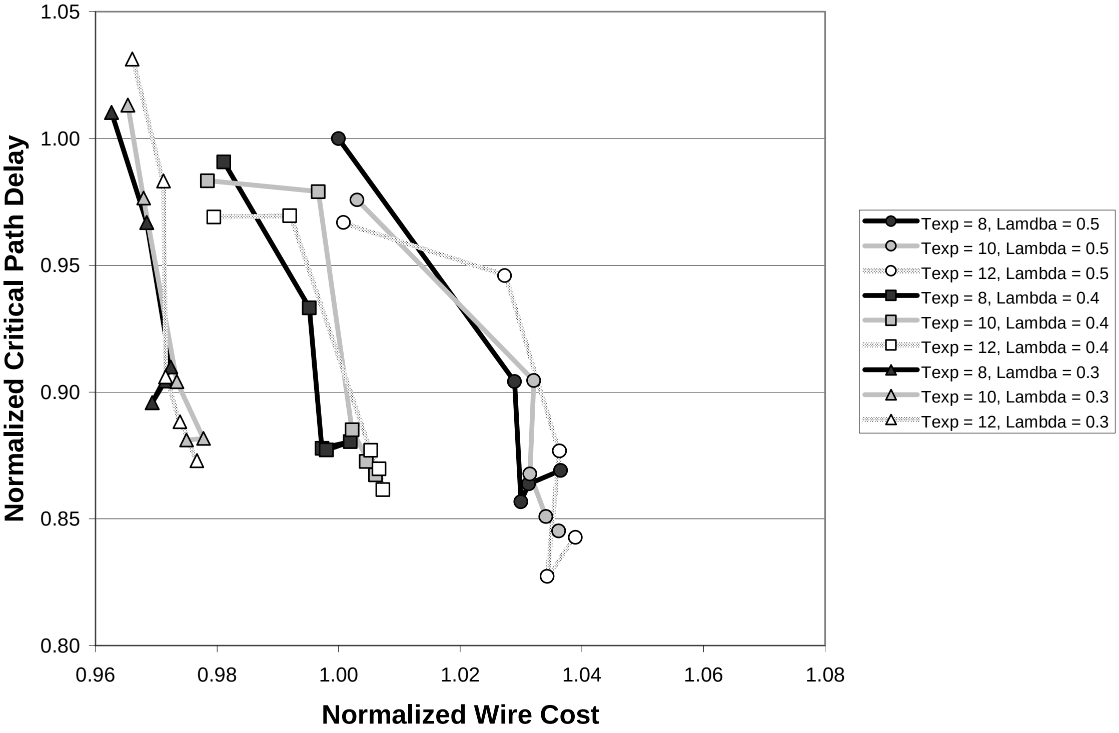


Figure 5.4: VPR Placement λ and Criticality Exponent Tuning for Conventional MCNC Netlists
The top left point of each line represents placement with 1 static timing analysis per temperature iteration. Each subsequent point towards the bottom right denotes 10, 100, 1000 or 10,000 static timing analysis runs per temperature iteration.

Table 5.2: VPR Placement λ and Criticality Exponent Tuning for Conventional MCNC Netlists

		Combinational Circuits Only		Sequential Circuits Only		All Circuits	
Crit Exp , λ	STA	Normalized Wire Cost	Normalized Routed CPD	Normalized Wire Cost	Normalized Routed CPD	Normalized Wire Cost	Normalized Routed CPD
8, 0.5	1	1.000	1.000	1.000	1.000	1.000	1.000
	10	1.020	0.919	1.038	0.890	1.029	0.904
	100	1.016	0.897	1.044	0.819	1.030	0.857
	1000	1.017	0.938	1.045	0.795	1.031	0.864
	10000	1.023	0.921	1.050	0.820	1.036	0.869
10, 0.5	1	1.010	0.974	0.996	0.978	1.003	0.976
	10	1.030	0.928	1.034	0.881	1.032	0.905
	100	1.021	0.920	1.042	0.819	1.031	0.868
	1000	1.025	0.894	1.044	0.810	1.034	0.851
	10000	1.022	0.892	1.050	0.801	1.036	0.845
12, 0.5	1	1.014	0.990	0.988	0.944	1.001	0.967
	10	1.026	1.004	1.028	0.892	1.027	0.946
	100	1.026	0.923	1.047	0.833	1.036	0.877
	1000	1.027	0.893	1.041	0.767	1.034	0.827
	10000	1.029	0.901	1.049	0.788	1.039	0.843
8, 0.4	1	0.988	0.994	0.975	0.988	0.981	0.991
	10	0.995	0.948	0.996	0.919	0.995	0.933
	100	0.994	0.906	1.000	0.851	0.997	0.878
	1000	0.994	0.937	1.010	0.827	1.002	0.880
	10000	0.993	0.923	1.003	0.834	0.998	0.877
10, 0.4	1	0.988	0.998	0.969	0.969	0.978	0.983
	10	0.999	1.004	0.994	0.955	0.997	0.979
	100	0.996	0.940	1.008	0.834	1.002	0.885
	1000	1.005	0.919	1.008	0.819	1.006	0.867
	10000	0.996	0.948	1.013	0.803	1.005	0.873
12. 0.4	1	0.993	0.993	0.966	0.945	0.979	0.969
	10	1.004	0.985	0.980	0.954	0.992	0.970
	100	1.004	0.915	1.007	0.841	1.005	0.877
	1000	1.001	0.887	1.013	0.837	1.007	0.861
	10000	1.003	0.939	1.010	0.806	1.007	0.870
8, 0.3	1	0.976	1.030	0.950	0.991	0.963	1.010
	10	0.977	0.960	0.960	0.974	0.968	0.967
	100	0.978	0.947	0.967	0.874	0.972	0.910
	1000	0.975	0.931	0.963	0.862	0.969	0.896
	10000	0.977	0.935	0.966	0.874	0.971	0.904
10, 0.3	1	0.979	1.001	0.952	1.026	0.965	1.013
	10	0.978	0.968	0.958	0.985	0.968	0.977
	100	0.982	0.942	0.965	0.868	0.973	0.904
	1000	0.980	0.921	0.976	0.844	0.978	0.882
	10000	0.984	0.932	0.966	0.833	0.975	0.881
12, 0.3	1	0.979	1.035	0.953	1.027	0.966	1.031
	10	0.980	0.989	0.962	0.978	0.971	0.983
	100	0.981	0.941	0.962	0.872	0.972	0.906
	1000	0.983	0.931	0.965	0.848	0.974	0.888
	10000	0.985	0.924	0.968	0.825	**0.977**	**0.873**

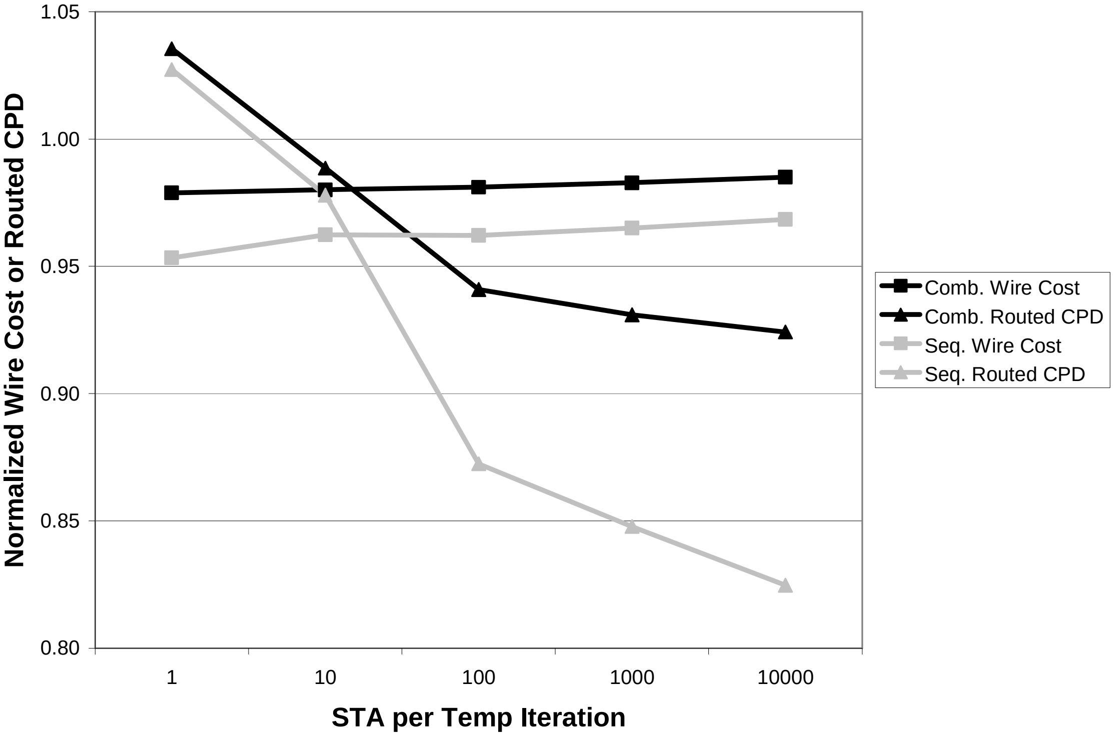


Figure 5.5: Discrepancy in VPR Placement for Conventional Combinational and Sequential MCNC Netlists, (λ = 0.3, Criticality Exponent = 12)

5.3: Characteristics of Registered Applications

One importation observation should be noted before moving on. While essentially all of the benchmarks benefited from the increased accuracy in timing information afforded by a larger amount of static timing analysis during placement, as seen in Figure 5.5 the sequential circuits seemed to respond much more strongly than the purely combinational netlists. Denoted in grey triangles, the improvement in routed critical path delay for the sequential benchmarks is 0.825x while, denoted in black triangles, the improvement for the combinational circuits is 0.924x.

One possible explanation for this phenomenon is that the registers in these sequential benchmarks create some intrinsic characteristic that causes the timing of the system to change much more quickly for these circuits during the annealing process. This would make increasing the accuracy of the timing information during placement far more important; the higher the accuracy, the better the results. Conversely, it can be thought that placement performed in the classical manner can be far more detrimental to sequential circuits. Furthermore, it follows that increasing the number of registers in a netlists may cause this problem to get worse. This is a potentially very significant concern and a concept central to this dissertation.

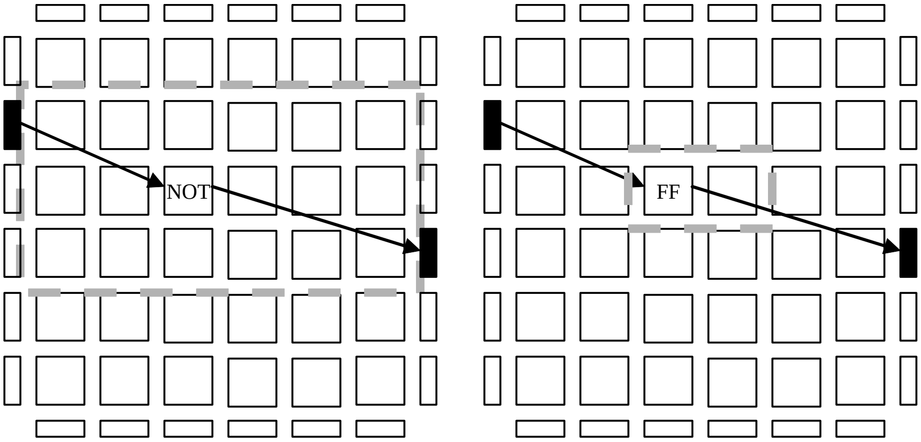


Figure 5.6: Timing Implications of Combinational Logic vs. Registers

A simple thought experiment can illustrate this issue. Consider the combinational circuit on the left of Figure 5.6. If this device has unit-length communication wires, there is a large envelope of locations in which the placer can put the inverter that does not change the timing of the circuit. Delay is simply shifted from the input of the inverter to the output. However, the criticality of all of the nets and the overall timing situation of the system as a whole does not change. Thus, as long as the placer does not elect to move the inverter outside of this window there is very little need to update the timing information. However, for the sequential circuit on the right of Figure 5.6 this is not the case. Here, there is a very small window in which the flip-flop can move that does not make the critical path delay worse. For that matter, even moving the flip-flop to its alternate location changes the criticality of the input and output nets. As will be discussed in the following sections, this makes two issues very important. First, accurately tracking timing information is critical for registered circuits. Second, this information must be carefully applied to obtain high quality placements.

5.4: Registered Netlists & Placement Stability

At first glance, the discussion in Section 5.3 would seem to indicate that computational complexity is the only hurdle for conventional placement with frequent static timing analysis. Also, following the former line of thought, one would expect that it would be highly beneficial to increase the amount of static timing analysis as the number of registers in prospective circuits goes up. However, in practice, heavily registered circuits can actually uncover a unique kind of degenerate situation during this kind of placement. That is, conventional placement with frequent static timing analysis can induce serious annealing convergence problems for these types of netlists. Furthermore, this problem can get worse as the frequency of static timing analysis is increased.

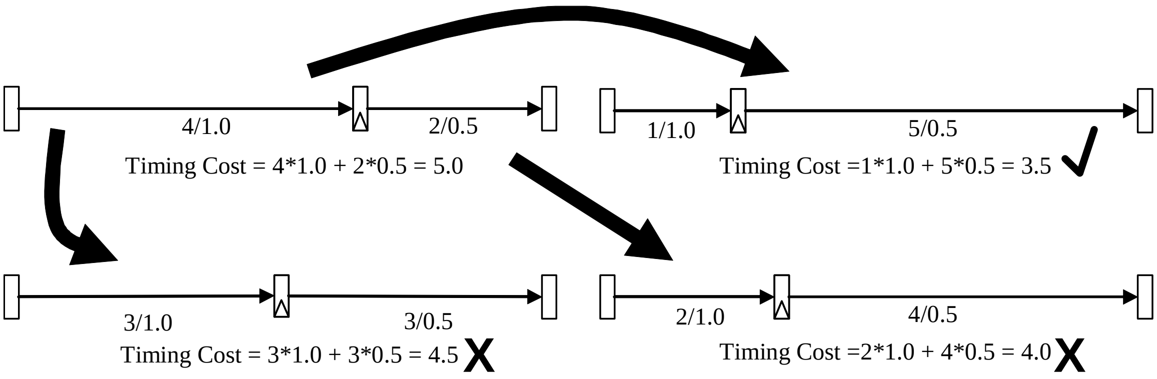

4/1.0
2/0.5
Timing Cost = 4*1.0 + 2*0.5 = 5.0
1/1.0
5/0.5
Timing Cost =1*1.0 + 5*0.5 = 3.5 ✓
3/1.0
3/0.5
Timing Cost = 3*1.0 + 3*0.5 = 4.5 X
2/1.0
4/0.5
Timing Cost =2*1.0 + 4*0.5 = 4.0 X

Figure 5.7: Registered Netlists & Placement Oscillation
Notation on nets: delay / criticality

As shown in Figure 5.7, consider what happens during the placement of a very simple registered circuit. For simplicity sake, the placement of the I/O pins will be fixed and the annealer will only try to find the best location for the register. In this example, the initial placement shown in the top left sets the register slightly off center with regards to the input and output pins. Thus, the input net is 100% critical and the output net is 50% critical. VPR first performs static timing analysis to obtain criticality information. The placer then performs a series of annealing moves based upon this information, and then static timing analysis is repeated to obtain new criticality values. At this point the entire process begins again. Thus, after the net criticalities of this initial placement are determined, the annealer is ready to consider random swaps. Figure 5.7 shows three new possible locations for the register. The bottom left is a placement with the register in the optimal location, the bottom right is a solution that is equally unbalanced in the opposite direction, and the top right shows an even less balanced solution. Unfortunately, VPR will tend toward the arrangement on the top right which has the worst possible critical path delay.

This occurs because the placement tool evaluates new possible locations for the register using old net criticalities. In a similar situation as the example in Figure 5.2, this causes the placer to try and remove as much delay from slow connections as possible. To compensate, this could mean adding as much “cheap” delay as possible to formerly fast connections. This can cause the placement tool to favor increasingly extreme placements, as opposed to better, more moderate solutions. Figure 5.7 shows that, based upon the timing cost of the three alternate placements, the annealer will tend towards the worst solution.

While this can also occur with combinational circuits (it is possible to create a similar situation for the example shown in Figure 5.2), this becomes a larger concern and affects the overall stability of placement for registered netlists because, as discussed earlier, the criticalities of the nets in a registered circuit can change much more rapidly during placement as compared to a purely combinational netlist. Thus, it is far more likely that the placer will find these degenerate situations while placing heavily registered netlists.

Furthermore, as soon as the system performs another timing analysis, the placement problem will reverse and the register will tend to head for the extreme solution in the other direction. In some sense, the register will try to occupy two very different locations depending upon which net it believes is critical. As timing analysis is performed more often, the preferred location of the register will oscillate faster.

This instability in the “optimal” location for registers presents a very difficult, constantly moving target to the annealer and can destabilize the system enough to cause the placement to not converge. This was less of a concern under the classical placement scheme with infrequent static timing analysis because although the placer was not necessarily optimizing towards the correct goal, at least the guiding forces in the placement within a given temperature iteration were consistent. In that way it could always make forward progress, albeit to a potentially less than optimal destination.

The problem with placement convergence can be demonstrated by repeating the static timing analysis testing on heavily registered circuits. As seen in Appendix A, all 22 original MCNC benchmarks were converted into *depth=1* versions. That is, each circuit was pipelined, C-slowed, and Leiserson/Saxe retimed such that the maximum logical depth of the circuit was a single LUT. To most faithfully simulate the modifications that an application developer might perform to optimize a netlist for better throughput, the minimum amount of pipelining and C-slowing was applied to obtain a depth of one LUT.

Figure 5.8 shows two placement runs of the depth=1 *ex5p* netlist. Just as in the example shown in Figure 5.3, placement was performed with both the default one static timing analysis per temperature iteration (shown in black) and 1000 static timing analysis runs per temperature iteration (shown in gray). For this testing the λ and criticality exponent parameters were left at their default values ($\lambda = 0.5$, crit. exponent = 8). Looking at this graph, the placement performed with very frequent timing analysis clearly suffers from convergence issues. First, although the amount of static timing analysis was increased to improve the accuracy of the timing information, the critical path delay for this supposedly enhanced annealing approach never truly improves beyond that of the initial placement. This is most likely due to the tendency for the annealer to pull registers from one degenerate solution to another.

Of even greater concern, this oscillation also seems to affect the basic functionality of the annealer – wirelength optimization. The criticality exponent used by VPR begins at one and is slowly increased during the placement process. Judging by the sudden change in wire cost optimization that occurs around temperature iteration 45, when the system begins to seriously optimize for delay by increasing the criticality exponent, the entire placement process is disrupted. Since the wire cost of the final placement performed with frequent static timing analysis is approximately two to three times that of the results from placement with the default parameters, not only does this placement have an extremely high critical path delay, it will likely fail to route on any architecture with a reasonable channel width. Thus, although a user may attempt to improve critical path delay by updating timing information more often, they may end up derailing the annealer entirely instead.

The instability of placement with frequent timing analysis for all of the depth = 1 MCNC netlists is shown in Table 5.3. As with the earlier testing, the netlists were packed with T-VPack, placed onto minimum-sized *4lut_sanitized* architectures with 1.2x the minimum channel width as found by default VPR and routed using the built-in VPR timing-driven routing tool with *A** disabled. Here, the problems began as soon as the amount of timing analysis is increased beyond the default amount. While performing 10 static timing analysis runs per temperature iteration improves the routed critical path delay for most of the netlists, 3 fail to route due to annealing convergence problems. This issue only gets worse as the amount of

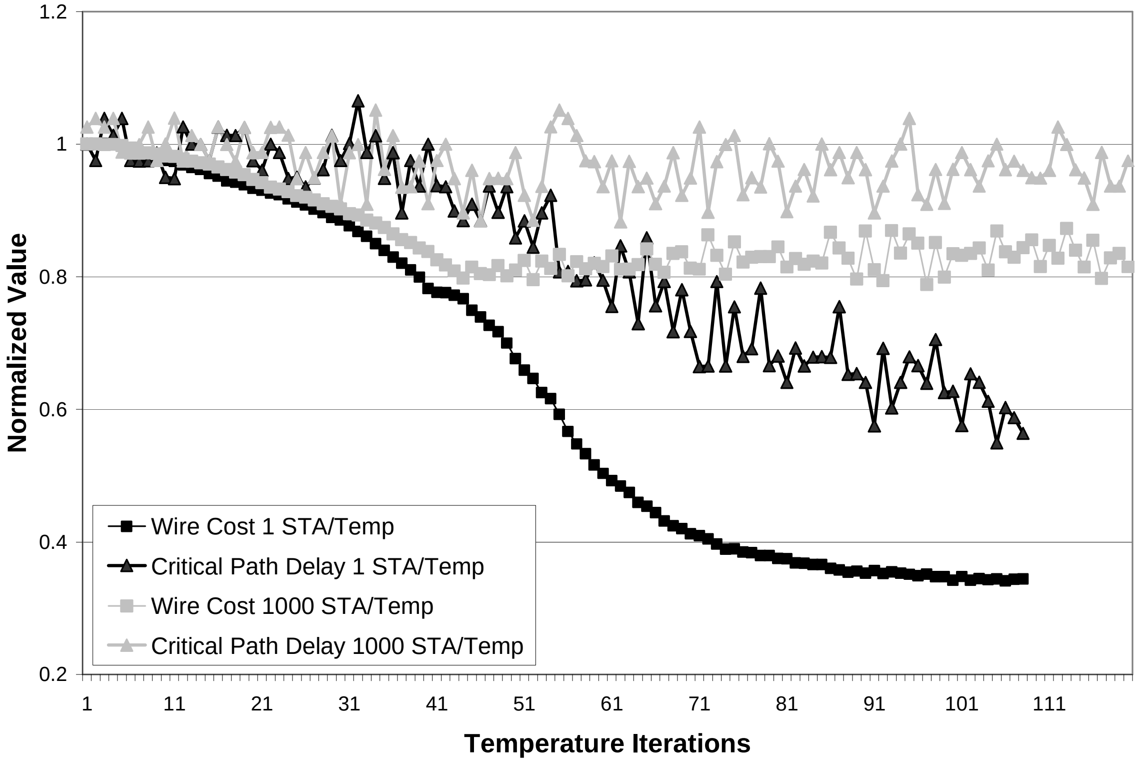


Figure 5.8: VPR Placement Convergence Problem with Depth = 1 MCNC Netlist

Table 5.3. Instability of VPR Placement with Frequent Static Timing Analysis for Depth = 1 MCNC Netlists (Default λ, Default Criticality Exponent)

Static Timing Analysis/Temp	Normalized Wire Cost	Normalized Routed CPD
1	1.000	1.000
10	1.053*	0.952* (3 failed to route)
100	1.031*	0.749* (5 failed to route)
1000	1.106*	0.682* (16 failed to route)

* Indicates that some of the netlists failed to route on the 1.2x minimum channel width architecture.
The wire and routed critical path delay shown exclude the failed netlists.

static timing analysis is increased. 100 static timing analysis runs per temperature iteration cause 5 netlists to have problems and 1000 causes ¾ of the tested netlists to fail routing.

Thus, to allow the placement tool to take advantage of more up-to-date timing information, something must be done to dampen the oscillations in the system. Since these oscillations are caused by the timing optimizations performed by the annealer, reducing the emphasis on timing considerations could solve some of these problems. While in some sense this counteracts the entire purpose of increasing the frequency of static timing analysis, to be completely fair every possibility should be explored. Of the placement parameters available, a user could either reduce λ to emphasize wirelength more heavily or reduce the criticality exponent to lessen the impact of highly critical nets. In a similar manner to the testing used for the conventional lightly registered benchmarks, testing for the depth = 1 circuits was repeated varying both the λ and criticality exponent.

The first phase of testing, shown in Figure 5.9 with details in Table 5.4, investigated the possibility of reducing the criticality exponent from 8 to 1. For a given λ and criticality exponent, the amount of static timing analysis was increased until two or more netlists failed to route on the provided architecture. The testing performed in Table 5.3 is shown in Figure 5.9 with the black circle at (1.00, 1.00). Since performing 10 static timing analysis runs per temperature iteration caused three of the netlists to fail to route, no further points are shown for the default values of (λ = 0.5, crit. exponent = 8). The next test kept the criticality exponent the same, but reduced λ (λ = 0.4, crit. exponent = 8) in the hope that this would achieve better results. Shown in black squares, these parameters indeed performed much better. However, although performing more static timing analysis runs per temperature iteration improved critical path delay significantly, it also encountered some convergence problems that increased the average normalized wire cost. This caused one of the netlists to fail to route at 1,000 static timing analysis runs per temperature iteration and three netlists to fail at 10,000.

Therefore, the next test reduced λ again (λ = 0.3, crit. exponent = 8). Indicated in Figure 5.9 with black triangles, although the average wire cost for routable placements performing anywhere between 1 to 10,000 static timing analysis runs per temperature iteration remains below 1.00, one of the placements obtained performing 1,000 static timing analysis runs per temperature iteration failed to route. Thus, just as a precaution (λ = 0.2, crit. exponent = 8) was tested next. These parameters produced routable placements for all of the tests. However, as indicated with black diamonds, these parameters also begin to trade benefits in critical path delay for an average normalized wire cost far below 1.00. Thus, the best results using a criticality exponent of 8 can probably be obtained with λ = 0.3 and 10,000 static timing analysis runs per temperature iteration.

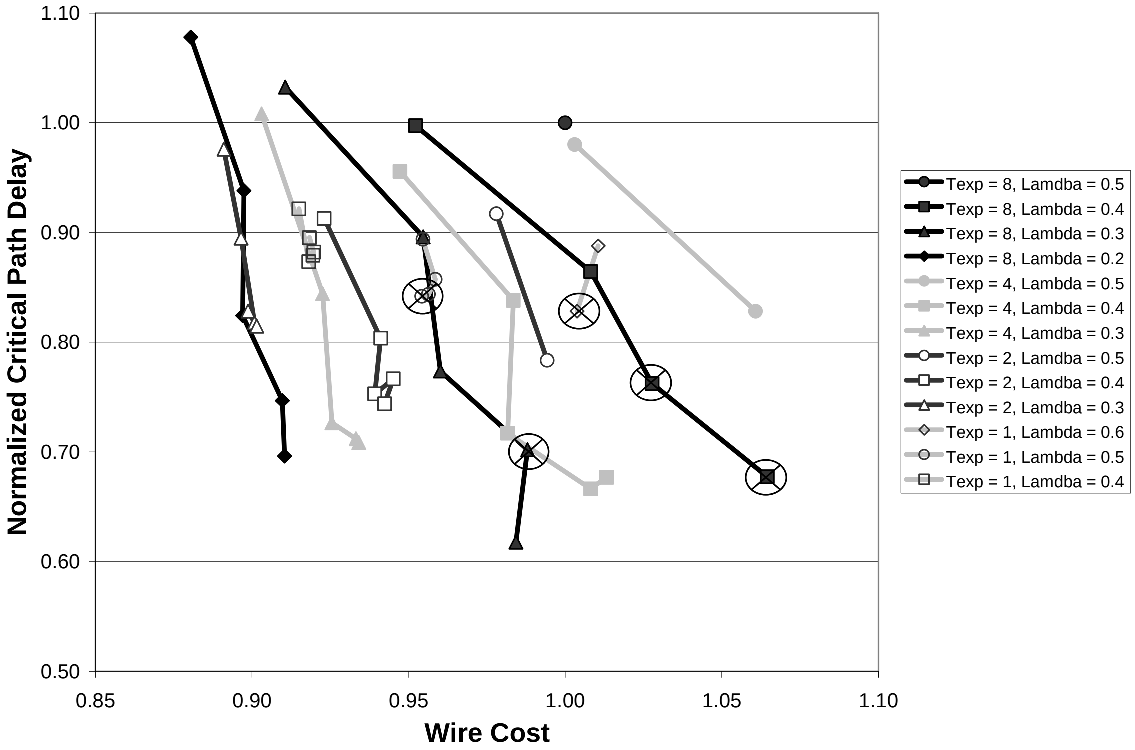


Figure 5.9: VPR Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists, Phase 1
"X" denotes that a single netlist failed to route on the 1.2x minimum channel width architecture. The wire and routed critical path delay shown exclude the failed netlist. Results with more than one unroutable netlist are excluded entirely.

The next round of testing began back at λ = 0.5, but reduced the criticality exponent to 4. The testing methodology used to explore the benefits of reducing λ for a criticality exponent of 8 was repeated. The best results with a criticality exponent of 4 that had an average wire cost below 1.00 were obtained with λ = 0.3 and 10,000 static timing analysis runs per temperature iteration. Similar testing was repeated for criticality exponents of 2 and 1. Based on these results, a second phase of testing, shown in Figure 5.10 and Table 5.5 explored the possibilities of reducing λ further, but increasing the criticality exponent. A similar testing methodology was used to find the best critical path delay results for each criticality exponent from 8 to 12. Like the previous testing, this focused on finding parameters that produced placements with an average normalized wire cost below 1.0.

These two rounds of testing showed that VPR obtained the best placements with the parameters (λ=0.3, crit. exponent = 8) and 10,000 static timing analysis runs per temperature iteration. Although very slow and potentially flirting with instability in the placement, this showed enormous potential. The geometric mean routed critical path delay was improved by 0.618x while the geometric mean wire cost was improved by 0.984x. Furthermore, this testing also corroborates the supposition made in Section 5.3 regarding the

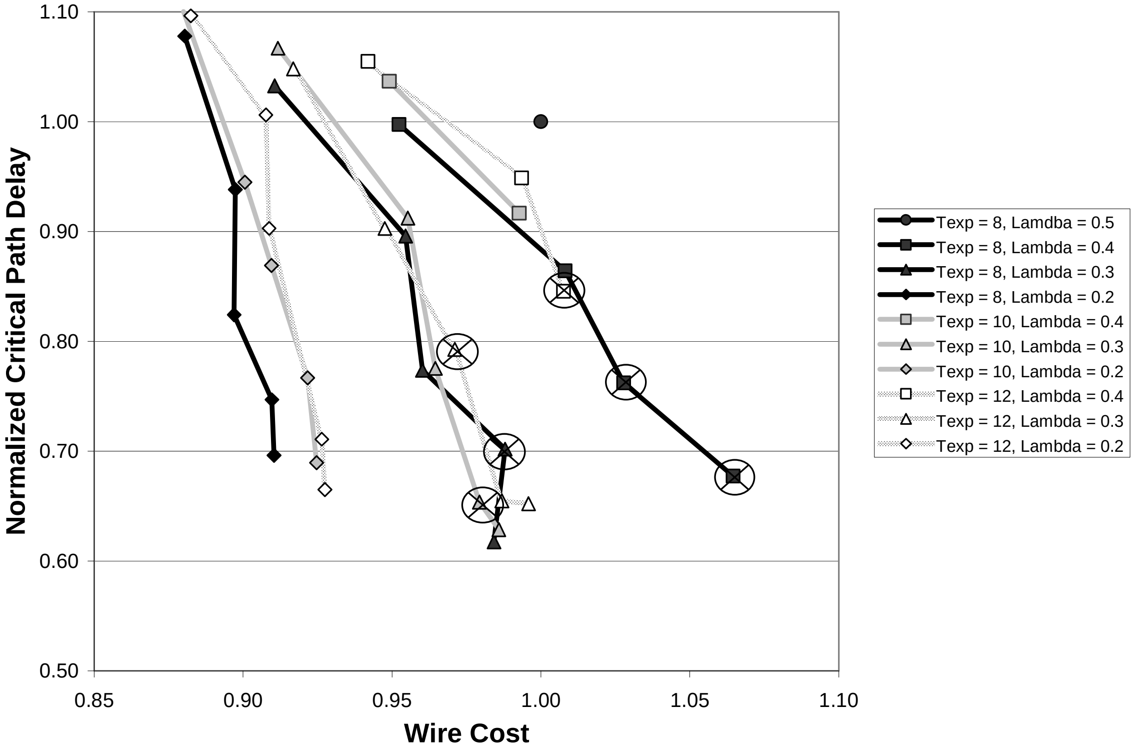


Figure 5.10: VPR Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists, Phase 2

"X" denotes that a single netlist failed to route on the 1.2x minimum channel width architecture. The wire and routed critical path delay shown exclude the failed netlist. Results with more than one unroutable netlist are excluded entirely.

way that registers affect circuit timing during placement. As seen in Figure 5.11, the large discrepancy between the benefits seen by combinational and sequential circuits has largely evaporated. This is likely because both sets of netlists now contain a large number of registers, making all of them relatively sensitive to stale timing information.

Taking a step back for a moment, the difficulties encountered producing high-quality timing-driven placements, particularly for pipelined netlists, should not be surprising. Placement for pipelined netlists has been a known difficult problem for some time. For example, the deeply pipelined radio cross-correlator in [41] was laboriously hand-placed by the author to achieve good performance. This painstaking process even inspired the authors of [4] to develop a specific tool to assist in manual pipelining and placement. The extreme difficulty of such an endeavor, given the scale of even relatively small FPGA designs, is likely indicative of the complexities these netlists present to the design flow.

Table 5.4: VPR Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists, Phase 1

		Combinational Circuits Only		Sequential Circuits Only		All Circuits	
Crit Exp , λ	STA	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD
8, 0.5	1	1.000	1.000	1.000	1.000	1.000	1.000
	10	1.040	0.982	1.071*	0.911* (3)	1.053*	0.952* (3)
	100	1.016	0.767	1.060*	0.719* (5)	1.031*	0.749* (5)
	1000	1.176*	0.687* (9)	1.072*	0.680* (7)	1.106*	0.682* (16)
8, 0.4	1	0.969	1.005	0.936	0.990	0.952	0.997
	10	0.990	0.882	1.027	0.846	1.008	0.864
	100	0.995	0.783	1.065*	0.740* (1)	1.028*	0.762* (1)
	1000	1.047	0.707	1.084*	0.646* (1)	1.065*	0.677* (1)
	10000	1.089	0.736	1.112*	0.585* (3)	1.099*	0.668* (3)
8, 0.3	1	0.939	1.011	0.883	1.055	0.911	1.033
	10	0.947	0.885	0.962	0.907	0.955	0.896
	100	0.942	0.835	0.978	0.717	0.960	0.774
	1000	0.963	0.733	1.014*	0.669* (1)	0.988*	0.702* (1)
	10000	0.952	0.628	1.018	0.607	**0.984**	**0.618**
8, 0.2	1	0.900	1.005	0.861	1.157	0.880	1.078
	10	0.912	0.926	0.883	0.950	0.897	0.938
	100	0.910	0.808	0.884	0.841	0.897	0.824
	1000	0.909	0.722	0.911	0.772	0.910	0.747
	10000	0.912	0.706	0.908	0.687	0.910	0.696
4, 0.5	1	0.999	0.942	1.007	1.020	1.003	0.980
	10	1.022	0.823	1.111*	0.835* (1)	1.061*	0.828* (1)
	100	1.008	0.688	1.079*	0.624* (5)	1.033*	0.664* (5)
4, 0.4	1	0.950	0.901	0.945	1.014	0.947	0.956
	10	0.964	0.815	1.003	0.862	0.983	0.838
	100	0.959	0.741	1.005	0.693	0.982	0.717
	1000	0.965	0.682	1.054	0.652	1.008	0.666
	10000	0.966	0.689	1.062	0.665	1.013	0.677
4, 0.3	1	0.914	0.966	0.892	1.052	0.903	1.008
	10	0.923	0.828	0.922	0.861	0.923	0.844
	100	0.926	0.738	0.925	0.716	0.926	0.726
	1000	0.924	0.715	0.942	0.708	0.933	0.712
	10000	0.923	0.755	0.946	0.664	0.934	0.708
2, 0.5	1	0.972	0.902	0.984	0.932	0.978	0.917
	10	0.975	0.817	1.014	0.751	0.994	0.783
	100	0.980	0.719	1.008*	0.668* (2)	0.993*	0.695* (2)
2, 0.4	1	0.931	0.893	0.915	0.932	0.923	0.913
	10	0.933	0.821	0.949	0.786	0.941	0.804
	100	0.936	0.784	0.943	0.723	0.939	0.753
	1000	0.940	0.793	0.950	0.741	0.945	0.766
	10000	0.936	0.765	0.949	0.723	0.942	0.744
2, 0.3	1	0.904	0.936	0.878	1.018	0.891	0.976
	10	0.906	0.850	0.888	0.942	0.897	0.895
	100	0.905	0.813	0.897	0.823	0.901	0.818
	1000	0.908	0.850	0.889	0.807	0.899	0.828
	10000	0.906	0.837	0.897	0.793	0.901	0.815
1, 0.6	1	1.003	0.891	1.018	0.884	1.011	0.888
	10	0.996	0.845	1.013*	0.810* (1)	1.004*	0.828* (1)
	100	0.994	0.876	1.011*	0.867* (2)	1.002*	0.872* (2)
1, 0.5	1	0.956	0.909	0.953	0.879	0.955	0.894
	10	0.956	0.884	0.961	0.831	0.959	0.857
	100	0.955	0.860	0.958	0.827	0.956	0.844
	1000	0.958	0.875	0.950*	0.806* (1)	0.954*	0.842* (1)
	10000	0.957	0.845	0.958*	0.810* (2)	0.957*	0.829* (2)
1, 0.4	1	0.921	0.942	0.909	0.902	0.915	0.922
	10	0.920	0.899	0.916	0.848	0.918	0.873
	100	0.926	0.882	0.913	0.876	0.920	0.879
	1000	0.923	0.889	0.917	0.875	0.920	0.882
	10000	0.925	0.909	0.911	0.882	0.918	0.895

*Indicates that some of the netlists failed to route on the 1.2x minimum channel width architecture provided. The number of failed netlists is indicated in parenthesis. The wire and routed critical path delay shown exclude the failed netlists.

Table 5.5: VPR Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists, Phase 2

		Combinational Circuits Only		Sequential Circuits Only		All Circuits	
Crit Exp , λ	STA	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD
8, 0.5	1	1.000	1.000	1.000	1.000	1.000	1.000
	10	1.040	0.982	1.071*	0.911* (3)	1.053*	0.952* (3)
	100	1.016	0.767	1.060*	0.719* (5)	1.031*	0.749* (5)
	1000	1.176*	0.687* (9)	1.072*	0.680* (7)	1.106*	0.682* (16)
8, 0.4	1	0.969	1.005	0.936	0.990	0.952	0.997
	10	0.990	0.882	1.027	0.846	1.008	0.864
	100	0.995	0.783	1.065*	0.740* (1)	1.028*	0.762* (1)
	1000	1.047	0.707	1.084*	0.646* (1)	1.065*	0.677* (1)
	10000	1.089	0.736	1.112*	0.585* (3)	1.099*	0.668* (3)
8, 0.3	1	0.939	1.011	0.883	1.055	0.911	1.033
	10	0.947	0.885	0.962	0.907	0.955	0.896
	100	0.942	0.835	0.978	0.717	0.960	0.774
	1000	0.963	0.733	1.014*	0.669* (1)	0.988*	0.702* (1)
	10000	0.952	0.628	1.018	0.607	**0.984**	**0.618**
8, 0.2	1	0.900	1.005	0.861	1.157	0.880	1.078
	10	0.912	0.926	0.883	0.950	0.897	0.938
	100	0.910	0.808	0.884	0.841	0.897	0.824
	1000	0.909	0.722	0.911	0.772	0.910	0.747
	10000	0.912	0.706	0.908	0.687	0.910	0.696
10, 0.4	1	0.968	1.005	0.930	1.070	0.949	1.037
	10	0.994	0.898	0.992	0.936	0.993	0.917
	100	0.991	0.818	1.063*	0.764* (2)	1.026*	0.793* (2)
10, 0.3	1	0.935	0.988	0.889	1.152	0.912	1.067
	10	0.951	0.898	0.960	0.927	0.955	0.912
	100	0.946	0.795	0.984	0.757	0.965	0.775
	1000	0.963	0.690	0.997*	0.616* (1)	0.980*	0.654* (1)
	10000	0.969	0.675	1.003	0.585	0.986	0.629
10, 0.2	1	0.900	1.049	0.861	1.154	0.880	1.100
	10	0.916	0.904	0.885	0.988	0.901	0.945
	100	0.923	0.910	0.896	0.830	0.910	0.869
	1000	0.919	0.746	0.925	0.788	0.922	0.767
	10000	0.923	0.720	0.926	0.661	0.925	0.690
12. 0.4	1	0.966	1.031	0.919	1.079	0.942	1.055
	10	0.990	0.919	0.997	0.980	0.994	0.949
	100	0.988	0.846	1.034*	0.845* (1)	1.011*	0.846* (1)
	1000	1.064	0.748	1.070*	0.633* (2)	1.067*	0.694* (2)
12, 0.3	1	0.939	1.029	0.895	1.067	0.917	1.048
	10	0.953	0.912	0.942	0.893	0.948	0.903
	100	0.955	0.861	0.987*	0.724* (1)	0.971*	0.793* (1)
	1000	0.978	0.648	0.995	0.662	0.987	0.655
	10000	0.995	0.681	0.997	0.625	0.996	0.652
12, 0.2	1	0.910	1.035	0.856	1.162	0.882	1.096
	10	0.925	0.975	0.891	1.038	0.908	1.006
	100	0.919	0.888	0.899	0.918	0.909	0.903
	1000	0.921	0.678	0.932	0.746	0.926	0.711
	10000	0.930	0.674	0.925	0.656	0.927	0.665

* Indicates that some of the netlists failed to route on the 1.2x minimum channel width architecture provided. The number of failed netlists is indicated in parenthesis. The wire and routed critical path delay shown exclude the failed netlists.

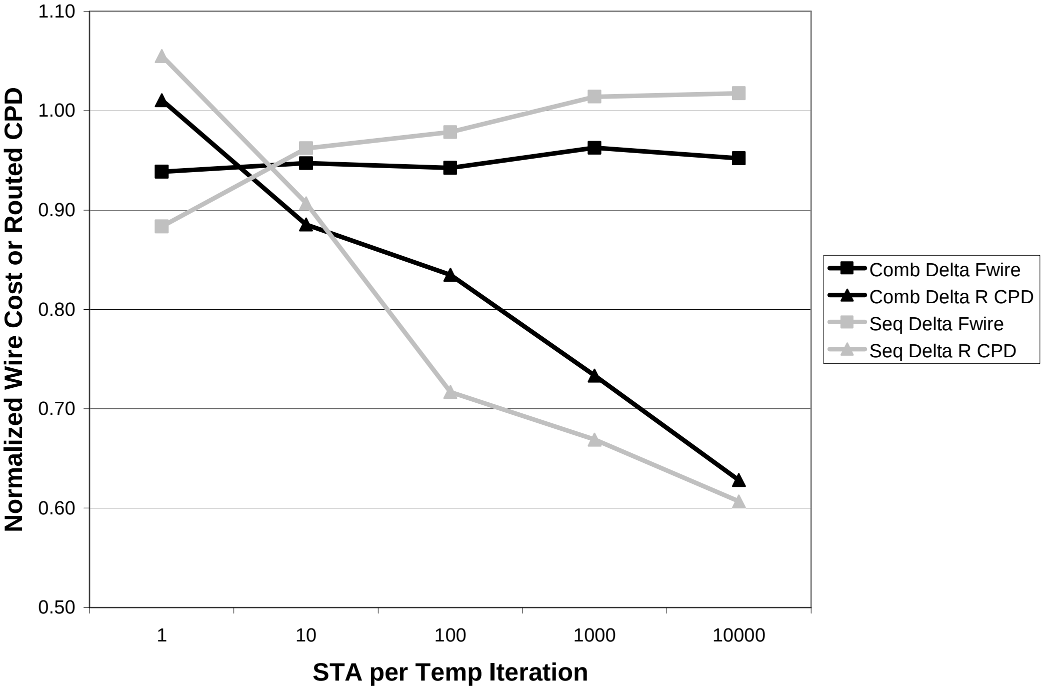


Figure 5.11: Similarity in VPR Placement for Depth = 1 Combinational and Sequential MCNC Netlists, (λ = 0.3, Criticality Exponent = 8)

5.5: Efficient and Stable Placement

Looking back at the problems encountered during placement, two primary issues come forward. First, to produce high quality placements the annealer must have up-to-date criticality information. How can this be obtained without resorting to the computationally impractical solution of performing a full static timing analysis after each move? Second, worrisome instability develops during the annealing process when fresh timing information is used during the placement of registered netlists. What can be done to stabilize the system?

Current timing information can be obtained with low computational effort by making simple incremental changes to link slack. Although the methodology outlined in this section can only, in the worst case, estimate criticality, it does provide enough information to the placement tool to reveal shifts in timing significance. While nothing can replace a full static timing analysis performed at the beginning of each temperature iteration, this approach can help maintain the relevance of criticality information in the meantime by reflecting changes in link delay on link slack.

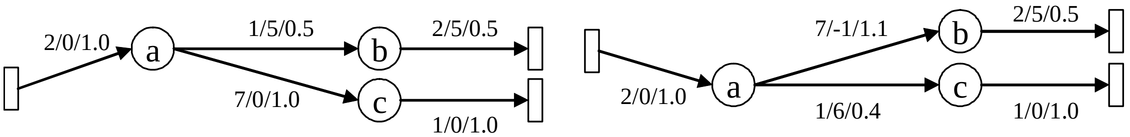
Each time an annealing move is made, VPR’s timing-driven placement algorithm already evaluates the change in link delay for all sources and sinks connected to the migrated blocks. This is seen in Equation 5.3. However, as seen in Equations 5.9 and 5.10, if this change in link delay is subtracted from the link slack, an estimated source/sink criticality for the new placement can be easily recalculated. While less accurate than a complete timing analysis, this only requires two additional add/subtracts and one multiplication/division to preserve the majority of the accuracy of the netlist's criticality information.

$$Slack(i, j, k, l) = Slack(i, j, k, l-1) - \Delta Delay(i, j, k, l) \qquad (5.9)$$

$$Criticality(i, j, k, l) = 1 - \frac{Slack(i, j, k, l)}{D_{\max}(k)} \qquad (5.10)$$

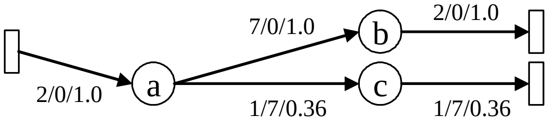
The top left and top right illustrations of Figure 5.12 show this technique in action. Here, the example from Figure 5.2 is revisited, but now the placement tool incrementally updates the slack and link criticality information. The suggested move decreases the delay on (*a*, *c*) by six units from 7 to 1 and increases the delay on (*a*, *b*) by six units from 1 to 7. To evaluate the quality of the new placement, this change is reflected on the links' slacks. Since (*a*, *c*) was on the critical path, the original slack was 0. Thus, the six unit drop in delay can be accounted for and the new slack on this link becomes (0 - (-6) = 6). This updated slack can then be easily turned into a new criticality. In this case, the system still believes that the critical path is 10 units, so the new criticality of (*a*, *c*) is 0.4. Similarly the six unit increase in delay on (*a*, *b*) can be accounted for by updating the slack to (5 - 6 = -1). This makes the criticality of this link 1.1. Finally, the timing cost of this new placement can be computed based upon the incrementally updated timing information. From this the annealer can now see that the new placement is not as good as the previous one.

Although this methodology does effectively address the large-scale problem of placement in the face of inaccurate timing information, it should be noted that this technique cannot guarantee perfect criticality



Initial placement & static timing analysis:
Timing Cost = (2+7+1)*1.0 + (1+2)*0.5 = 11.5
Critical Path Delay = 10

If slack updated incrementally, then link criticality recalculated:
Timing Cost = (2+1)* 1.0 + 7*1.1 + 1*0.4 + 2*0.5 = 12.1
Still believes critical path is 10



If full timing analysis is run instead:
Timing Cost =(2+7+2)* 1.0 + (1+1)*0.36 = 11.72
Critical Path = 11

Figure 5.12: Incremental Slack, Criticality Updating and Accuracy

data – that would require true static timing analysis. The bottom diagram of Figure 5.12 shows the link slack, criticality and timing cost of the new placement as calculated with exact static timing analysis information. Comparing the details of calculations performed for the two techniques, there are at least two small problems. First, the incremental slack approach does not realize that the current critical path delay for the system has changed. Second, the emphasis placed on the links between blocks *b* and *c* and the output pads is incorrect. However, the suggested estimates do track well, especially considering the extremely low computational requirements. Furthermore, the accuracy of this technique is particularly high for heavily registered circuits. The problem of inaccuracy mainly stems from the fact that this approach only updates the criticality information of the nets directly connected to the moved block. However, the timing of the links between blocks *b* and *c* and the output pads changes because *b* and *c* are logic blocks. If these were registers, the criticality of the connections to the output pads would not change unless the critical path delay of the entire system changed. Thus, because the computation is broken into so many separate pieces in a heavily registered netlist, this technique largely correctly calculates link criticality, at least relative to the critical path delay found during the last static timing analysis.

However, absolutely perfect timing information is not necessarily desirable. Rather, relative criticality is far more important. Figure 5.13 revisits the registered example from Figure 5.7, but calculates the timing cost before and after a move with completely correct timing information. Using this methodology, the placement tool does shy away from the more unbalanced solution in the top right, but still tends towards the equally unbalanced solution on the bottom right. The optimal solution on the bottom left is not chosen because both the input and output nets are critical. Although the critical path delay is lower, two critical links become more expensive as compared to one critical and one semi-critical connection. To prevent this, the placement tool must take into account the relative criticality of links before and after each move. Figure 5.14 again calculates the timing cost before and after a move, but now uses the old critical path delay to calculate the criticality of links in the new placement. This technique allows the system to realize

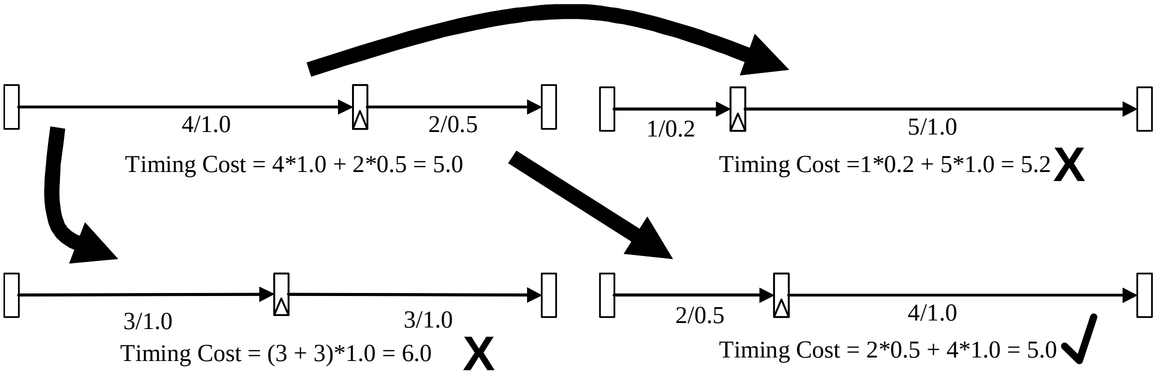

4/1.0
2/0.5
Timing Cost = 4*1.0 + 2*0.5 = 5.0
1/0.2
5/1.0
Timing Cost =1*0.2 + 5*1.0 = 5.2
X
3/1.0
3/1.0
Timing Cost = (3 + 3)*1.0 = 6.0
X
2/0.5
4/1.0
Timing Cost = 2*0.5 + 4*1.0 = 5.0
✓

Figure 5.13: Problems with Perfect Timing Information

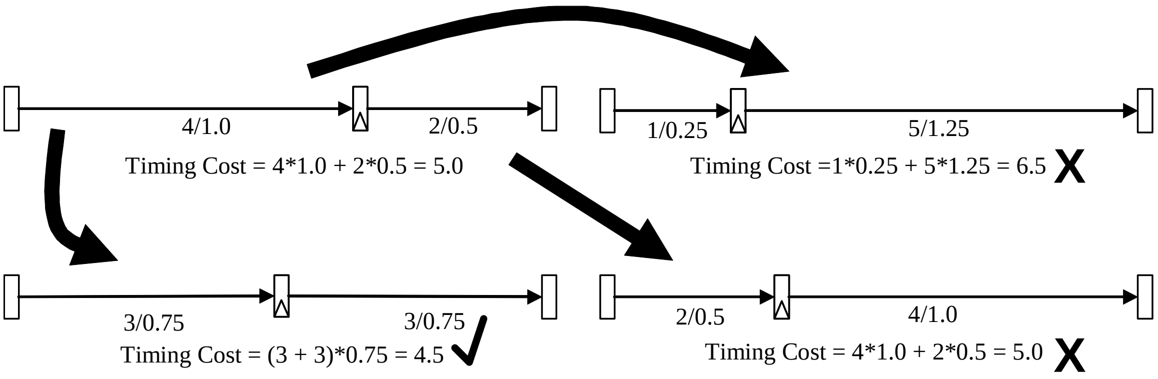

4/1.0
2/0.5
Timing Cost = 4*1.0 + 2*0.5 = 5.0
1/0.25
5/1.25
Timing Cost =1*0.25 + 5*1.25 = 6.5
X
3/0.75
3/0.75
Timing Cost = (3 + 3)*0.75 = 4.5
✓
2/0.5
4/1.0
Timing Cost = 4*1.0 + 2*0.5 = 5.0
X

Figure 5.14: Calculating Relative Change in Criticality

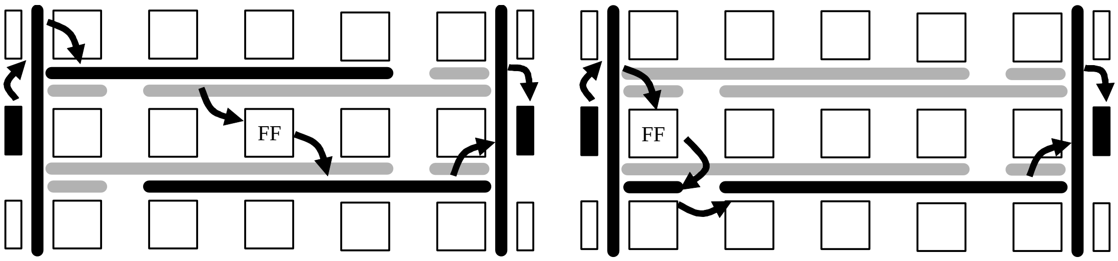
that reducing the overall critical path delay can be far more important, even if this means creating multiple highly critical connections. This type of behavior is naturally built into the incremental update technique.

This discussion leads directly to the second issue – what is the source of the instability in the classical placement technique for registered circuits? Essentially, the problems encountered were caused by the fact that even when the tool updated its timing information, it did not account for the change in criticality before and after a given move. This created a mismatch between the real criticalities in the new placement and the criticalities used to calculate the cost of the new placement. This caused the system to unwittingly prefer unbalanced delay, which opened the door for potential oscillation. However, since the incremental criticality updating technique described above makes it possible to evaluate relative timing information after every single move, the placement tool can compare the cost of the old placement, calculated with the old criticalities, with the cost of the new placement, calculated with the new relative criticalities. This leads to subtle, yet extremely importance difference in the cost function.

More formally, given the incremental slack update approach in Equations 5.9 and 5.10, the new criticality of each source/sink link is determined after a move based upon the critical path delay of the system found at the beginning of the temperature iteration. Since the delay of each source/sink pair is updated after each move, the timing cost of a given placement can be defined as the summation of all source/sink delays multiplied by their current estimated criticality. This is shown in Equations 5.11 and 5.12

$$\text{Timing\_Cost}(i, j, k, l) = Delay(i, j, k, l) * Criticality(i, j, k, l)^{Crit\_Exp} \qquad (5.11)$$

$$\text{Timing\_Cost}(k, l) = \sum \text{Timing\_Cost}(i, j, k, l) \qquad (5.12)$$



Balanced Delay
Delay_x ➔ FF ➔ Delay_x
Critical Path = Delay_x

Imbalanced, Contracted Delay
Delay_y ➔ FF ➔ Delay_z
Critical Path = Delay_z

$\text{Delay}_z > \text{Delay}_x$, but $2 * \text{Delay}_x > \text{Delay}_y + \text{Delay}_z$

Figure 5.15: Example of Contraction and Imbalance

Taking a look at how this affects the way changes between two placements actually manifest, the timing cost delta is now calculated in an inherently different way. This is shown in Equation 5.13. Here, the previous delay is multiplied by the previous criticality and the new delay is multiplied by the new criticality. This is quite different from the timing cost delta shown in Equation 5.3 and leans heavily towards the most accurate algorithm suggested by the example in Figure 5.14.

$$\Delta \mathrm{TC}(i,j,k,l) = \begin{bmatrix} Delay(i,j,k,l)\,* \\ Criticality(i,j,k,l)^{Crit\_Exp} \end{bmatrix} - \begin{bmatrix} Delay(i,j,k,l-1)\,* \\ Criticality(i,j,k,l-1)^{Crit\_Exp} \end{bmatrix} \quad (5.13)$$

5.6: Delay Imbalance and Optimality

One key feature of the approach described above is that it removes the tendency of the system to prefer unbalanced placements. However, the examples shown thus far have made some assumptions regarding the underlying architecture. If a netlist is placed on a device that provides different resources, this can change the behavior of the suggested technique and may cause the annealer to favor unbalanced placements.

For example, the scenario in Figure 5.14 assumes that as long as the register is placed somewhere between the input and output pins, the total amount of delay on the input and output nets summed together will be the same regardless of the balance between these two connections. That is, to make one link slower, another link must get faster by an equal amount. While this is generally true, this is not necessarily the case, particularly in devices with longer wire segments. This difference in total delay along a path can affect the way the placer deals with balanced versus unbalanced connections.

Consider the two placements in Figure 5.15 on an architecture with length-four wires. The placement on the left is faster because the delay between the input and output connections is balanced. However, the annealer may prefer the placement on the right because the total delay on the input and output nets is slightly smaller. Rather than using four full wires, the placement on the right uses three full wires and a short length-one stub. The timing cost for the two placements is shown in Equation 5.14 and Equation 5.15.

$$\text{Timing Cost}_{\text{balanced}} = 2 * Delay_x \left(\frac{Delay_x}{CPD} \right)^{Crit\_Exp} \tag{5.14}$$

$$\text{Timing Cost}_{\text{Imbalanced}} = Delay_y \left(\frac{Delay_y}{CPD} \right)^{Crit\_Exp} + Delay_z \left(\frac{Delay_z}{CPD} \right)^{Crit\_Exp} \tag{5.15}$$

Comparing these two equations and simplifying, the annealer will prefer the unbalanced placement if Equation 5.16 is true. Thus, some obvious questions are: 1) for what values of $Delay_y$ and $Delay_z$ does this relationship hold, and 2) how much slower can $Delay_z$ be compared to $Delay_x$?

$$2 * Delay_x{}^{(Crit\_Exp+1)} > Delay_y{}^{(Crit\_Exp+1)} + Delay_z{}^{(Crit\_Exp+1)} \tag{5.16}$$

To answer these questions, the three delay terms can be related to each other by incorporating two additional variables: a *contraction* term and a *balance* term. As seen in Equation 5.17, the contraction term defines how much smaller the total delay of the unbalanced placement is compared to the balanced placement. As seen in Equation 5.18, the balance term determines how much larger $Delay_z$ is compared to $Delay_y$.

$$Delay_y + Delay_z = Contraction * 2 * Delay_x \tag{5.17}$$

$$Delay_y = Balance * Delay_z \tag{5.18}$$

Plugging Equations 5.17 and 5.18 into Equation 5.16 and solving for the contraction term results in Equation 5.19.

$$Contraction < \left(\frac{2 * (1 + Balance)^{(Crit\_Exp+1)}}{2^{(Crit\_Exp+1)} * (1 + Balance^{(Crit\_Exp+1)})} \right)^{1/(Crit\_Exp+1)} \tag{5.19}$$

This equation can then be graphed varying both the balance and the criticality exponent terms. This is shown in Figure 5.16. Any contraction value below the indicated lines will cause the annealer to prefer the unbalanced placement. For example, using a criticality exponent of 1 allows the system to prefer unbalanced placements with relatively little contraction. $Delay_z$ can be twice as large as $Delay_y$ (balance = 0.5) as long as the total amount of delay along the unbalanced placement is less than about 0.95x the total delay along the balanced placement ($Delay_z + Delay_y \leq 0.95 * [Delay_x + Delay_x]$).

These contraction and balance values can be plugged back into Equation 5.17 and 5.18 to get the values of $Delay_z$, normalizing $Delay_x$ to 1.0. This is shown in Figure 5.17. In this case, the annealer will prefer the unbalanced placement if $Delay_z$ is below the values indicated by the various lines. For the parameters used previously (crit exponent = 1, balance = 0.5, contraction ≈ 0.95), this means that $Delay_z$ can be nearly 1.27x $Delay_x$.

However, taking at closer look at Figure 5.16 and Figure 5.17, the potential sub-optimality of the placement tool cannot get very bad for typical criticality exponent values. From the prospective of placement imbalance, the slope of the criticality exponent 8, 10 and 12 lines in Figure 5.16 is relatively high. For example, an imbalance of ($Delay_x = 0.75 * Delay_z$) requires contraction factor of less than about 0.93x for any criticality exponent larger than 8. However, it is unlikely that such paths will exist in real FPGAs. While there may be a slight difference between the fastest paths through different register locations in some architectures, this difference will likely be relatively small, perhaps no more than a few percent. Therefore, it is unlikely that the placement tool will encounter a situation in which such a viable unbalanced placement exists.

For that matter, this will also generally not affect the final critical path delay. This is because, as seen in Figure 5.17, the maximum allowable values of $Delay_z$ drop very quickly as the criticality exponent is raised. For criticality exponents of 8, 10 and 12, $Delay_z$ can only become about 1.08x, 1.07x, and 1.05x worse, respectively. Thus, while the system may prefer unbalanced placements to a certain extent under some special circumstances, the potential for this to cause larger problems is likely relatively low. This is particularly true if the criticality exponent is kept relatively high. Although unbalanced placements will not genuinely affect the testing performed in this chapter since the architecture used has unit-length wires, it is likely best to keep the criticality exponent as high as possible. This will become important in Chapter 6.

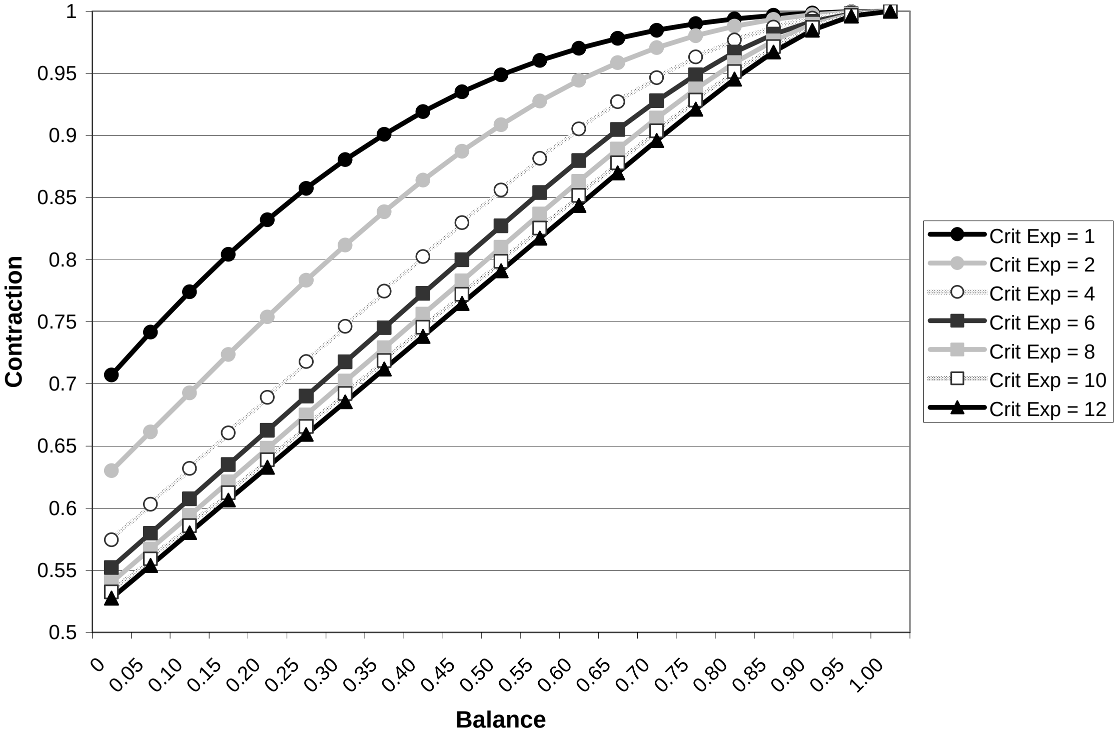


Figure 5.16: Total Delay Contraction as a Function of Criticality Exponent and Balance

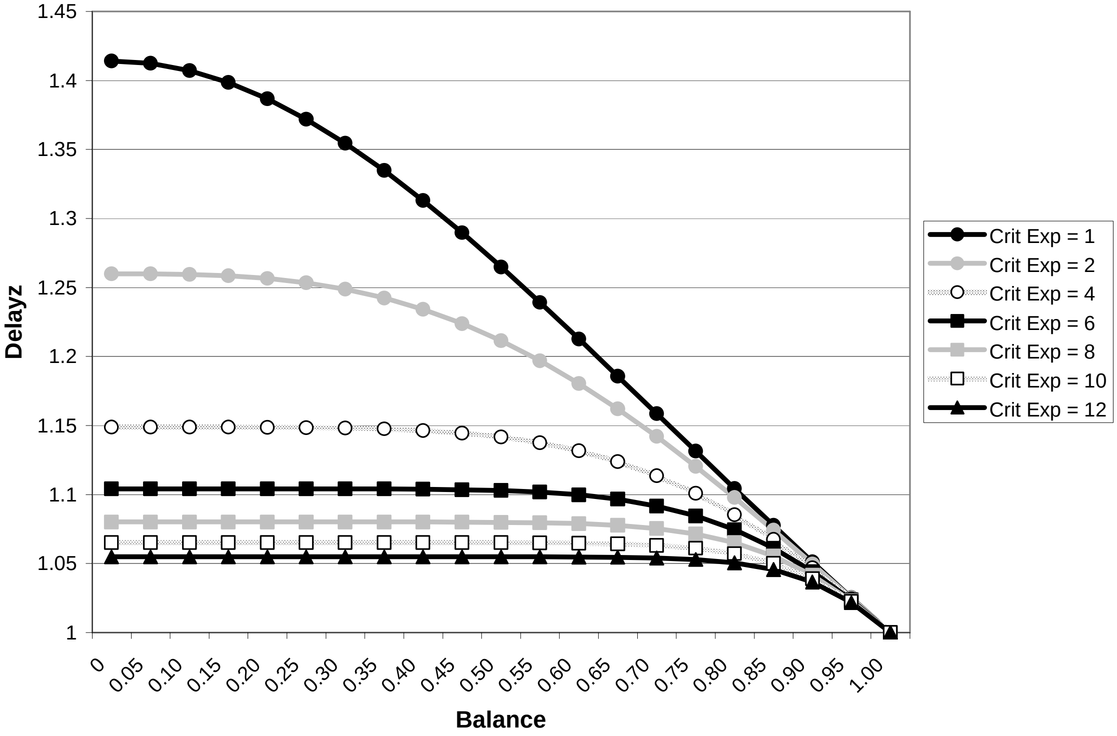


Figure 5.17: Solving for Maximum $Delay_z$ Values

5.7: Testing and Results

The improved timing-driven placement technique with incremental criticality updating and the reformulated cost function was tested using the same set of 22 classical and 22 depth = 1 MCNC netlists. With the exception of the placement tool, all other considerations were kept the same. That is, the netlists were packed with T-VPack, placed onto minimum-sized *4lut_sanitized* architectures with 1.2x the minimum channel width as found by default VPR and routed using the built-in VPR timing-driven routing tool with A^* disabled. Due to the fundamental changes made to the annealing structure, different λ and criticality exponent parameters were explored. The results of this testing for the original MCNC netlists are shown in Figure 5.18 and the results for the depth = 1 netlists are shown in Figure 5.19. More detailed results for this tuning process are provided in Table 5.6 and Table 5.7. As with the earlier testing, to provide easy comparison with the results from VPR, all wire costs and post-routing critical path delays reported have been normalized to the default VPR results. Also, as in the earlier testing, the best placement parameters were determined by selecting the results with the best geometric mean critical path that still maintained a geometric mean wire cost below 1.0.

The most obvious result of this testing is that placement with the new cost formulation requires much smaller values of λ to produce good results. While VPR obtained the best placements with ($\lambda = 0.3$), this new tool required ($\lambda = 0.1$) for the conventional MCNC netlists and ($\lambda = 0.025$) for the heavily registered circuits. Looking at Equations 5.9 and 5.10, the cause of this tendency becomes clear. When the new placer reduces delay on a given link, from the standpoint of the classical VPR framework, the modified cost formulation somewhat double-counts this reduction. This is because, unlike what VPR is expecting, the criticality of this link will also be updated to reflect the smaller delay. Thus, when the two factors are multiplied together, the new delta timing cost is naturally much larger than the range that the existing VPR framework is expecting. A similar situation holds true for when delay is increased on a given link.

Looking at the results in Figure 5.18 and Table 5.6, the new incremental slack update technique combined with the reformulated cost function produces the best placements on the purely combinational or lightly registered original MCNC netlists when the parameters ($\lambda = 0.1$, criticality exponent = 12) are used. The new placement approach was able to produce an average critical path delay 0.888x faster than the default VPR placer with a slightly better 0.981x average wire cost. Additional details of the placement results on the original MCNC netlists with the parameters ($\lambda = 0.1$, criticality exponent = 12) are shown in Table 5.8. As an aside, it should be noted that while some of the placements performed with the new incremental update approach failed to route, unlike VPR with frequent static timing analysis, this is likely not due to convergence problems caused by instability within the placer itself but simply because the λ factor was too high, guiding the annealing towards placements with slightly larger wire costs.

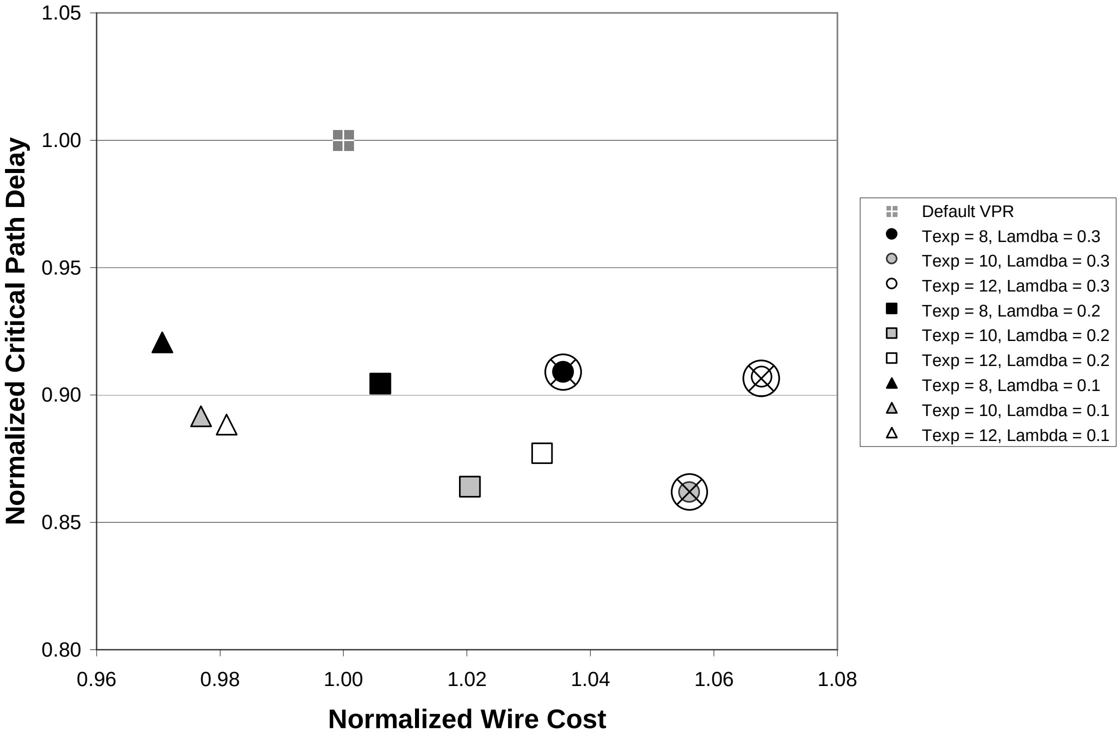


Figure 5.18: Incremental Criticality Update Placement λ and Criticality Exponent Tuning for Conventional MCNC Netlists

"X" denotes that a single netlist failed to route on the 1.2x minimum channel width architecture.
The wire and routed critical path delay shown exclude the failed netlist.

Table 5.6: Incremental Criticality Update Placement λ and Criticality Exponent Tuning for Conventional MCNC Netlists

	Combinational Circuits Only		Sequential Circuits Only		All Circuits	
Crit Exp , λ	**Norm. Wire Cost**	**Norm. Routed CPD**	**Norm. Wire Cost**	**Norm. Routed CPD**	**Norm. Wire Cost**	**Norm. Routed CPD**
8, 0.3	1.038	0.927	1.032*	0.885* (1)	1.036*	0.909* (1)
8, 0.2	1.004	0.952	1.008	0.859	1.006	0.904
8, 0.1	0.970	1.002	0.971	0.846	0.971	0.921
10, 0.3	1.058	0.906	1.054*	0.816* (1)	1.056*	0.862* (1)
10, 0.2	1.013	0.923	1.028	0.809	1.021	0.864
10, 0.1	0.976	0.952	0.977	0.835	0.977	0.892
12, 0.3	1.076	1.002	1.059*	0.813* (1)	1.068*	0.907* (1)
12, 0.2	1.022	0.951	1.043	0.809	1.032	0.877
12, 0.1	0.983	0.963	0.979	0.820	**0.981**	**0.888**
Default VPR	1.000	1.000	1.000	1.000	1.000	1.000
Best VPR w/ Frequent STA	0.985	0.924	0.968	0.825	0.977	0.873

* Indicates that some of the netlists failed to route on the 1.2x minimum channel width architecture. The number of failed netlists is indicated in parenthesis. The wire and routed critical path delay shown exclude the failed netlists.

Looking at the results in Figure 5.19 and Table 5.7, the new placement technique produces the best results on the heavily registered depth = 1 MCNC netlists when the parameters (λ = 0.05, criticality exponent = 8) are used. This produced 0.581x better post-routing critical path delay compared to default VPR placement

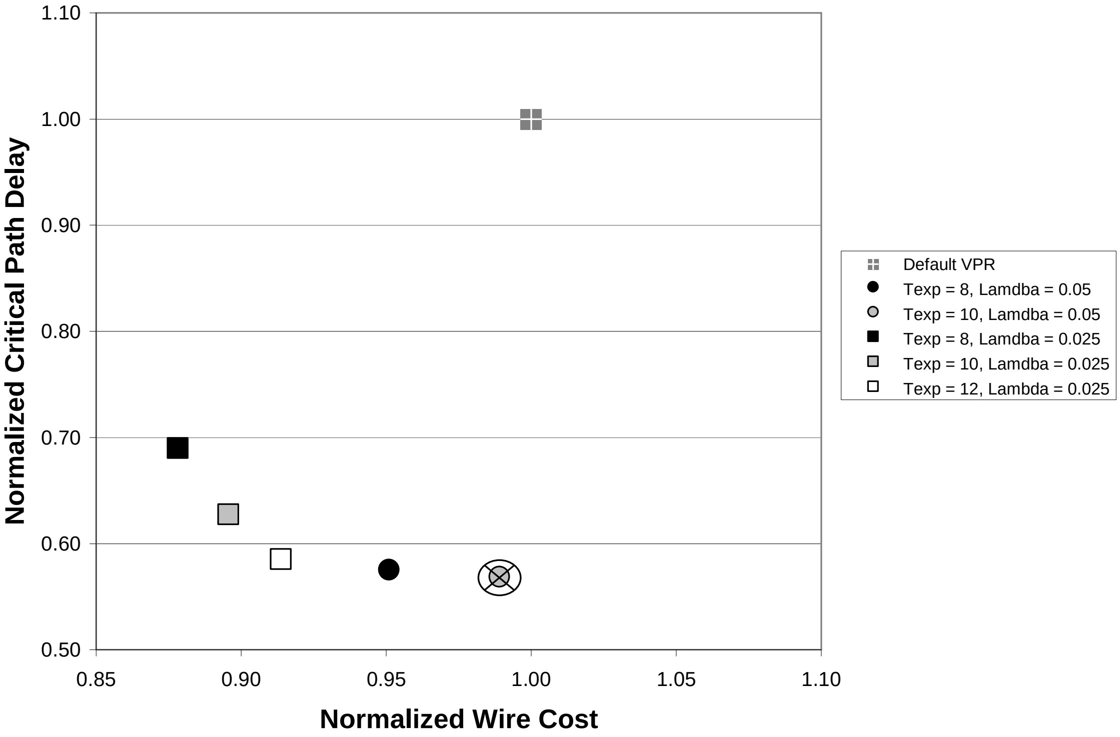


Figure 5.19: Incremental Criticality Update Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists

"X" denotes that a single netlist failed to route on the 1.2x minimum channel width architecture. The wire and routed critical path delay shown exclude the failed netlist. Results with more than one unroutable netlist are excluded entirely.

Table 5.7: Incremental Criticality Update Placement λ and Criticality Exponent Tuning for Depth = 1 MCNC Netlists

	Combinational Circuits Only		Sequential Circuits Only		All Circuits	
Crit Exp , λ	**Norm. Wire Cost**	**Norm. Routed CPD**	**Norm. Wire Cost**	**Norm. Routed CPD**	**Norm. Wire Cost**	**Norm. Routed CPD**
8, 0.1	1.021	0.604	1.134*	0.530* (3)	1.067*	0.572* (3)
8, 0.05	0.938	0.604	0.965	0.548	**0.951**	**0.576**
8, 0.025	0.894	0.709	0.862	0.672	0.878	0.690
10, 0.1	1.045	0.622	1.127*	0.490* (5)	1.073*	0.572* (5)
10, 0.05	0.963	0.611	1.019*	0.526* (1)	0.989*	0.569* (1)
10, 0.025	0.907	0.619	0.884	0.636	0.896	0.628
12, 0.1	1.071	0.592	1.115*	0.453* (6)	1.085*	0.548* (6)
12, 0.05	0.976	0.578	1.070*	0.461* (3)	1.017*	0.523* (3)
12, 0.025	0.916	0.612	0.912	0.560	0.914	0.585
Default VPR	1.000	1.000	1.000	1.000	1.000	1.000
Best VPR w/Frequent STA	0.952	0.628	1.018	0.607	0.984	0.618

* Indicates that some of the netlists failed to route on the 1.2x minimum channel width architecture. The number of failed netlists is indicated in parenthesis. The wire and routed critical path delay shown exclude the failed netlists.

with 0.951x better wire cost. Additional details of the placement results on the depth = 1 MCNC netlists with the parameters (λ = 0.05, criticality exponent = 8) are shown in Table 5.9.

Both of these results also compare favorably with the best results produced by VPR with frequent static timing analysis. Perhaps most easily seen in Figure 5.20, the results for the purely combinational or lightly registered original MCNC netlists only differ from the results produced by VPR with 10,000 static timing analysis runs per temperature iteration by a few percent. However, the new placement technique produces these results with several orders of magnitude less computation. This is because the new placement method only performs one static timing analysis per temperature iteration with extremely fast incremental updates in between. The depth = 1 netlists produce similar results. In this case, both placement approaches produce dramatically faster circuits, but the results obtained by the incremental criticality update technique are not only slightly better, but are also free of the runtime and stability issues associated with the more traditional placement approach with performed frequent static timing analysis.

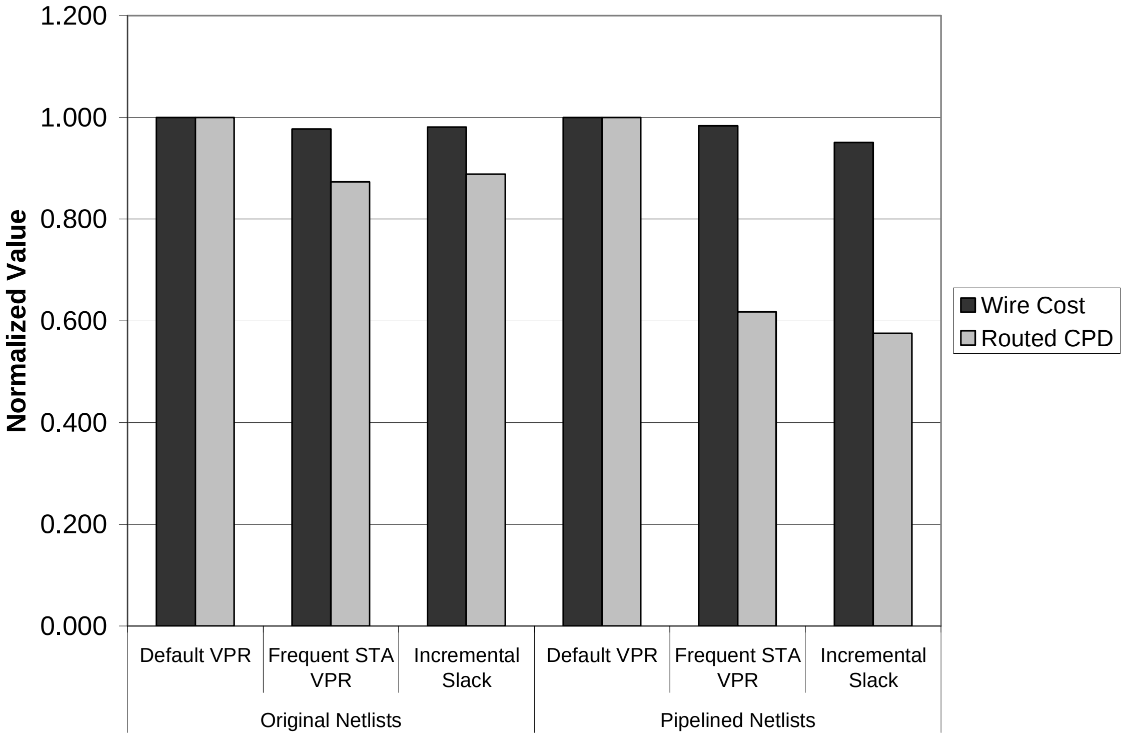


Figure 5.20: Comparison Between VPR and Incremental Criticality Update Placement

Table 5.8: Conventional MCNC Netlist Placement Comparison

	Default VPR **λ = 0.5, *CritExp* = 8.0**		**Frequent STA VPR** **λ = 0.3, *CritExp* = 12, 10K STA/Temp**				**Incremental Slack** **λ = 0.1, *CritExp* = 12**			
			Raw Values		**Norm. Values**		**Raw Values**		**Norm. Values**	
Netlist	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**
e64	30.21	3.12E-08	29.48	3.12E-08	0.976	1.001	29.81	3.18E-08	0.987	1.019
ex5p	178.17	6.75E-08	171.49	6.78E-08	0.963	1.005	169.60	6.99E-08	0.952	1.037
apex4	192.57	7.74E-08	189.14	7.49E-08	0.982	0.967	184.70	8.46E-08	0.959	1.093
misex3	199.39	7.34E-08	196.62	6.91E-08	0.986	0.942	194.78	6.75E-08	0.977	0.920
alu4	201.10	7.83E-08	199.62	7.72E-08	0.993	0.986	199.37	7.58E-08	0.991	0.969
des	249.48	9.12E-08	245.25	7.10E-08	0.983	0.778	258.01	7.16E-08	1.034	0.785
seq	259.92	7.90E-08	255.99	7.01E-08	0.985	0.887	254.12	8.12E-08	0.978	1.028
apex2	280.18	9.66E-08	274.22	8.37E-08	0.979	0.867	272.07	8.61E-08	0.971	0.892
spla	625.59	1.35E-07	634.98	1.48E-07	1.015	1.099	627.76	1.37E-07	1.003	1.019
pdc	934.04	1.49E-07	912.37	1.33E-07	0.977	0.892	916.11	1.54E-07	0.981	1.035
ex1010	678.37	1.81E-07	677.56	1.43E-07	0.999	0.791	663.71	1.52E-07	0.978	0.840
s1423	16.37	5.82E-08	15.95	5.93E-08	0.974	1.020	15.56	7.05E-08	0.950	1.213
tseng	102.62	5.53E-08	96.77	5.17E-08	0.943	0.936	95.51	5.58E-08	0.931	1.010
dsip	199.69	7.34E-08	193.36	5.39E-08	0.968	0.734	228.00	4.82E-08	1.142	0.656
diffeq	157.43	6.24E-08	149.72	6.47E-08	0.951	1.037	147.88	6.24E-08	0.939	1.001
bigkey	206.92	7.56E-08	204.73	5.27E-08	0.989	0.697	237.22	4.32E-08	1.146	0.572
s298	228.22	1.32E-07	217.23	1.28E-07	0.952	0.971	211.04	1.33E-07	0.925	1.009
frisc	584.86	1.62E-07	557.68	1.26E-07	0.954	0.780	536.85	1.29E-07	0.918	0.798
elliptic	502.36	1.11E-07	483.49	1.07E-07	0.962	0.964	465.58	9.55E-08	0.927	0.862
s38584.1	678.84	1.06E-07	673.69	7.32E-08	0.992	0.694	686.95	7.14E-08	1.012	0.677
s38417	693.47	1.02E-07	675.70	7.69E-08	0.974	0.751	663.02	8.10E-08	0.956	0.792
clma	1481.57	2.42E-07	1472.54	1.50E-07	0.994	0.622	1424.74	1.59E-07	0.962	0.658
Geometric Mean					**0.977**	**0.873**			**0.981**	**0.888**

Table 5.9: Depth = 1 MCNC Netlist Placement Comparison

	Default VPR **λ = 0.5, *CritExp* = 8.0**		**Frequent STA VPR** **λ = 0.3, *CritExp* = 8.0, 10K STA/Temp**				**Incremental Slack** **λ = 0.05, *CritExp* = 8**			
			Raw Values		**Norm. Values**		**Raw Values**		**Norm. Values**	
Netlist	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**	**Wire**	**CPD**
e64	44.35	1.99E-08	41.59	1.17E-08	0.938	0.589	41.79	1.12E-08	0.942	0.560
ex5p	224.83	2.65E-08	218.35	2.12E-08	0.971	0.799	216.26	1.64E-08	0.962	0.618
apex4	213.56	3.24E-08	208.40	2.13E-08	0.976	0.658	204.83	1.87E-08	0.959	0.577
misex3	269.73	3.53E-08	250.93	2.89E-08	0.930	0.817	242.11	2.30E-08	0.898	0.651
alu4	291.84	3.83E-08	256.71	3.12E-08	0.880	0.814	258.31	2.64E-08	0.885	0.688
des	352.68	4.65E-08	345.94	2.43E-08	0.981	0.522	347.54	2.05E-08	0.985	0.440
seq	355.04	4.49E-08	331.02	2.42E-08	0.932	0.539	332.32	2.36E-08	0.936	0.525
apex2	407.76	4.03E-08	372.87	2.43E-08	0.914	0.603	369.61	2.43E-08	0.906	0.602
spla	846.56	5.33E-08	827.47	3.13E-08	0.977	0.587	806.31	6.08E-08	0.952	1.142
pdc	1185.60	7.69E-08	1175.18	3.48E-08	0.991	0.453	1125.07	3.30E-08	0.949	0.429
ex1010	876.20	5.40E-08	866.71	3.51E-08	0.989	0.649	825.82	3.46E-08	0.943	0.641
s1423	75.38	2.24E-08	73.87	1.67E-08	0.980	0.745	68.83	9.90E-09	0.913	0.442
tseng	308.53	4.55E-08	326.61	2.61E-08	1.059	0.573	318.98	2.64E-08	1.034	0.579
dsip	259.39	4.31E-08	227.70	2.90E-08	0.878	0.672	235.66	2.58E-08	0.909	0.598
diffeq	485.70	5.38E-08	507.69	2.67E-08	1.045	0.496	492.11	2.66E-08	1.013	0.495
bigkey	269.51	4.70E-08	258.60	2.78E-08	0.960	0.590	254.68	3.30E-08	0.945	0.701
s298	456.04	4.85E-08	442.93	3.60E-08	0.971	0.742	438.58	2.70E-08	0.962	0.556
frisc	1427.26	7.17E-08	1554.43	2.79E-08	1.089	0.388	1454.20	2.78E-08	1.019	0.388
elliptic	1430.86	8.41E-08	1453.92	4.52E-08	1.016	0.537	1395.34	4.76E-08	0.975	0.566
s38584.1	1721.81	1.19E-07	1812.62	9.16E-08	1.053	0.769	1617.91	8.12E-08	0.940	0.681
s38417	1976.38	7.21E-08	2399.60	4.74E-08	1.214	0.657	2006.69	3.40E-08	1.015	0.472
clma	2414.58	9.42E-08	2329.40	5.91E-08	0.965	0.627	2167.83	6.11E-08	0.898	0.649
Geometric Mean					**0.984**	**0.618**			**0.951**	**0.576**

5.8: Conclusions and Future Research

This chapter identified a longstanding but relatively poorly understood problem surrounding FPGA placement. Although previous research has shown that timing-driven placement can improve critical path delay for conventional netlists, existing methodologies have a fundamental shortcoming. Specifically, classical placement relies solely on the link criticality information provided by static timing analysis. However, static timing analysis is too computationally expensive to perform very often, so the bulk of the optimizations performed by conventional annealing is done with stale and potentially very inaccurate timing information.

Although this can be mitigated somewhat by simply running static timing analysis more often, not only does this dramatically increase the computational requirements of the placement tool, it does not truly address the larger scale problem. Inherently, the placement tool must be able to accurately evaluate the change in timing considerations before and after each annealing move. While increasing the amount of static timing analysis can improve the wider-scope accuracy of the timing information, conventional placement approaches still use old timing information on a move-by-move basis. Although subtle, this approach makes the intrinsic assumption that the critically of any given connection in the system will not change very quickly. However, as demonstrated, this is clearly not true, even for simple registered circuits. This very basic incorrect assumption can cause the system to prefer degenerate solutions. While this limits the potential benefits of more accurate timing information, more seriously it can open the door for oscillations during the placement of heavily registered applications. These oscillations can results in severe convergence problems that destroy the basic functionality of the placement tool. Oddly enough, this is an issue that can also plague timing-driven routers and this concept will be revisited in Chapter 7 during the discussion of register-aware routing.

This chapter suggests two modifications to the classic timing-driven placement approach that address these issues. First, the accuracy of timing information can be maintained very efficiently by applying incremental changes during placement. While this approach cannot guarantee completely accurate criticality information, this fundamental difference enables the system to evaluate the timing situation of a placement on a per-move basis. This new capability naturally leads to a change in the cost function. Degenerate solutions and the accompanying oscillations can be avoided by reflecting potential changes made to link criticality in the cost of a move. This new approach produces much higher quality placements without significant affecting the computational requirements. For conventional combinational or lightly registered netlists it produce placements that are on average 0.888x faster in terms critical path delay with no degradation in routability. For heavily registered netlists it generated placements that are 0.581x faster with 0.951x better wirelength.

While this approach dramatically improves placement quality, this is not to say that this topic has necessarily been fully explored. Because FPGAs have fixed, finite resources and because the placement tool directly affects the interconnect characteristics of the final system, placement is often a lynchpin in the CAD process. While the next chapter will discuss many of the timing-related issues concerning how the placement tool interacts with earlier parts of the netlist compilation flow, there are still open questions regarding how accurate timing information affects the system within the placement tool itself.

For example, while it is obvious that updating link criticality before the cost of a new placement is evaluated is important, moderately registered netlists and applications with a low logic depth present a unique opportunity for a different approach. One primary problem that has been discussed is that static timing analysis of the entire circuit is impractical to perform frequently during annealing. However, it is possible to perform an incremental static timing analysis after each annealing move. As discussed earlier, determining the new critical path of the system is not essential. Rather, determining the change in relative criticality is much more important. Thus, it is possible to propagate changes in the delay on a moved block forwards and backwards through the circuit in some limited fashion without evaluating the entire netlist. These changes would only have to spread along the fan-in and fan-out cone of the moved block until they reached a register or an I/O pin.

Of course, one reason that the incremental timing update technique described above performs so well is that it is very fast and its error for heavily registered circuits is extremely low. However, circuits with a moderate number of registers and applications with a small number of logic blocks along the deepest path are particularly amenable to updating with a limited static timing analysis. This is because the number of logic blocks that would need to be updated after a move is relatively small. Thus, it is possible that a limited, but incremental static timing analysis can be performed very quickly and could provide even better accuracy. Incorporating such a technique into the placer could lead to even better results.

Furthermore, this chapter has focused on simulated annealing-based placement. While the basic issues addressed in this chapter are important for virtually all placement algorithms, actually applying these techniques and the impact they will have is not necessarily clear for other placement techniques. It is generally accepted that although simulated annealing produces good results, it generally comes at the cost of a large runtime. Thus, while the discussion of computational requirement in this chapter is particularly relevant, many commercial systems to handle very large circuits often avoid simulated annealing as much as possible. These types of tools use a two-stage placement process in which a faster, but less accurate approach is first used to obtain a *global placement*. This type of tool takes the place of the early high temperature annealing to determine the large-scale orientation of the blocks and leaves a much simpler *detailed placement* problem for a following annealing-based placer. In this case, since only smaller optimizations need to be made, the annealing is generally started at a much lower temperature. This, of course, leads to a much shorter runtime.

While the problem of inaccurate timing information would seem to be a problem for any iterative placement algorithm, the challenges that such a two-phase system faces may be different. First, global placement tools such as quadratic placement [18] or forced-directed placement [5] have dramatically different techniques to incorporate timing information during the placement process [30]. This in itself poses a problem because it is not obvious how these tools might integrate more up-to-date timing information. However, the problem even changes somewhat within the secondary annealing-based placement phase. Because the larger structure of the placement has already been determined by the global placer, the optimizations options that are available to the annealer are much more limited and any improvements must be done much more quickly. While the basic issue of annealing with stale timing information still stands, it would be interesting to measure the effect the suggested improvements can have in such a different placement situation.

Chapter 6: Register-Aware Placement

The enhanced timing-driven simulated annealing algorithm described in Chapter 5 showed the benefits of using more accurate timing information during placement. However, in some sense there is still an inherent limit to the performance benefits that can be achieved because of systemic problems in the basic toolflow itself. As discussed in Chapter 4, early portions of the netlist compilation tool chain, such as logic synthesis and packing, define the netlist that following tools, such as placement and routing, work with. However, these early tools must make design choices with very little information about the interconnect characteristics of the final implementation. Since the traditional toolflow is purely feed-forward, conventional placement and routing tools have no opportunity to fix these errors, even once this information is known.

This chapter will describe some of the basic limitations that applications developers can encounter with the traditional feed-forward CAD toolflow. The discussion will further focus on how registers in a netlist can make these problems worse. This will lead to a summary of existing attempts to address this problem and the introduction of a new technique for placement that incorporates aspects of *physical synthesis*. Physical synthesis optimizations change parts of the netlist based upon information that can only be obtained late in the netlist compilation process. By allowing the placement tool to modify a netlist during placement, the system is able to significantly improve both wire cost and critical path delay.

6.1: Feed-Forward Design Flow – Implications for Packing, Retiming and Placement

Similar to writing high-performance software, developing applications for an FPGA is generally a very iterative process. Until the HDL code is compiled to a routed netlist, it is very difficult to determine the performance or area requirements of an application. First, the logical requirements of a netlist cannot be accurately measured until the application has been through synthesis, technology mapping, and packing. However, even at this point the packed netlist only serves as a lower bound on the necessary FPGA size and an upper bound on the achievable clock frequency.

This is only a lower bound on the FPGA area because the subsequent placement and routing may require a larger fabric to provide sufficient communication resources to connect all the logic blocks together. This is because some applications may have many signals that need to traverse a specific area of the chip. If the number of signals exceeds the communication capacity of that area, the netlist needs to be mapped to a larger FPGA so that the logic blocks in congested regions can be spread out, distributing traffic over more routing channels.

Along the same lines, this is only an upper bound on the clock frequency of the design because while the delay through the necessary logic can be determined, this only represents a portion of the overall delay in the final circuit. The majority of the delay in a modern FPGA is accumulated in the programmable interconnect. Since the precise path a signal will take is not determined until after routing is completed, it is very difficult to determine a large portion of a net's timing requirement. All of these factors combine and FPGA application developers must generally go through multiple iterations from HDL code to routed circuit to meet performance or device area specifications.

While the long engineering and debug cycle of FPGA application design can complicate the development of high performance circuits, in some sense the tool chain itself is somewhat constrained by its highly compartmentalized and feed-forward nature. For example, as discussed in Section 4.2, conventional CAD tools group registers and logic together during the packing process. However, this limits the optimizations that the placement tool can perform since it can only move entire logic blocks around. While this probably does not create a concern for conventional netlists, the large number of registers in heavily pipelined, C-slowed and retimed applications can cause problems. This is because packing algorithms such as T-VPack [1] implicitly assume that flip-flops will be driven by a LUT and the two should be packed into the same CLB whenever possible. While this approach is likely sufficient if the number of registers in the circuit is relatively low, heavily registered netlists will likely have signals with many flip-flops.

These multi-register connections create two problems. Consider the example in Figure 6.1. If this circuit is mapped to an architecture that has two LUTs and 2 flip-flops per CLB, the packing tool will wrap *LUT A* and two of its following flip-flops into a single atomic unit before placement. This greatly limits the potential for the placer to use these registers to mitigate interconnect delay if the LUT's output signal requires a long wire and ends up being timing critical. Furthermore, packing can fuse unrelated logic blocks and flip-flops together. The third register on the output of *LUT A* cannot fit into the same CLB as its source, so it will be arbitrarily combined with some other logic block before placement. Not only does this limit the placer's ability to use registers to distribute interconnect delay, this artificially ties unrelated parts of the circuit together, making the placement problem more difficult.

Furthermore, following the conventional toolflow, operations that can restructure the netlist, such as retiming, must be performed prior to packing. Unfortunately, since packing is performed before placement, this general approach can encounter problems. First, the retiming may not be very effective. Without any

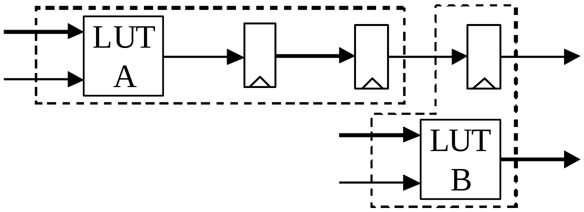


Figure 6.1: Packing Implications for Heavily Registered Netlists

placement information the retiming tool can only very roughly estimate interconnect delay. In general, it must retime using a simple unit delay model for logic blocks and largely ignore the potentially significant delay accumulated in the interconnect.

On the other hand, once the circuit has gone all the way through the entire CAD toolflow, if the resulting implementation does not meet timing specifications an application developer might attempt to repeat retiming on the original netlist for another run of packing, placement and routing. However, it is unclear how useful it might be to try and forward timing information from a previous placement and routing back to the retimer for another iteration of the CAD tools. This is because there is no guarantee that this information would be accurate or relevant to the new implementation – the placement may change considerably in the meantime. Nets that were timing critical in an earlier placement may not remain so. This holds true even if the netlist were not changed at all but simply re-placed. Thus, a subsequent retiming may actually degrade the performance of the circuit instead of improving it. This is referred to as a problem with *timing closure*.

6.2: Previous Retiming-Aware Approaches

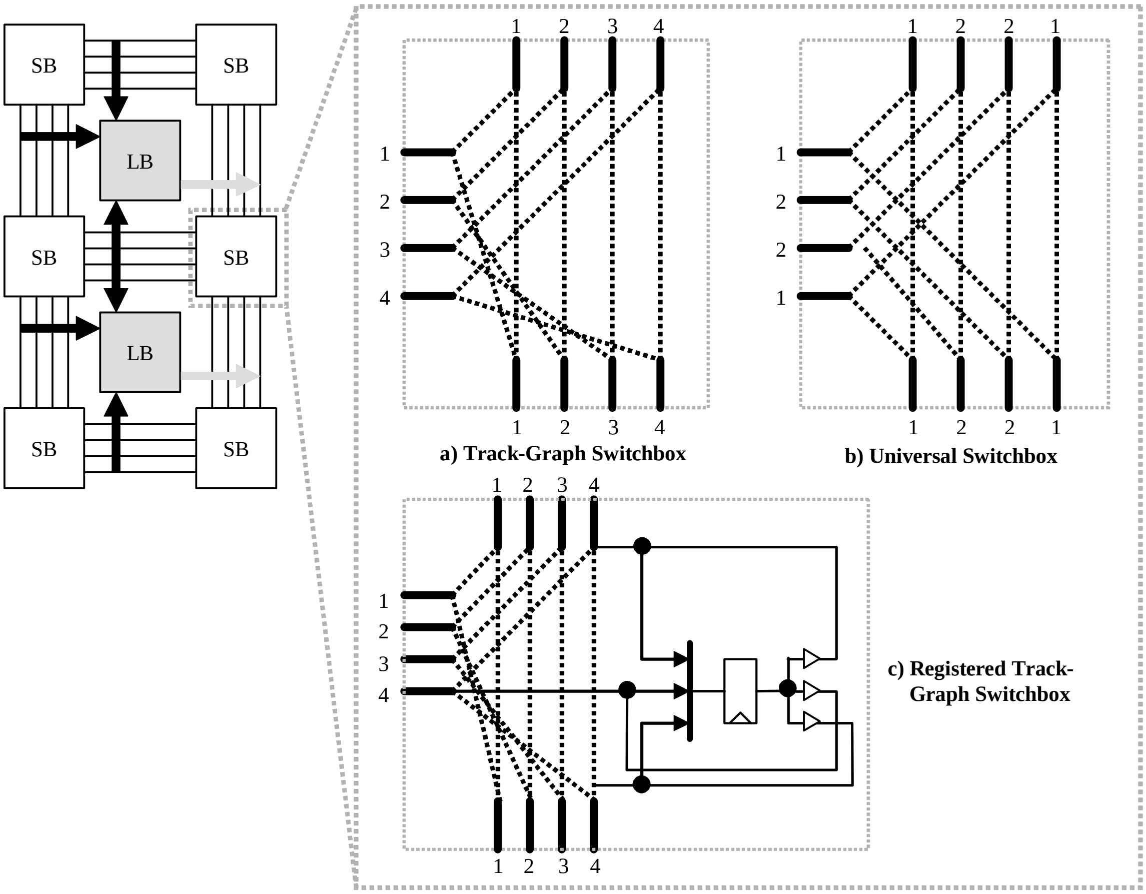
Since the precise delay of each net cannot be known until the later stages of the CAD process, multiple research groups have taken steps towards applying retiming after placement or routing. These efforts can be split into two general categories. The first devises specific architectures that are particularly amenable to absorbing the registers generated by retiming. In these systems, allocating new registers is easy due to the unique characteristics of the underlying hardware. Thus, retiming can be applied after routing without changing the existing paths. The second general approach relies on sophisticated CAD tools that incorporate the new registers caused by retiming into an existing placed netlist. Although the precise delay of each net cannot be known for certain until after routing due to congestion concerns, as discussed in Chapter 5, the placement can generally give a relatively accurate idea of signal criticality. Thus, while retiming can be applied with much more precision, the challenge that these tools face is merging the registers generated by retiming into the existing placement without changing the larger-scale characteristics.

Unfortunately, in some sense all of these previous approaches still struggle with the same basic problems of the conventional approach. That is, late in the toolflow it is much safer to apply retiming very conservatively. However, this also makes the potential benefits quite limited. On the other hand, if retiming is applied very aggressively, the new registers introduced into the system can overwhelm the register resources that are available and cause a dramatic or unpredictable change to the existing placement.

6.2.1: Previous Architectural Retiming Solutions

Perhaps the most straightforward manner to deal with the problems associated with retiming is to modify the architecture itself to allow retiming to be performed after placement and routing, without disturbing the existing configuration. In this way, the system can sidestep any problems with timing closure.

For example, the system suggested in [38] is a registered track-graph FPGA. A track-graph FPGA is unique because the entire communication network is split into completely separate, but overlaid routing domains. If the switchbox architecture in Figure 6.2a is used, once a signal is routed onto a given wire, all of the other wires it can connect to are located in the same relative position in their respective routing channels. Stated another way, all of the routing domain N wires are connected together, with no cross-connections to the wires in other routing domains. In Figure 6.2a, a signal that enters the switchbox on the first track from the left can only reach the first track in the routing channels exiting the top and bottom. Although for clarity only an edge case is shown, the same segregated connectivity is maintained throughout the rest of the FPGA. Thus, this architecture will have 4 completely separate sets of communication



a) Track-Graph Switchbox

b) Universal Switchbox

c) Registered Track-Graph Switchbox

Figure 6.2: Track-Graph, Universal and Registered Track-Graph Switchboxes

resources that are merely sitting side-by-side. In contrast, if the universal switchbox architecture in Figure 6.2b is used, signals can connect to wires with different relative positions in their routing channel. A signal that enters the switchbox on the first track from the left can reach multiple tracks exiting the top and bottom. For that matter, the signal can even return out the left side on the bottom-most track.

While the differences between these two routing architectures is somewhat subtle, as discussed in [44], more flexible switchboxes such as the universal design tend to improve the routability of the FPGA as a whole. However, from a CAD standpoint track-graph architectures are attractive because signals on one routing domain cannot interfere with signals on another. It is exactly this characteristic that the authors of [38] exploit to incorporate specialized retiming registers.

The authors of [38] replace the conventional track-graph switchboxes with registered switchboxes like the one shown in Figure 6.2c. In this case, the connections that use routing domain *4* have the option to enter a register at each switchbox. The results in [38] suggest architectures should replace approximately 25-50% of the routing domains with registered connections. The toolflow for this system encourages potentially timing-constrained connections to use registered track domains. It begins with conventional timing-driven placement and routing, ignoring the registers embedded in the interconnect. At this point, timing-critical links in the routed configuration are identified and singled out. If they are not already connected via a wire domain that is outfitted with optional registers, the connection is swapped to an equivalent wire domain that does. At this point, a restricted retiming algorithm is applied. Instead of performing true Leiserson/Saxe retiming, this approach limits the number of registers that can be pushed onto a specific connection to the number of optional retiming registers that already exist along the current route.

Unfortunately, while this is a simple solution, this greatly limits the optimizations available to the retimer. First, the retimer is specifically limited to only using the specialized retiming-specific registers added to the interconnect structure that are along the existing route. This makes efficiently using the registers in the system very difficult. For instance, this approach does not consider using the potentially large number of registers in switchboxes or logic blocks that are adjacent to, but not directly along, a given path because this would require changing the routing. Furthermore, the system completely segregates flip-flops present in the original netlist and registers created by retiming. Flip-flops within the CLB can only be used by registers in the original netlist and flip-flops embedded in the interconnect can only be used by registers moved by retiming. This can lead to fragmentation between the two essentially identical resources. The strict division in the CAD tools means that both a heavily registered netlist that does not require retiming or a relatively lightly registered netlist that requires extensive retiming will be unable to use all of the available registers in the system.

This problem with register efficiency leads to an even larger issue. This approach makes the amount of retiming that the architecture can support heavily affected by the number of additional retiming registers put into the system. Thus, providing sufficient resources to support applications with a lot of retimed registers is very expensive. In some sense this merely pushes the design paradox associated with retiming from being a problem for the CAD tool to being a problem for the FPGA architect. Adding too many retiming registers makes the overhead for the architecture very large. However, adding too few artificially limits the options for aggressive retiming. Of course, all of these issues are also on top of the more fundamental problem that this type of approach only works on a very specific and specialized registered interconnect structure.

6.2.2: Previous CAD Retiming Solutions

Efficiently supporting more general registering resources requires new CAD tools. Towards this goal, there have been a number of research projects that have attempted to perform retiming after placement. These approaches generally use multiple stages of processing, with a specialized placement tool followed by a retiming phase.

The work in [5] was among the first efforts to address retiming as a placement problem. Although this work actually involved *floorplanning*, a precursor to placement that can be thought of as a very rough global placement, it laid the groundwork for the work in [7]. The authors of [7] clearly define a three-stage approach for retiming-aware placement. This technique first borrows a cue from classical timing-driven placement by incorporating static timing analysis with a modified simulated annealing cost function to identify potentially critical nets. It uses this information to keep these links as short as possible. When the annealing is complete, they perform a classical retiming step to improve delay. This is followed by a short simulated annealing process to re-distribute registers that are created or deleted and keep the logic blocks relatively even in size. Unfortunately, this work targeted an ASIC development flow. Since ASICs create completely custom chips, the CAD tools are able to largely create or delete resources at will. Since FPGAs must use the finite resources offered by a specific architecture, there are strict limitations as to where the system can and cannot create a register.

These FPGA-specific concerns were addressed in works such as [31], [43] and [37]. [31] suggested a very straightforward solution in which conventional placement is followed by a constrained retiming step. Similar to the architectural solution described earlier, the retimer can only push a limited number of registers onto a specific link. In this case, the retimer could choose to either use or not use the flip-flops present in the BLEs already allocated by the placement phase. Again, while this is a simple and closed-form solution, like the approach in [38] this technique greatly limits the optimization available to the retimer since this does not allow the system to incorporate the resources that might exist in neighboring unoccupied BLEs.

In contrast, [43] explored the opposite end of the retiming problem. In this approach, the authors still begin with a good timing-driven placement, but then retiming is performed without any restrictions on the number of registers that can be placed on a given link. Although they include an algorithm to associate as many registers as possible with their host LUT to maximize the use of the flip flops in the same BLE, additional registers are allocated by simply searching in a spiral pattern for the closest unused register. Thus, this technique offers no guarantee of timing convergence since the retimer can create an unlimited number of registers in potentially very sensitive areas of the array, with no good way of cleaning up the placement.

The approach discussed in [37] was the first technique that truly attempted to address the basic problem between balancing potential retiming improvements and issues with timing closure. Their approach follows the work in [7] relatively closely with a three-stage retiming-aware placement process. They first use a modified simulated annealing cost function to identify timing-sensitive nets, specifically targeting feedback loops and the relationship between critical paths and near-critical paths. This is followed by a heuristic retiming step. Primarily, this retimer tries to move registers in the netlist, keeping in mind CLB *legalization* issues. CLB legalization is a problem because this work focused on architectures with logic blocks that do not have full input and output connectivity, like those in Figure 6.3a. Retiming creates new registers that need to be integrated into the rest of the netlist. Thus, this disturbs the original packing of the netlist. This change in packing makes it possible that certain CLBs may not have enough input or output pins to accommodate the new contents. The authors of [37] attempt to retime while minimizing this impact by estimating the cost associated with each potential retiming move. They identified three possible situations in which they could insert a register into a net. In order of preference, these cases are:

1) Where a register is pushed onto a net very close to the output of the LUT and the entire net uses the registered result. In this case, the LUT and flip-flop can share a BLE.
2) Where a register is inserted somewhere between the output of a BLE and some of the sinks. In this case they require an additional BLE, either because the flip-flop associated with the source LUT is already used or because the net requires access to both the pipelined and unpipelined LUT output.
3) Where a register is pushed onto a net very close to one specific sink. In this case not only is an additional BLE needed, this register is also only closely associated with one specific logic block.

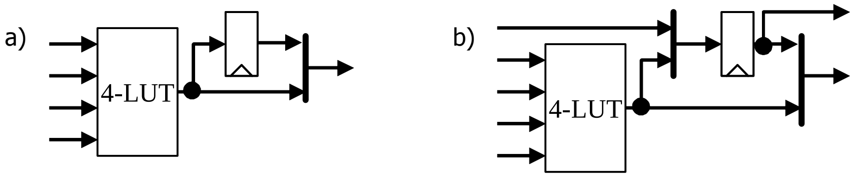


Figure 6.3: Non-Independently and Independently Accessible Flip-Flop Architectures

Similar to [7], this FPGA-centric retiming step is followed by a very short iterative legalization phase. This step is primarily concerned with resolving any illegal CLBs created by the retiming process. Of course, because the retiming phase preferentially creates registers that can be easily absorbed by the source BLE, the tool generates relatively few registers that require new BLEs. This lowers the demands on the legalization phase.

Unfortunately, this approach still has two issues. First, much of their work focuses on solving architecture-specific CLB input and output legalization problems. However, this is not necessarily a concern for modern devices. Recent FPGAs such as the Virtex II [45] do not require cluster legalization. This is because they not only provide independent access to LUTs and flip-flops (Figure 6.3b), they offer full CLB input and output connectivity. For example, if there are eight 4-LUTs and eight flip-flops in a CLB, the logic blocks will have the capability to take 40 independent inputs (8 x 4 LUT inputs + 8 flip-flop inputs) and produce 16 independent outputs (8 LUT outputs + 8 flip-flop outputs).

More importantly, this methodology still may not produce feasible or convergent placements. This is because the retiming is still wholly decoupled from the legalization phase. This means that the retimer may produce a netlist that requires registers in an area that currently does not have any available in the existing placement. At this point, the post-processing step has to choose between producing an illegal placement or risk disrupting the timing of the system. This type of situation is particularly likely given netlists with a large number of registers, since, by the very nature of the netlist itself, there might be relatively few empty register locations in the array and many of the nets may be critical or nearly critical. Thus, the retimer must be tuned very conservatively to specifically avoid these kinds of circumstances.

Taking a step back, perhaps it is a better idea to consider the source of these problems. All of the complications regarding retiming stem from the fundamental approach that has been used. The problem is that retiming cannot be performed as an isolated, single-shot optimization step if the system is to retime as aggressively as possible while still maintaining the original placement that provided the timing information. Essentially, all of the approaches discussed so far are still fundamentally patchwork tools in that they rely on completely distinct placement and retiming phases. To obtain the best results from heavily registered netlists, it is likely that retiming needs to be considered in a more holistic sense. In other words, retiming needs to be a far more integral part of the placement process itself. This philosophy would allow the system to apply retiming more effectively and predictably.

The work in [36] is the most encouraging work to date on FPGA retiming because it provides the most unified placement and retiming approach. This technique begins much like the work in [37] with a relatively standard timing-driven placement. However, after annealing, the placement is given to an iterative incremental retiming and placement tool. Here, instead of performing a single traditional retiming run, such as Leiserson/Saxe, followed by a single legalization phase, the tool alternates multiple times between very short, incremental retiming steps and CLB legalization phases. The retiming tool is incremental because it does not try to solve the timing problems of the entire circuit at once. A systemic retiming can potentially move all of the registers in the system through multiple levels of logic in a single step. Of course, such a drastic change to the netlist will entirely disrupt the existing placement. Rather, this tool simply examines the effect of gradually pushing a register through one level of logic at a time. It examines each of the registers in the netlist once in turn to see if its input or output net is critical or near critical. If the output net is critical, the retimer attempts to push the register forwards through the logic blocks directly driven by its output net. Conversely, if the input net is critical, the retimer attempts to pull the register backwards through the logic block that drives its input net. Of course, this type of incremental retiming limits the scope of the improvements that can be made in a single step, but the retimer will have multiple chances to further improve the system.

Each of these comparatively gentle retiming phases is followed by a greedy legalization phase. Again, the primary goal of this tool is to eliminate the overuse of CLB input and output pins. This legalization tool is referred to as greedy because while it attempts random swaps like simulated annealing, unlike annealing it only considers making moves that reduce the total number of illegal CLBs. If the placement remains illegal after a relatively small number of attempts, the new retiming is considered un-placeable and the system reverts to the previous netlist. This step-wise retiming and legalization process is repeated until no more improvements are made to the circuit’s critical path delay.

6.3: Integrated Placement and Physical Synthesis

While the approach described in [36] integrates retiming into the placement process far more than previous tools, it still uses a somewhat artificially segmented technique. For example, although it begins with standard simulated annealing for placement, it uses a solely greedy post-retiming legalization phase to integrate new registers into the existing placement. This shift in placement approaches seems largely unnecessary. Since simulated annealing provides such a powerful optimization framework, it is ideal for merging new registers into the system gracefully. For that matter, while the approach in [36] remains overwhelmingly preoccupied with CLB legalization, in some sense the basic philosophy that it uses does not make this a priority. When retiming, additional registers may need to be created. However, rather than retiming first, putting new registers into CLBs that make the placement illegal and then trying to fix the problem later, it is likely safer to only retime the system when it is certain that reasonable legal locations are available. This allows the natural optimization characteristics of the simulated annealing process to migrate registers into the proper locations without introducing additional legalization worries. Furthermore, this approach only considers retiming after annealing has finished. It is entirely possible that retiming a particular register is a good idea, but to see this requires larger-scale changes to the system that can only occur during simulated annealing. Finally, there is also the matter of CLB packing. The approach in [36] does not address the two tendencies discussed in Section 6.1 in which the packing tool unnaturally combines multiple registers into a single CLB or fuses unrelated logic and registers together. Dealing with this problem during placement is critical to producing good placements for heavily registered applications.

This section introduces a new technique that addresses all of these concerns by performing simultaneous simulated annealing-based placement and physical re-synthesis. This approach begins by first incorporating both traditional CLB-level moves and *FF-level* moves into the conventional placement framework. FF-level placement moves give the annealer the ability to migrate individual registers separately from the rest of their host CLB. This allows the placement tool to change the packing of registers and more effectively use them to distribute interconnect delay. This approach continues by integrating retiming into placement. Although similar to the work in [36], the technique presented here merges retiming moves more smoothly into placement by treating them as much as possible like conventional placement moves. Essentially, retiming moves are accepted or rejected by the same temperature/cost/benefit structure as normal logic block swaps. This level of integration allows the retiming to more fully leverage the power of simulated annealing placement.

6.3.1: Packing and FF-Level Placement

While packing reduces the problem size presented to the annealer, in some sense it also interferes with the optimizations that are made during placement. As discusses earlier, this is because packing locks registers into specific logic blocks early in the compilation process. However, dealing with this problem is not necessarily as simple as reverting to placement at the individual LUT and flip-flop level. This is because such an approach raises several serious concerns. While this has obvious dramatic implications for the annealing runtime, it can also lead to problems simply finding high quality placements. For the majority of registers it makes sense for a LUT and its companion flip-flop to reside in the same CLB. Specifically, this configuration is special because the connection between the LUT and flip-flop does not incur the delay or potential wiring congestion associated with exiting a CLB, traveling along shared interconnect wires, and re-entering another CLB. However, if the placement tool is only able to move LUTs and flip-flops independently from one another, it makes it very easy for a LUT and flip-flop to separate, but much more difficult for them to reunite.

Consider the two possible states that a LUT and flip-flop can be in (together in the same CLB or apart in different CLBs), shown in Figure 6.4. If the LUT and flip-flop are initially together within a 5x5 grid of CLBs, all possible moves of either the LUT or flip-flop will break them apart. However, once in this state, only two in the 48 possible moves (24 possible new locations for the LUT and 24 possible new locations for the flip-flop) will bring them back together. Furthermore, once they do reunite, after the annealing cools past a certain critical temperature it is unlikely that the placement tool will be able to move the LUT/flip-flop pair to any other CLB location. This is because moving both would require the placement tool to first separate them, with some comparatively high cost, before reuniting them in the new CLB. This will cause the placement tool to artificially stall out relatively early in the annealing process because it is unable to make further improvements.

This problem can be addressed by adding a hybrid CLB-level/FF-level move function to the basic placement tool. Since the incremental timing update placement approach from Chapter 5 produces such good results, this is an obvious platform to begin with. As seen in lines 4, 12 and 18 of Figure 6.5, this technique requires four new placement parameters: a FF-level placement *activation point*, *criticality threshold*, *separation probability*, and *homing probability*. These parameters allow this technique to compensate for the problems associated with FF-level placement.

The FF-level placement activation point determines when the system turns on the capability to move registers in the netlist separately from their host CLBs. Since the annealing begins with an arbitrary initial placement, the early portion of the annealing process is primarily devoted to simply roughing out the large-scale structure of the netlist. It is likely that moving registers separately during these early stages is not necessary or desirable since moving entire CLBs allows the system to settle down more rapidly. As seen in lines 4-6 of Figure 6.5, this technique uses a built-in feature of the placement tool to determine how far the overall annealing has progressed - the range limit window. The FF-level placement activation point is simply some fraction of the maximum annealing window size. It could vary between the 1.0, beginning

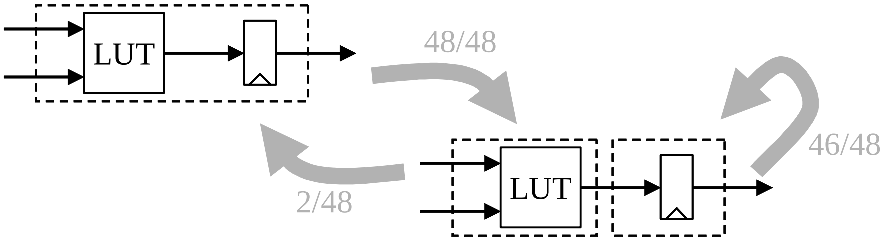


Figure 6.4: Probability of LUT and Flip-Flop Separation Versus Reunion

```
Placement with CLB and FF-Level Moves
0   randomly place logic blocks onto architecture
1   determine initial temperature
2   while(!done)
3       for i = 0 to numAnnealMovesPerTemp
4           if range limit window <= (FF-level activation point * max window size)
5               activate FF-level placement
6           end if
7           select random LUT or FF in netlist
8           if (selected LUT || !FF-level placement active)
9               swap entire CLB contents with random CLB
10          else
11              if FF in same CLB as source
12                  if (FF max link criticality >= FF-level placement criticality thresehold) &&
                                        (rand <= FF-level placement separation probability)
13                      swap FF with random FF in move window
14                  else
15                      swap entire CLB contents with random CLB in move window
16                  end if
17              else
18                  if rand <= FF-level placement homing probability
19                      swap FF with a FF in source CLB
20                  else
21                      swap FF with random FF in move window
22                  end if
23              end if
24          end if
25          accept or reject move(ΔCost, currTemp)
26      end for
27      update critical path delay
28      update currTemp
29      update range limit window
30      evaluate exit criteria
31  end while
```

Figure 6.5: Pseudo-Code for Incorporating FF-Level Placement Moves

FF-level moves right from the start of placement, to 0.0, beginning FF-level moves very late in the annealing process.

As seen in lines 7-10 of Figure 6.5, the move selection of this approach begins by selecting a random LUT or flip-flop in the netlist. If a LUT is selected, or if FF-level placement has not been turned on yet, the entire contents of the host CLB is swapped with another random CLB within the movement window. However, if a flip-flop is selected and FF-level placement has been activated, the system performs several tests to determine what to do next.

As seen in lines 11-16, if the register is in the same CLB as its source, the placement tool checks the criticality of the nets to which it is connected. Ostensibly, this approach would like to disturb the original packing only when it senses that the current arrangement is limiting the options of the placement tool to use a register to evenly distribute delay. Thus, the placement tool only has the potential to perform a flip-flop level move to separate the register from its host CLB if the register is connected to a net that has a criticality equal to or larger than the FF-level placement criticality threshold. If the register is along a highly critical path, the probability of performing the separation is controlled by the FF-level placement separation probability. If either the net criticality or separation probability checks fail, a conventional CLB-level move is performed.

On the other hand, as seen in lines 17-22, if a flip-flop is selected that is not in the same CLB as its source, the placement tool has the potential to explicitly reunite the two. This probability is controlled by the FF-level placement homing probability. Again, although packing a register into the same CLB as its source is generally advantageous, once the two have been separated it is comparatively hard for them to find each other again. The homing probability factor can increase the likelihood that the register will return to the same CLB as its source. However, if the system does not elect to return the register to the same CLB as its source, it simply swaps with some other flip-flop within the movement window.

6.3.2: Retiming

As mentioned earlier, although Leiserson/Saxe retiming has some unique optimality characteristics, it is likely that the key to better overall results is a more incremental approach that can be better integrated into the placement process. Thus, the second part of this new simulated annealing-based physical re-synthesis approach borrows many concepts from the technique in [36]. However, this new approach leverages the inherent optimization aspects of simulated annealing and applies it to retiming. Here, conditional incremental retiming moves are applied alongside standard placement moves as an integral part of the annealing process. Of course to accomplish this, basic simulated-annealing retiming moves must be defined and multiple issues must be addressed.

First, how does the placement tool actually implement an annealing-based retiming move? Essentially, the retiming itself can operate much like the incremental retiming moves in [36], in that individual registers are either pushed or pulled one by one through a single level of logic. However, it is integrated much more fully into the placement process itself because instead of using the complicated cost structure from [37] to determine whether or not a given register will likely cause legalization problems, the cost of the new retimed placement is simply evaluated using the same method as any other placement move. Stated more plainly, after each retiming move is made, the wire and timing costs of the new retimed placement are compared with the costs of the old placement. The retiming move is either accepted or rejected using the same probabilistic technique as conventional placement moves.

This approach can use a unified cost function because it deals with newly created registers slightly differently. For example, moving between Figure 6.6a and Figure 6.6b, two new registers are created on the inputs of *LUT B* to retime the register backwards. Unlike the approach in [36], before the placement tool attempts this move, it first ensures that the retiming is feasible. It is entirely possible that there are not enough register locations available in the architecture to support the new registers needed to perform the retiming. If this is the case, rather than creating an illegal placement, the annealer will not attempt this retiming move at all. However, if it is a feasible move, it places these new registers into the closest available legal register locations to the source of their respective signals. However, retiming does not necessarily have to create a new register. For example, moving between Figure 6.6c and Figure 6.6d, one of the inputs to *LUT B* can share the input to *LUT C.* In either case, because each retiming move is required to produce a legal placement and is evaluated individually based upon the change in cost, the placement tool will likely retime the netlist as much or as little as the prevailing conditions will allow.

The second obvious question is how and when should the annealer attempt a retiming move versus a placement move? While the placer could simply flip a coin each time it selects an eligible logic block, the computational ramifications of performing a retiming move should also be considered. Specifically, as discussed in Chapter 5, the underlying placement tool that this approach is built on relies on the more accurate timing information provided by the incremental slack analysis approach. Therefore, the impact that retiming moves have on the accuracy of timing information should be examined.

It is likely that retiming moves will disrupt the system more than conventional placement swaps. Furthermore, retiming moves specifically focus on improving the critical path delay of the existing placement. These two factors together indicate that retiming should probably be performed using the most accurate timing information possible. However, completely accurate timing information can only be obtained by performing a relatively computationally expensive static timing analysis. Thus, similar to the issues brought up in Chapter 5, this means that static timing analysis cannot be performed before and after each incremental retiming move. Of course, this problem becomes worse as the number of registers in the netlist gets larger. Thus, to maximize the accuracy of the timing information while minimizing the

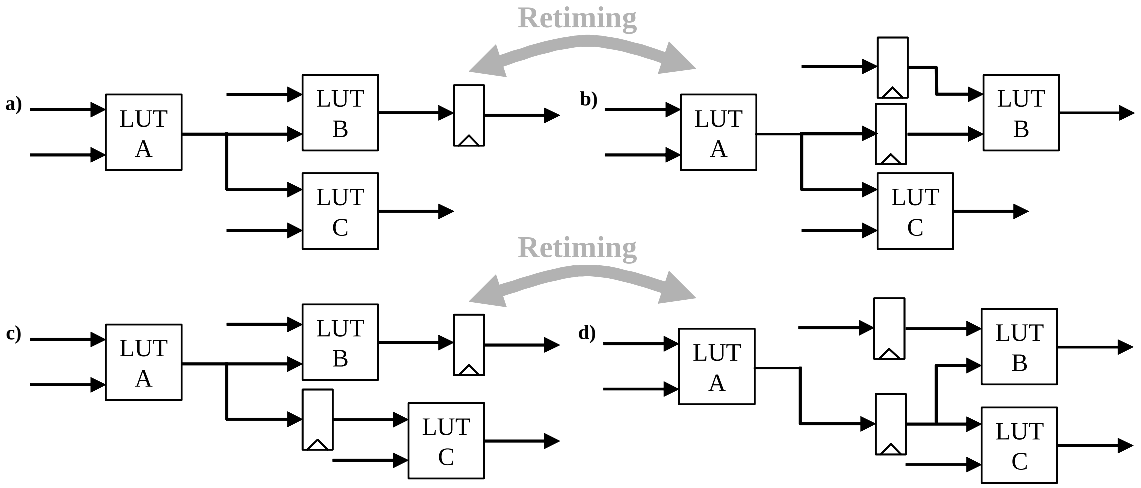


Figure 6.6: Incorporating New Registers Created By Retiming

computational cost, much like the approach in [36], the placement tool bundles multiple retiming moves together. Full static timing analysis is performed once before the retiming moves are attempted, and once after the series has completed.

Following the spirit of the improved placement tool in Chapter 5, the timing information of the system can be incrementally updated between each retiming move. Consider the example in Figure 6.7. The placement tool will update the slack on all of the labeled nets. The slack on nets *1'* and *2'* are obvious because the departure time of sources do not change and the required time of all registers is equal to the current critical path delay of the system. However, updating the slack on nets *4'*, *5*, and *6* is a bit more complicated. This is because the placement tool has to recalculate the departure and required times of LUT *B*. Since the departure time of all registers is zero, the departure time of LUT *B* can be calculated as:

$$\text{LUT } B_{\text{Departure Time}} = FF_{\text{clock to output}} + Max(Delay_5, Delay_6) \tag{6.1}$$

Furthermore, the required time of whatever LUT *B* drives also does not change. Thus, the required time of LUT *B* can be calculated as:

$$\text{LUT } B_{\text{Required Time}} = \text{Sink}_{\text{Required Time}} - Delay_{4'} \tag{6.2}$$

If the retiming was reversed and the registers from the inputs of LUT *B* were pushed forwards to the output, a similar incremental computation could be performed to recalculate the departure and required times of LUT *B*.

As seen in the pseudo-code in Figure 6.8, this integrated retiming and placement technique takes 3 new placement parameters: a retiming *activation point*, *criticality threshold*, and *frequency*. Lines 4-6 show that the retiming activation point functions much like the FF-level placement activation point from the previous section. Essentially, this parameter controls when the placement tool will begin to attempt retiming and placement, as opposed to placement only. Again, since the annealing begins with an arbitrary initial placement, the early portions of the placement process can change the system dramatically. Thus,

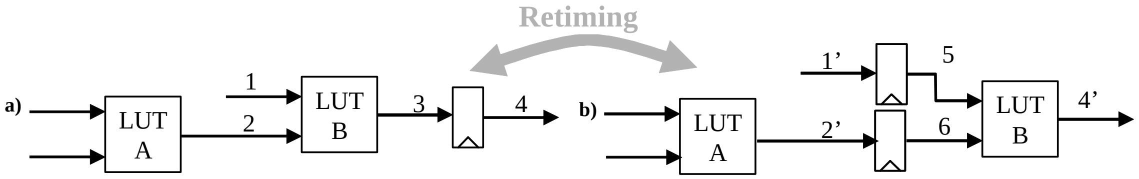


Figure 6.7: Updating Timing Information for New Retiming Registers

```
Placement with Integrated Retiming Moves
0   numMovesPerRetiming =   numAnnealMovesPerTemp / retiming frequency
1   randomly place logic blocks onto architecture
2   determine initial temperature
3   while(!done)
4       if range limit window <= (retiming activation point * max window size)
5           activate retiming placement
6       end if
7       for i = 0 to numAnnealMovesPerTemp
8           if retiming active && (i%numMovesPerRetiming == 0)
9               update critical path delay
10              for all logic blocks
11                  if max input criticality >= retimeCrit && can retime backwards
12                      try to retime once backwards
13                      accept or reject retiming(ΔCost, currTemp)
14                  end if
15                  if max output criticality >= retimeCrit && can retime forwards
16                      try to retime once forwards
17                      accept or reject retiming(ΔCost, currTemp)
18                  end if
19              end for
20              update critical path delay
21          end if
22          attempt placement move
23          accept or reject move(ΔCost, currTemp)
24      end for
25      update critical path delay
26      update currTemp
27      update range limit window
28      evaluate exit criteria
29  end while
```

Figure 6.8: Pseudo-Code for Simulated Annealing-Based Retiming

retiming is unlikely to contribute much during these early stages. Rather, the extra noise retiming creates in the netlist would probably only serve to create problems for the placement tool. Instead, it is likely better to wait until the placement begins to settle down, and leave retiming to the later stages of the placement process. Similarly, the retiming criticality threshold plays the same role as the FF-level placement criticality threshold. As shown in lines 11 and 15, the retiming criticality threshold filters logic blocks that are eligible for retiming based upon the maximum criticality of their input or output connections. Obviously, the more critical a given path is, the more important it becomes to retime the logic blocks along it. Since it is probably best to disrupt the placement as little as possible, the placement tool avoids retiming logic blocks that are not along highly critical paths. Lastly, as shown in lines 0 and 8, the retiming frequency factor controls how often the placement tool attempts to perform a concentrated suite of conditional retiming moves.

6.4: Testing and Results

Like the placement approach in Chapter 5, this new simultaneous placement and physical re-synthesis approach was tested using the MCNC netlists provided with VPR. However, because this approach focuses on the packing and retiming of registers, obviously the 11 purely combinational MCNC circuits are not suitable. Thus, these circuits were not part of the testing process. Furthermore, while the same 22 depth = 1 netlists used in Chapter 5 were part of the testing of this new tool, 22 *depth = 0.33* netlists were also created. These netlists have at least 3 flip-flops after each LUT, rather than only 1. This set of benchmarks simulate an application developer's attempt to not only pipeline the logic of a circuit but also encourage the system to pipeline the interconnect wires. These netlists were created in a very similar manner to the depth = 1 netlists. Specifically, each netlist was minimally pipelined, C-slowed and Leiserson/Saxe retimed such that the maximum logical depth of the circuit was a single LUT. In addition, a single register was placed on each of the primary input pins. After this, the entire netlist was 3-slowed to provide 3 flip-flops on each connection along the critical path. Additional information regarding these benchmarks can be found in Appendix A.

These three groups of netlists were also placed onto a new architecture. While the single LUT/single flip-flop/unit-length wire architecture used in the previous chapter provided a very simple platform for testing and tuning the incremental timing analysis placement approach, this does not necessarily accurately reflect the types of resources present in modern commercial FPGAs. Thus, this new tool was tested using a much more realistic architecture with four 4-LUTs, 4 flip-flops, 20 input pins, and 8 output pins per CLB and length 4 interconnect wires. While not exactly the same as the resources available in recent Altera devices, this does provide a reasonable analog of commercial architectures and is very similar to the architecture suggested by [1].

Testing began by first performing CLB-level placement using the enhanced timing placement technique described in Chapter 5. This provided a good baseline for comparison. Each of the netlists were packed using T-VPack and routed using timing-driven PathFinder. The primary placement parameters (λ and criticality exponent) used to gather these results were set to values suggested by the testing in Chapter 5. Although the exact parameters found during this earlier testing were used for the original sequential MCNC netlists ($\lambda = 0.3$, criticality exponent = 12), due to the discussion in Section 5.6, the best criticality exponent = 12 parameters were used for the depth = 1 netlists ($\lambda = 0.025$, criticality exponent = 12). The same parameters were also used for the depth = 0.33 netlists. These λ and criticality exponent values were maintained throughout the rest of the testing process.

The first round of testing focused on determining good values to use for the new FF-level placement parameters: the activation point, criticality threshold, separation probability, and homing probability. Due to the very large number of potential axes, two simplifications were made to the exploration process. First, rather than testing with all 55 benchmarks, 9 relatively small representational benchmarks were selected – 3 from each group of logic depths. These included *s1423*, *diffeq*, and *bigkey*. Second, some preliminary testing was performed to gather reasonable values for all of the parameters before more detailed testing was implemented on each individually. This preliminary testing found that reasonable results were obtained with an activation point of 0.9, a criticality threshold of 0.9, a separation probability of 0.1 and a homing probability of 0.01.

As seen in Table 6.1, the FF-level placement activation point was first swept through a range of values while holding the criticality threshold at 0.9, the separation probability at 0.1 and the homing probability at 0.01. Four activation points were selected for testing. First, 1.0 started FF-level placement from the beginning of the annealing process. The activation points of 0.9 and 0.0001 started FF-level placement when the movement window reached a value of 90% the overall size of the array and 1, respectively. These activation points roughly represent beginning the FF-level placement one-third or two-thirds of the way through the placement process. An activation point of 0.0 waited until the conventional placement was finished and then restarted the annealing at a slightly higher temperature for an additional short FF-level placement phase. This typically gave the annealer another 5 - 10 temperature iterations to improve the placement with FF-level moves.

The results of this testing are shown in Table 6.1. All 9 of the exploratory netlists were placed and routed 3 times for each set of placement parameters. The placement with the smallest routed critical path delay for a given setting was selected as the best placement. Table 6.1 shows the geometric mean normalized wire cost and post-routing critical path delay for all three netlists within each group of benchmarks. Beginning FF-level placement at an activation point of 0.9 produced the best FF-level placements for all three groups of benchmarks. Thus, this activation point was used for all subsequent testing.

As seen in Table 6.2, the FF-level placement criticality threshold was tested next. Here, the criticality threshold was swept through a range of values while holding the activation point at best value found by the previous experiment (0.9), the separation probability at 0.1 and the homing probability at 0.01. This testing showed that performing FF-level placement on registers that were connected to nets that were 90% critical or more produced the best results for the original sequential and depth = 1 netlists used for exploration. A criticality threshold of 80% or greater produced the best results for the depth = 0.33 netlists tested. Again, these values were passed on to the subsequent rounds of testing. Finally, as seen in Table 6.3 and Table 6.4, the FF-level placement separation probabilities and homing probabilities were tested. Here, a separation probability of 0.1 and a homing probability of 0.1 produced the best results for all of the groups of netlists.

The best FF-level placement parameters discovered by the initial round of testing were used to place all of the 11 original sequential MCNC netlists, the 22 depth = 1 netlists and the 22 depth = 0.33 netlists. This is shown in Table 6.5. Looking at these results, the original sequential netlists do not seem to respond to FF-level placement. The routed critical path delay for these circuits actually degraded very slightly (1.007x).

In some sense, this is to be expected since packing via traditional methods makes sense given the small number of registers in these applications. However, the depth = 1 and depth = 0.33 netlists do seem to benefit quite a bit from performing FF-level placement. The depth = 1 netlists obtain an average of 0.870x

Table 6.1: FF-Level Placement Activation Point Exploration (Clustered Architecture)

		Original Sequen. Netlists Crit. Thres. = 0.9 Sep. Prob. = 0.1 Homing Prob = 0.01		**Depth = 1** Crit. Thres. = 0.9 Sep. Prob. = 0.1 Homing Prob = 0.01		**Depth = 0.33** Crit. Thres. = 0.9 Sep. Prob. = 0.1 Homing Prob = 0.01	
	Activation Point	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD
CLB-Level Placement	-	1.000	1.000	1.000	1.000	1.000	1.000
From Beginning	1.0	0.999	1.011	0.915	0.959	0.859	0.633
From 1/3 Complete	0.9	**0.998**	**1.007**	**0.916**	**0.957**	**0.842**	**0.631**
From 2/3 Complete	0.0001	0.997	1.031	0.954	0.966	0.910	0.724
Post-Processing	0.0	0.997	1.031	0.954	0.966	0.910	0.724

Best of 3 placement and routing attempts

Table 6.2: FF-Level Placement Criticality Threshold Exploration (Clustered Architecture)

	Original Sequential Netlists Activation = 0.9 Sep. Prob. = 0.1 Homing Prob = 0.01		**Depth = 1** Activation = 0.9 Sep. Prob. = 0.1 Homing Prob = 0.01		**Depth = 0.33** Activation = 0.9 Sep. Prob. = 0.1 Homing Prob = 0.01	
	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD
CLB-Level Placement	1.000	1.000	1.000	1.000	1.000	1.000
Crit. Threshold = 0.95	0.999	1.013	0.911	0.982	0.860	0.639
Crit. Threshold = 0.9	**0.998**	**1.007**	**0.916**	**0.957**	0.842	0.631
Crit. Threshold = 0.8	1.004	1.011	0.909	0.963	**0.872**	**0.622**

Best of 3 placement and routing attempts

Table 6.3: FF-Level Placement Separation Probability Exploration (Clustered Architecture)

	Original Sequential Netlists Activation = 0.9 Crit. Thres. = 0.9 Homing Prob = 0.01		**Depth = 1** Activation = 0.9 Crit. Thres. = 0.9 Homing Prob = 0.01		**Depth = 0.33** Activation = 0.9 Crit. Thres. = 0.8 Homing Prob = 0.01	
	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD
CLB-Level Placement	1.000	1.000	1.000	1.000	1.000	1.000
Separation = 0.1	**0.998**	**1.007**	**0.916**	**0.957**	**0.872**	**0.622**
Separation = 0.2	1.004	1.007	0.913	0.965	0.874	0.622
Separation = 0.4	1.001	1.014	0.911	0.970	0.891	0.689

Best of 3 placement and routing attempts

Table 6.4: FF-Level Placement Homing Probability Exploration (Clustered Architecture)

	Original Sequential Netlists Activation = 0.9 Crit. Thres. = 0.9 Sep. Prob = 0.1		**Depth = 1** Activation = 0.9 Crit. Thres. = 0.9 Sep. Prob = 0.1		**Depth = 0.33** Activation = 0.9 Crit. Thres. = 0.8 Sep. Prob = 0.1	
	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD
CLB-Level Placement	1.000	1.000	1.000	1.000	1.000	1.000
Homing = 0.001	1.015	1.023	1.053	1.048	1.113	0.917
Homing = 0.01	0.998	1.007	0.916	0.957	0.872	0.622
Homing = 0.1	**1.001**	**1.006**	**0.892**	**0.949**	**0.750**	**0.608**

Best of 3 placement and routing attempts

Table 6.5: FF-Level Placement Results (Clustered Architecture)

	Original Sequential Netlists Activation = 0.9 Crit. Thres. = 0.9 Sep. Prob = 0.1 Homing Prob = 0.1		**Depth = 1** Activation = 0.9 Crit. Thres. = 0.9 Sep. Prob = 0.1 Homing Prob = 0.1		**Depth = 0.33** Activation = 0.9 Crit. Thres. = 0.8 Sep. Prob = 0.1 Homing Prob = 0.1	
	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD
e64			0.937	0.903	0.830	0.622
ex5p			0.926	0.889	0.762	0.749
apex4			0.971	0.959	0.760	0.732
misex3			0.919	0.945	0.726	0.744
alu4			0.946	0.929	0.726	0.786
des			0.916	0.930	0.682	0.610
seq			0.925	0.993	0.704	0.740
apex2			0.912	1.008	0.701	0.642
spla			0.896	0.887	0.655	0.512
pdc			0.899	0.823	0.642	0.623
ex1010			0.852	0.890	0.604	0.315
s1423	1.001	1.026	0.916	0.938	0.871	0.719
tseng	0.999	0.959	0.837	1.008	0.722	0.700
dsip	0.997	1.004	0.993	0.907	0.709	0.572
diffeq	0.996	0.982	0.845	0.924	0.714	0.573
bigkey	0.998	1.014	0.976	1.010	0.677	0.545
s298	1.007	1.006	0.847	0.975	0.717	0.685
frisc	0.992	1.000	0.735	0.864	0.576	0.588
elliptic	0.991	1.015	0.804	1.006	0.632	0.634
s38584.1	0.998	1.018	0.666	0.475	0.575	0.330
s38417	0.983	0.983	0.682	0.442	0.538	0.532
clma	0.992	1.068	0.747	0.798	0.584	0.363
Geo Mean	**0.996**	**1.007**	**0.865**	**0.870**	**0.682**	**0.588**

Best of 3 placement and routing attempts

better critical path delay with 0.865x better wire cost. The depth = 0.33 netlists respond even more positively with an enormous 0.588x improvement in critical path delay and 0.682x better wire cost. This behavior clearly shows the difficulties that a large number of registers pose to existing packing approaches.

The next phase of testing examined the benefits of adding simultaneous retiming on top of FF-level placement. A similar testing methodology was used to tune this aspect of the tool, but here only the retiming activation point was explored. This is because, first, the exploration into FF-level placement provided a great deal of information regarding how nets of different criticalities interact. Thus, it makes sense to use the same criticality threshold parameters found in Table 6.2 for retiming. Second, the retiming frequency was pegged to 1. This represents attempting to retime one set of registers either backward or forwards through each logic block per simulated annealing temperature iteration. This mimics the behavior of the tool in [36].

Table 6.6 shows the exploration of different retiming activation points from 0.9 to 0.0. Similar to the FF-level placement activation point, a retiming activation point of 0.9 begins retiming moves when the placement window has reached 90% the maximum size of the array, a retiming activation point of 0.0001

Table 6.6: Retiming Activation Point Exploration (Clustered Architecture)

		Original Sequential Netlists Crit. Thres. = 0.9 Retiming Freq. = 1.0		**Depth = 1** Crit. Thres. = 0.9 Retiming Freq. = 1.0		**Depth = 0.33** Crit. Thres. = 0.8 Retiming Freq. = 1.0	
	Activation Point	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD
CLB-Level Placement	-	1.000	1.000	1.000	1.000	1.000	1.000
Best FF-Level Placement	-	1.001	1.006	0.892	0.949	0.750	0.608
From 1/3 Complete	0.9	1.044	1.018	0.919	0.987	0.771	0.634
From 2/3 Complete	0.0001	1.007	1.030	**0.909**	**0.966**	0.769	0.642
Post-Processing	0.0	**1.002**	**0.999**	0.914	0.983	**0.759**	**0.615**

Best of 3 placement and routing attempts

Table 6.7: Simultaneous Retiming and Placement Results (Clustered Architecture)

	Original Sequential Netlists Activation = 0.0 Crit. Thres. = 0.9 Retiming Freq. = 1.0		**Depth = 1** Activation = 0.0001 Crit. Thres. = 0.9 Retiming Freq. = 1.0		**Depth = 0.33** Activation = 0.0 Crit. Thres. = 0.8 Retiming Freq. = 1.0	
	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD	Norm. Wire Cost	Norm. Routed CPD
e64			0.904	0.882	0.830*	0.622*
ex5p			0.926*	0.889*	0.762*	0.749*
apex4			0.971*	0.959*	0.781	0.732
misex3			0.919*	0.945*	0.726*	0.744*
alu4			0.946*	0.929*	0.726*	0.786*
des			0.897	0.885	0.682*	0.610*
seq			0.903	0.966	0.701	0.701
apex2			0.885	1.000	0.701*	0.642*
spla			0.864	0.844	0.666	0.505
pdc			0.858	0.797	0.642*	0.623*
ex1010			0.852*	0.890*	0.604*	0.315*
s1423	1.007	0.991	0.905	0.922	0.869	0.705
tseng	0.999	0.959	0.840	1.002	0.720	0.700
dsip	0.997	1.000	0.993*	0.907*	0.709*	0.572*
diffeq	0.998	0.982	0.845*	0.924*	0.734	0.573
bigkey	0.998*	1.014*	0.976	1.003	0.677*	0.545*
s298	1.007*	1.006*	0.847*	0.975*	0.708	0.678
frisc	0.992*	1.000*	0.735*	0.864*	0.571	0.588
elliptic	0.995	0.978	0.800	1.000	0.631	0.586
s38584.1	0.998*	1.018*	0.645	0.473	0.570	0.330
s38417	0.983*	0.983*	0.645	0.431	0.538*	0.532*
clma	0.999	1.045	0.747*	0.798*	0.584*	0.363*
Geo Mean	**0.998**	**0.998**	**0.854**	**0.860**	**0.683**	**0.583**

Best of 3 placement and routing attempts. *Indicates result reverted to values from FF-level placement.

begins retiming moves when the placement window has reached 1 and a retiming activation point of 0.0 runs an additional post-placement retiming and annealing phase. This retiming was performed on top of FF-level placement with the best parameters suggested by Table 6.4. Again, this initial testing was performed by placing and routing each of the 9 exploratory benchmarks 3 times for each set of placement parameters. The placement with the smallest routed critical path delay for a given setting was selected as the best placement. Table 6.6 shows the geometric mean wire cost and routed critical path delay normalized to the results produced by performing CLB-level placement.

Unfortunately, looking at these results, it appears as though the addition of retiming does not greatly improve upon the performance of only applying FF-level placement. While retiming with an activation point of 0.0 very slightly helped the original sequential MCNC netlists used for exploration, even the best setting of retiming actually degraded the results for the depth = 1 and depth = 0.33 netlists compared to only performing FF-level placement. These mediocre results were further confirmed when testing was expanded to the full set of benchmarks, as shown in Table 6.7. It should be noted that the results reported here are actually a mixture of the results obtained by performing only FF-level placement and retiming on top of FF-level placement. Since the retiming activation point suggested by the testing in Table 6.6 for the original sequential and the depth = 0.33 netlists is actually after normal placement has been completed, any degradation caused by the retiming can be eliminated by simply reverting the system to the placement found before retiming was activated. These corrected results are denoted with an asterisk. While a bit more difficult for the depth = 1 netlists, since the best retiming activation point found during the previous testing was 0.0001, a similar correction can be made by performing two partial placement runs once the movement window reaches 1 – once implementing retiming and once without. That said, despite best attempts, the benefits of retiming seem relatively small compared to only performing FF-level placement. Performing retiming did not seem to improve critical path delay by more than about 1%.

These results are somewhat surprising. The testing in [37] and [36] showed a 0.838x and 0.930x improvement in critical path delay for their respective integrated retiming approaches. However, when comparing the results of this new technique to previous results, the baseline that these previous papers used should also be kept in mind. Both of these papers only compared their retiming approach to relatively classical placement techniques. On the other hand, the technique suggested here is compared to a highly enhanced placement approach and a placement tool that implements FF-level placement. This change in comparison target creates two fundamental differences between the results from these previous works and the results gathered here.

First, since the incremental slack analysis placement approach described in Chapter 5 already obtains such good results compared to conventional placement techniques, from the viewpoint of this toolflow, the baseline placements used for comparison in previous work likely contain a lot of room for improvement. Stated another way, incremental slack update placement already improves performance so much that it may subsume the gains reported by previous retiming efforts just by itself. For that matter, this also makes it much more difficult for any retimer built on top of this placement algorithm to achieve further gains.

Second, although very little is known about the placement tool used for comparison in [36], VPR certainly does not change the packing of a netlist during placement. Thus, in some sense the results reported include the gains provided by FF-level placement. This is because the legalization phases of both the tools in [37] and [36] migrate registers between different CLBs in order to make room for the new registers created by retiming. These legalization tools specifically seek out registers based upon net criticality, so it is possible that a large portion of their respective gains are specifically due to changes in CLB packing, not necessarily the retiming itself.

However, this is not to say that retiming during placement should be ignored. Rather, the results indicate that there are other forces at work in the testing performed thus far. First, placement with more accurate timing information improves circuit quality significantly. Second, packing is a much more pressing issue when mapping netlists with a large number registers to architectures with multiple BLEs per CLB. Lastly, there is likely some characteristic that these new architectures have that makes retiming after placement less fruitful.

The impact of a placement tool that has a more accurate way of tracking timing is relatively easy to quantify. This can be done by simply repeating placement with a VPR-style timing update approach. Unfortunately though, evaluating the contribution of changing the packing of the system is a bit more difficult. However, this factor can be minimized by mapping to an architecture that only has one BLE per CLB. While retiming can still be performed on applications mapped to this kind of architecture, it does not pose a very large packing problem since each flip-flop must be mapped to its own CLB. Although some flip flops must still be packed into CLBs with LUTs, the system does not have to worry about the initial packing limiting the placement because it put registers from different parts of the circuit into the same CLB. This problem can be further reduced by eliminating the very heavily registered depth = 0.33 netlists.

Finally, it is likely that retiming was less effective on the architectures used for the testing in Table 6.7 because they had multi-segment wires. Longer wires are generally incorporated into FPGAs because they allow the system to make connections using fewer programmable wire segments. However, this also means that each logic block can reach a much larger number of other logic blocks with a single wire delay. Consider the two architectures in Figure 6.9. While each logic block can reach four others with a single wire delay in the architecture with unit-length wires, each logic block can reach 26 others with a single wire delay in an architecture with length-4 wires. By the same token, the number of logic blocks that can be reached with two wire delays is 8 for an architecture with unit-length wires and 116 for an architecture with length-4 wires.

This much larger number of locations that can be reached quickly means that the timing-driven placement problem is easier. In turn, this makes retiming after placement less critical. On a unit-length wire architecture, certain wires have to be made longer because not every block can fit next to the other blocks

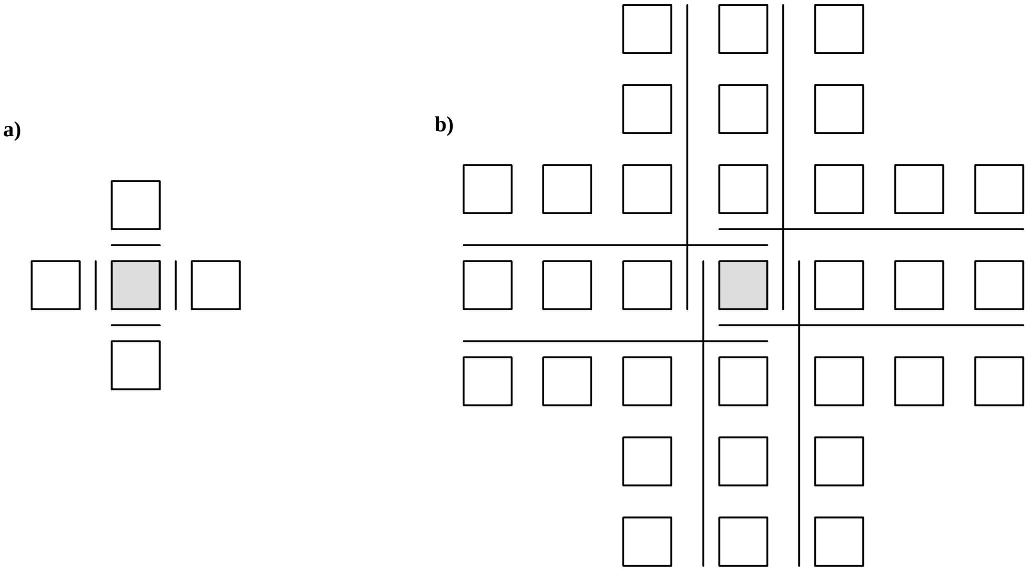


Figure 6.9: Logic Blocks Reachable with 1 Wire Segment

to which it is connected. These longer wires make certain connections slower than others. However, which connections are going to be the slow ones, and by how much, is impossible to determine without information about the placement. This makes retiming after placement very important. However, with length-4 wires, and further compounded by clustered CLBs, a much larger fraction of the blocks can be close to the other blocks to which they are connected. This strongly reduces the need for long, slow wires and makes the delay of all the connections in the system much more predictable.

This behavior can be seen by repeating the testing of both FF-level placement and retiming placement, this time mapping to an unclustered, unit-length wire architecture. As seen in Table 6.8 and Table 6.9, as expected, FF-level placement plays a much smaller role on these architectures. The critical path delay is only reduced by 0.991x for the original sequential netlists and by only 0.980x for the depth = 1 netlists. However, retiming improves critical path delay by a much larger amount, 0.947x for the original sequential netlists and 0.952x for the depth = 1 netlists. This testing not only shows that simultaneous retiming and placement can improve critical path delay, it also suggests that the benefit can be somewhat architecture dependent. Furthermore, it is also obvious that while the incremental slack analysis placement approach described in Chapter 5 produces much better placements than VPR, there is still a small amount of room for further improvement.

The effect that the architecture has on retiming can also be seen Table 6.10. Here, the fastest implementations found by FF-level placement were retimed using the Leiserson/Saxe method. However, unlike the Leiserson/Saxe retiming applied before placement, this retiming was performed using the actual wire delays in the final placement. Of course, the critical path delay reported by retiming netlists in this way is wildly optimistic. Not only does this run into the problems discussed earlier regarding finding enough registers and disturbing the existing placement, this “optimal” retiming also assumes that registers

Table 6.8: Comparison of FF-Level Placement and Retiming Placement (Unclustered Architecture) – Original Sequential MCNC Netlists

	VPR CLB-Level Placement		Incremental Slack CLB-Level Placement		FF-Level Placement		Retiming Placement	
	Wire	CPD	Wire	CPD	Wire	CPD	Wire	CPD
s1423	1.028	0.914	1.000	1.000	1.004	0.953	0.996	0.898
tseng	1.046	1.042	1.000	1.000	0.988	0.942	0.988*	0.942*
dsip	0.837	1.333	1.000	1.000	1.008	0.887	1.027	0.755
diffeq	1.051	1.119	1.000	1.000	1.001	1.052	0.998	1.012
bigkey	0.876	1.538	1.000	1.000	0.997	0.965	1.004	0.951
s298	1.079	0.990	1.000	1.000	0.997	0.985	0.997*	0.985*
frisc	1.081	1.067	1.000	1.000	0.999	0.986	1.010	0.961
elliptic	1.067	1.205	1.000	1.000	0.994	1.057	1.008	0.980
s38584.1	0.985	1.258	1.000	1.000	0.997	0.984	0.997	0.968
s38417	1.011	1.205	1.000	1.000	1.001	0.989	1.009	0.964
clma	1.053	1.451	1.000	1.000	1.025	1.125	1.023	1.022
Geo Mean	**1.007**	**1.179**	**1.000**	**1.000**	**1.001**	**0.991**	**1.005**	**0.946**

Best of 3 placement and routing attempts. *Indicates result reverted to values from FF-level placement.

Table 6.9: Comparison of FF-Level Placement and Retiming Placement (Unclustered Architecture) – Depth = 1 MCNC Netlists

	VPR CLB-Level Placement		Incremental Slack CLB-Level Placement		FF-Level Placement		Retiming Placement	
	Wire	CPD	Wire	CPD	Wire	CPD	Wire	CPD
e64	1.055	1.818	1.000	1.000	1.000	1.042	1.000*	1.042*
ex5p	1.057	1.619	1.000	1.000	0.997	1.059	0.996	1.026
apex4	1.065	1.608	1.000	1.000	1.005	1.002	1.004	1.002
misex3	1.146	1.445	1.000	1.000	1.005	0.983	1.007	0.863
alu4	1.162	1.366	1.000	1.000	0.998	1.002	0.995	1.000
des	1.048	1.756	1.000	1.000	1.029	0.906	1.018	0.906
seq	1.076	1.784	1.000	1.000	1.001	1.053	1.001	1.052
apex2	1.100	1.721	1.000	1.000	0.997	0.971	0.997	0.970
spla	1.081	1.213	1.000	1.000	1.014	0.921	0.997	0.730
pdc	1.080	2.104	1.000	1.000	0.999	1.002	0.995	0.940
ex1010	1.107	1.479	1.000	1.000	1.007	1.093	1.000	1.085
s1423	1.038	2.002	1.000	1.000	1.012	0.910	1.014	0.876
tseng	1.076	1.853	1.000	1.000	1.027	0.908	1.027	0.908
dsip	1.242	1.245	1.000	1.000	1.013	0.970	1.036	0.919
diffeq	1.051	1.870	1.000	1.000	1.031	0.939	1.051	0.918
bigkey	1.152	1.547	1.000	1.000	1.011	0.979	1.011*	0.979*
s298	1.108	1.484	1.000	1.000	0.997	0.942	0.990	0.922
frisc	1.000	2.452	1.000	1.000	0.997	0.973	0.997	0.971
elliptic	1.097	1.761	1.000	1.000	1.007	0.962	1.007	0.962
s38584.1	1.158	1.226	1.000	1.000	1.018	1.017	1.025	1.016
s38417	1.031	2.060	1.000	1.000	0.992	0.974	1.028	0.917
clma	1.109	1.350	1.000	1.000	0.997	0.991	0.999	0.984
Geo Mean	**1.091**	**1.643**	**1.000**	**1.000**	**1.007**	**0.980**	**1.009**	**0.951**

Best of 3 placement and routing attempts. *Indicates result reverted to values from FF-level placement.

Table 6.10: Effect of Architecture on Leiserson/Saxe Retiming After Placement

	Clustered Architecture Four BLEs, Length-4 Wires	Unclustered Architecture One BLE, Unit-Length Wires
Depth = N Netlists	0.936	0.717
Depth = 1 Netlist	0.734	0.639

Results normalized to critical path delay found after FF-level placement.

can be placed wherever they are needed – potentially in the middle of wires rather than in discrete CLB locations. That said, this "theoretical" retiming offers some upper bound on how much room for improvement there is after placement. While the depth = N netlists could only be improve by an average of 0.936x when placed on the clustered, length-4 wire architecture, they could be improved by an average of 0.717x when placed on the unclustered, unit-length wire architecture. Similarly, the depth = 1 netlists could only be improve by an average of 0.734x when placed on the clustered, length-4 wire architecture, they could be improved by an average of 0.639x when placed on the unclustered, unit-length wire architecture. Although these results are purely theoretical, they suggest that the placer can better balance delay along the critical path in netlists mapped to the clustered, length-4 wire architecture. This means that retiming after placement is probably less essential when mapping to more sophisticated architectures rather than simpler devices.

6.5: Conclusions and Future Research

This chapter investigated how classical packing, retiming and placement tools interact. While the conventional toolflow works relatively well for lightly registered applications, its highly compartmentalized and purely feed-forward nature can cause problems when attempting to deal with more heavily registered netlists.

Packing is particularly vulnerable to some of these issues because the conventional approach that tools like VPR use generally applies packing to a netlist followed by strictly CLB-level placement. Because traditional packing techniques tend to put registers into the same CLB as their source LUT, this can limit the capability of the system to use these registers to balance delay along long connections. Furthermore, when attempting to handle netlists with signals that have multiple registers, conventional packing tools can fuse unrelated parts of the circuit together. This makes the placement problem much more difficult, both from the standpoint of reducing wiring cost and improving critical path delay.

However, solving this problem is not simply a matter of opening the placement tool to FF-level annealing moves. Doing so can not only dramatically increase the time required for simulate annealing, it can create problems for the basic achievable quality as well. Moving flip-flops and LUTs strictly separate from each other can prevent the system from making larger-scale moves. Simply put, the placement tool needs to have the capability to perform both CLB and FF-level moves. This allows the system to change the packing of CLBs while still maintaining the ability to make coarser changes to the placement. Towards that end, this chapter introduces a new hybrid placement approach that gives the placement tool the capability to either move an entire CLB, or individually migrate highly critical flip-flops.

Retiming also presents a problem to the conventional toolflow. Since retiming can create or delete registers within a netlist, the most obvious point to apply it is before packing and placement. However, the optimizations that can be made at this point are very limited since very little is known about the potential delay required by the interconnect. Unfortunately, it is also not obvious how retiming could be applied after placement when more is known about the criticality of each of the nets. This is because retiming can change the netlist significantly, necessitating a brand new placement that may or may not have the same timing characteristics as the original. Although there have been multiple previous research attempts to deal with this problem, the majority of these approaches have still struggled with the same basic issue: how to support aggressive retiming without creating a problem for timing convergence.

Essentially, the problem is that it is unreasonable to expect that the system will be able to clean itself up satisfactorily when the retimer makes major and sudden changes to the netlist. Thus, this chapter also introduced a new integrated retiming and placement approach. This technique differs from previous work in three main ways. First, rather than performing retiming as a single, highly disruptive step, it applies multiple stages of more incremental annealing-based retiming moves. Second, the new registers created by these much smaller retiming steps are then integrated into the rest of the placement with a hybrid CLB/FF-level placement approach. Third, this technique avoids issues with CLB input and output legalization by never creating an illegal placement in the first place.

Unfortunately, determining how well this new integrated placement and physical re-synthesis approach performs was a bit difficult. Largely, this is because the problems that packing and retiming face are greatly dependant on the characteristics of both the incoming netlist and target architecture. Specifically, conventional packing works very well when the number of registers in a netlist is relatively low or when mapping to an architecture with few BLEs in each CLB. At best, the FF-level placement technique suggested in this chapter only provided a vanishingly small improvement for the original sequential MCNC netlists on both the clustered and unclustered architectures, as compared to CLB-level placement. This is because these netlists do not have enough registers to create a problem for conventional packing. The improvement for more heavily registered circuits is also relatively small for unclustered architectures. Compared to CLB-level placement, the depth = 1 netlists only obtained a 0.980x improvement in critical path delay when mapped to a single LUT/single flip-flop architecture. This is because the packing tool for an unclustered architecture does not inherently bundle enough LUTs and flip-flops together in a single CLB to greatly restrict the subsequent placement step. However, when the number of registers in a netlist is relatively high and there are multiple BLEs in each CLB of the architecture, packing becomes a much larger problem. The depth = 1 and depth = 0.33 netlists obtained a 0.890x and 0.625x improvement in critical path delay, respectively, when mapped to a four 4-LUT/four flip-flop architecture using the FF-level placement approach described in this chapter, as opposed to a CLB-level only technique.

Retiming has a similar issue regarding dependence on the architecture. Here, it is likely that the placement problem for the netlists used in this study was not hard enough to truly present a challenge on architectures with longer wire segments and clustered CLBs. At best, retiming provided a few percent critical path delay improvement for placement on architectures with length-4 wires and four BLEs in each CLB. This is probably because the logic blocks in these architectures can reach a dramatically larger number of other logic blocks with relatively few wires. This makes these architectures inherently faster and, perhaps more importantly, the delay of different nets more predictable, even prior to placement. However, retiming plays a much larger factor on architectures with shorter wires and unclustered CLBs. Retiming during placement on a unit-length wire architecture with one BLE per CLB improved delay by a factor of 0.954x for the original sequential MCNC netlists and by a factor of 0.970x for the depth = 1 netlists as compared to a FF-level placement approach without retiming.

Looking to the future, it is likely that the issues surrounding packing will only get worse. This has some interesting implications for the runtime of placement. First, as consumers demand more complex and higher performance devices, it is likely that the number of LUTs and registers in applications will go up. For that matter, recent trends in commercial devices tend towards using CLBs that incorporate a larger number of BLEs. Thus, while the hybrid CLB/FF-level placement approach described in this chapter appeared to work very well, the runtime of any algorithm based solely upon simulated annealing will likely be extremely long for future applications. However, as mentioned in Chapter 5, many commercial placement tools use a fast, but relatively inaccurate approach to provide a global placement. The natural speed advantage of these tools makes low-level placement far more tractable. It would be interesting to see how a non-simulated annealing global placement tool could interact with a hybrid CLB/FF-level placement tool.

Furthermore, there is a large body of work that has considered another kind of physical re-synthesis: *logic duplication*. Like retiming, logic duplication can cause problems for the classical toolflow. As discussed in [24], logic duplication attempts to replicate portions of a netlist that limit performance due to fanout. Consider the example in Figure 6.10. After placement, the original netlist on the left has a long wire to connect *LUT A* and *LUT C*. This type of situation could occur for a variety of reasons, but most likely the placement of the blocks that connect to *LUTs B* and *C* pull these blocks in opposite directions. However, as seen on the right, duplicating *LUT A* could reduce the number of long wires in the system. Of course, duplicating parts of the circuit restructures the netlist and increases the area requirements, so this must be done very carefully. Unfortunately, determining which nets present a bottleneck can only really be performed after placement, so logic duplication can suffer from the same type of problems regarding timing closure as retiming.

However, similar to retiming, the effect of logic duplication is also likely architecture dependent. In preliminary testing that attempted to replicate timing critical registers on an architecture with length-4 wires, the potential improvement in critical path delay was relatively minor. It is possible that this is due to the fact that architectures with longer wire segments do not need as much duplication, but some additional investigation is necessary to more fully explore the possibilities and limitations of duplication on modern architectures.

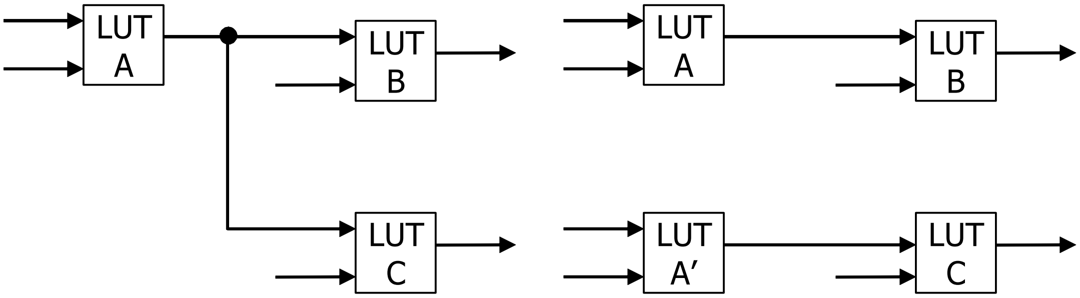


Figure 6.10: Logic Duplication

Chapter 7: Register-Aware Routing

Although all of the tools in a netlist compilation CAD flow play some role in determining the performance of an application mapped to an FPGA, addressing timing concerns during the routing process is particularly important. This is because routing sets the exact communication paths between different logic blocks. The communication in an FPGA-based application is critical because the delay accumulated in the interconnect contributes so heavily to the overall timing of the system. However, while Section 4.4 discussed a classical algorithm for timing-driven routing, this type of approach cannot necessarily be used to map applications to all FPGAs. This is because the register resources that some architectures provide pose a fundamentally different routing problem, breaking some of the basic assumptions necessary to use classical techniques.

This chapter will discuss the nature of some of these architectural design decisions and describe how the connectivity of the registers in an FPGA can affect the CAD algorithms needed to effectively use these resources. This will lead to a discussion of the *pipelined routing problem* and an introduction to the only two known heuristics that address it: *PipeRoute* and *QuickRoute*. Unfortunately, both of these algorithms are purely congestion-driven, and this chapter will outline some of the issues that prevent these approaches from borrowing existing timing-driven routing methodologies. Finally, this chapter will suggest a new timing-driven pipelined routing algorithm that avoids these problems and can significantly improve circuit performance for architectures that require pipelined routing.

7.1: Registers with Limited Connectivity

Some FPGA architectures may limit the accessibility of some of the registers in the system. For example, in the registered track-graph FPGA discussed in the last chapter [38], the flip-flops embedded in the communication network are only connected to a maximum of four wires. As seen in Figure 6.2c, the incoming and outgoing signals of each of these registers must be routed on one specific wire domain. Thus, to use one of these registers, it must be driven from one of four wires coming from either the top, bottom, left or right of the switchbox. The registered output can then leave on one of the wires on the remaining three sides. This extremely limited connectivity is a stark contrast to the accessibility of the more conventional registers found inside CLBs. Flip-flops within logic blocks are generally connected to all or most of the wires inside the channels that surround each CLB. Since the wire channels in modern FPGAs contain hundreds of individual wires, the routing flexibility of registers inside logic blocks is extremely high.

However, limited register accessibility is common on architectures that attempt to increase the number of registers they provide. In general, this is because these systems would like to introduce as many additional registers as they can while minimally disturbing the rest of the system. From an area and performance standpoint, an architecture like in [38] cannot afford to connect the registers embedded in the communication network to all of the wires that enter or exit its switchbox. As seen on the left of Figure 7.1, fully connecting even one register can require extremely wide input multiplexers and output demultiplexers. As seen on the right of Figure 7.1, FPGA architecture designers would rather increase the number of registers but decrease the communication flexibility of each one.

This architectural choice has a subtle but very important impact on the CAD tools. Specifically, the placer cannot map flip-flops in a netlist in the traditional way to registers in an architecture that have limited input and output connectivity. Although the placement tool can temporarily map flip-flops to these locations during the annealing process to get a general idea of local register supply versus demand, these assignments cannot be binding like the placement of LUTs and registers with a high degree of connectivity.

In the conventional toolflow, after placement is completed the locations of all the LUTs and flip-flops in the netlist are fixed. However, if the placement of flip-flops mapped to registers with limited connectivity is fixed after annealing, this can interfere with basic routability of the system. This is because fixing the location of these registers during placement also forces the system to use specific wires to get in and out of these resources. In some sense, because these registers are connected to so few wires, this also fixes the routing for these signals. This characteristic effectively blocks the capability of the router to choose the path of these signals. Unlike the more conventional registers mapped to logic block locations, the router cannot change the wires it uses to get to these registers to resolve congestion. This can dramatically affect the routability of netlists that make use of registers with low degrees of connectivity. Thus, to maintain the routability of the device, the CAD tools must be able to reassign the locations of flip-flops mapped to registers with limited connectivity after placement is completed.

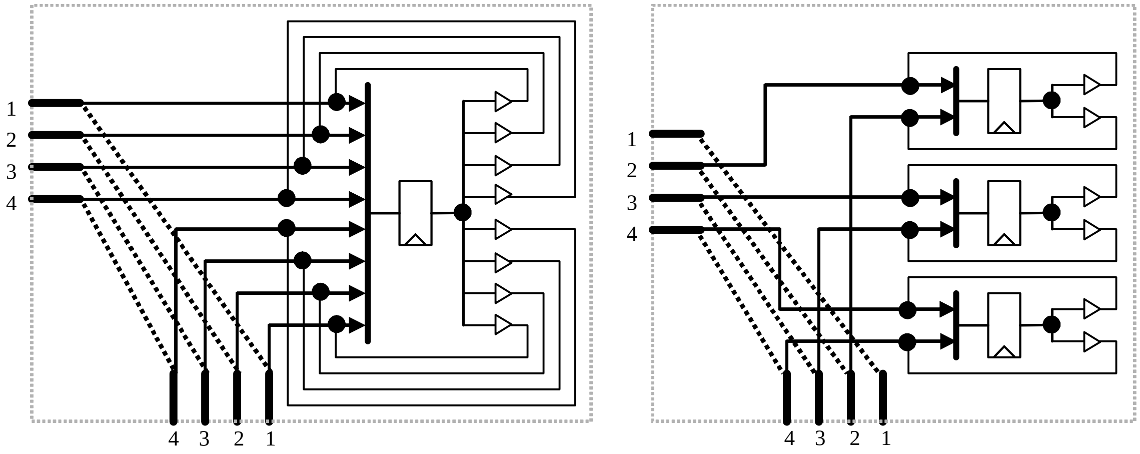


Figure 7.1: Impact of Connectivity on Area and Number of Switchbox Registers

7.2: Pipelined Routing Problem

In a way, the CAD tool flow described in [38] and discussed in Section 6.2 avoids this problem entirely on their architecture by only assigning flip-flops in a netlist to the registers in the interconnect after placement and routing has been completed. However, as mentioned, this severely limits the usability of these registers and makes it impossible for the system to map flip-flops that are not generated by retiming to these interconnect registers. This leads to poor register utilization.

A new CAD approach is necessary to more efficiently use registers with limited connectivity. While the discussion in the previous section suggests that the CAD tools should assign these registers during the routing process, this fundamentally changes the nature of routing itself. No longer is it simply a matter of finding the cheapest path between a source and sink, the router now needs to find a path between a source and sink that goes through exactly *N* registers. Formally introduced in [34], this problem is officially known as the *N-Delay Routing* problem and has been proven to be *NP-Complete*.

While PathFinder [28] and its predecessors demonstrated that the conventional routing problem of congestion resolution for multi-terminal, multi-net circuits is very difficult on most modern FPGA architectures, it breaks down into much simpler sub-problems. For example, ignoring congestion, Dijkstra's shortest-path algorithm can be used to quickly find routes for all two-terminal nets. The difficulty of the *N*-delay routing problem stems from the fact that the additional latency constraint on a signal precludes the use of Dijkstra's algorithm. This is a large handicap since conventional routing techniques generally use some form of Dijkstra's algorithm as a foundation.

The *N*-delay routing problem breaks Dijkstra's in two ways. First, the lowest cost path from source to the sink may not meet the specified latency requirement. More importantly, the cheapest path to any given node along the way may not be the best path, since it may not even form the prefix of any legal route. This issue is clearly illustrated in Figure 7.2. In this example, the router would like to find a path between the source *S* and the sink *K* that goes through exactly one pipelining register. Assuming a unit cost model, Dijkstra's algorithm fails to find a valid one-latency path. Obviously, (*S*, *d*, *e*, *f*, *K*) is the cheapest path, but it does not meet the one clock cycle latency requirement.

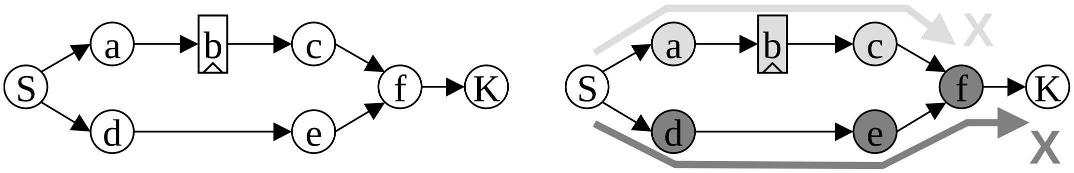


Figure 7.2: Failure of Dijkstra's Algorithm for the N-Delay Routing Problem

Of greater concern, though, is why Dijkstra's fails. The reason that Dijkstra's does not find the valid path through register *b* is because node *f* is explored first by the zero-latency search from (*S*, *d*, *e*). Since Dijkstra's algorithm marks all nodes when they are visited, this prevents the initially more expensive (*S*, *a*, *b*, *c*) route from continuing on to the sink. This problem becomes even more complicated when considering multi-terminal, multi-latency nets and the need for congestion resolution.

The two following sections describe the only known algorithms to address the *N*-Delay routing problem: PipeRoute and QuickRoute. Details of these algorithms are discussed and their advantages and disadvantages are examined.

7.2.1: PipeRoute

PipeRoute [33] was the first heuristic designed to confront the *N*-delay routing problem. Although it is NP-Complete, the authors prove in [34] that a one-latency route can be found in polynomial time. They begin by showing that a normal Dijkstra's breadth-first search is not sufficient given the difference between the input and output nets of a register. As seen in Figure 7.3, if *S* is both the source and sink, the router will not find a valid one-latency path if it simply marks nodes visited or not visited. This is because neither search can complete a path around the ring. Assuming a unit-cost model, the search from (*S*, *a*, *b*, *c*) cannot continue to node *f* because it has already been visited by the other half of the search through (*S*, *d*, *e*). By the same token, the search from (*S*, *d*, *e*, *f*) cannot continue to node *c* because it has already been visited by the other half of the search through (*S*, *a*, *b*). To solve this problem the router must also note the associated latency when a node is explored. That is, a post-register wave (latency=1) can expand to a given node even if it has already been explored by a pre-register wave (latency=0) and vice versa. This is called a *Combined-Phased Breath-First Search*.

However, the authors go on to show that even this is not entirely adequate. Consider the example in Figure 7.4. Even if the router allows nodes to be visited both at latency zero and latency one separately, it can enter a similar deadlock if the graph is slightly different. Here, the latency 0 search through (*S*, *d*, *e*, *c*) cannot continue to node *b* because it has already been explored at latency 0 through (*S*, *a*). However, the

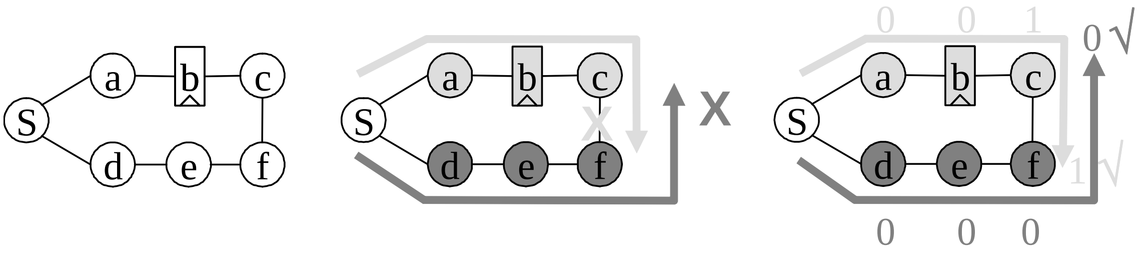


Figure 7.3: Combined-Phase Breadth-First Search

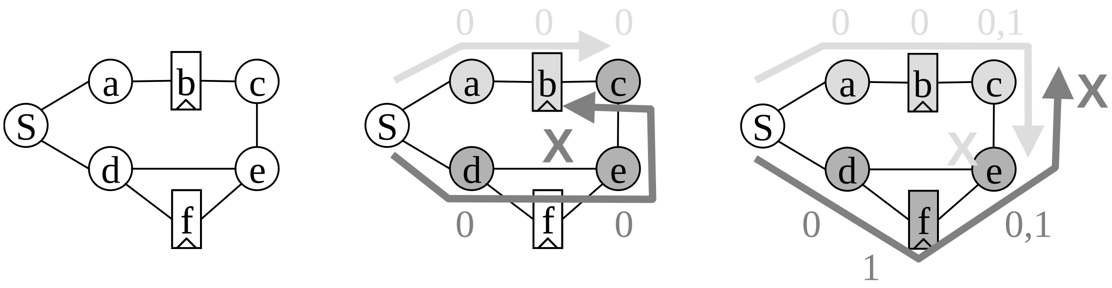


Figure 7.4: Failure of Combined-Phase BFS and Need for 2 Combined-Phase BFS

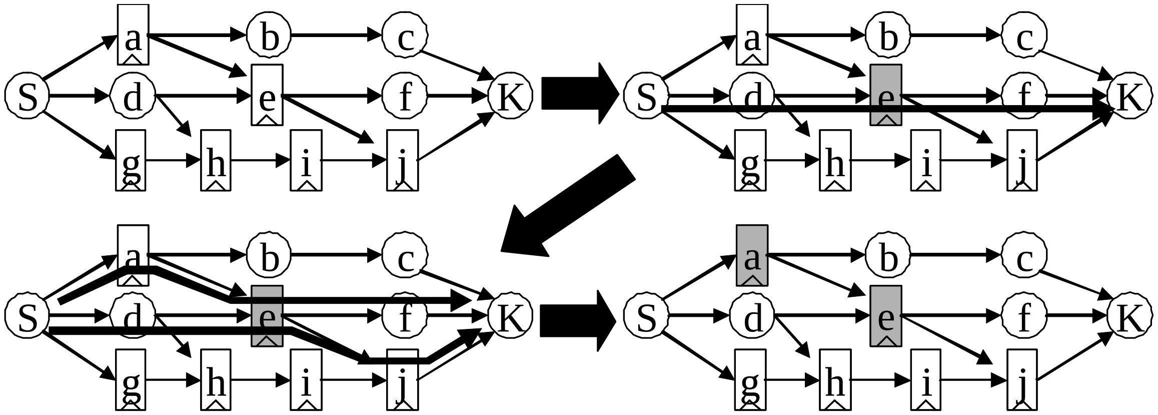


Figure 7.5: Greedy Accumulation of Multiple-Latency Routes

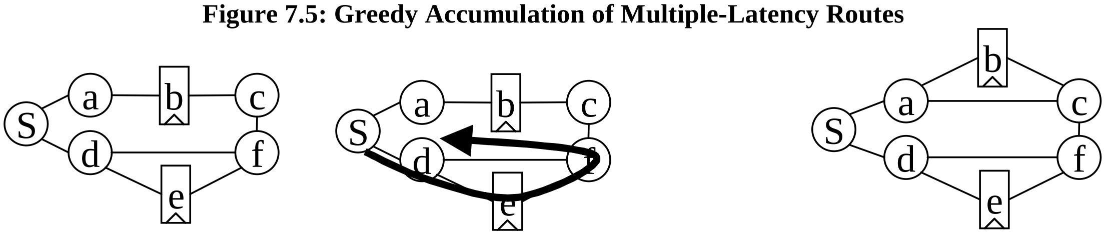


Figure 7.6: PipeRoute and Self Intersection

latency 1 search through (*S*, *a*, *b*, *c*) cannot continue to node *e* because it has already been explored at latency 1 through (*S*, *d*, *f*) and the latency 1 search through (*S*, *d*, *f*, *e*) cannot continue to node *c* because it has already been explored at latency 1 through (*S*, *a*, *b*). In [34] the authors prove that these problems can be avoided and they can guarantee optimality if the router allows nodes to be visited once at latency zero and twice at latency one. This is called a *2Combined-Phased BFS*.

PipeRoute uses this 1-delay router to iteratively form multiple latency paths. As seen in Figure 7.5, to find a two-latency path from the source *S* to the sink *K*, PipeRoute first attempts to find a one-latency path. If this initial single-register route elects to use register *e*, as in the top right of Figure 7.5, the next step is to attempt to replace either the link from *S* to *e* or *e* to *K* with its own one-latency route. As shown in the bottom left of Figure 7.5, PipeRoute would select the lowest cost alternative between the routes (*S*, *a*, *e*, *f*, *K*) and (*S*, *d*, *e*, *j*, *K*). Unfortunately, this is a greedy accumulation process. For example, if the netlist required a four-latency route and PipeRoute selected an interim three-latency path through (S, a, e, j, K), it would be unable to find a valid route. This is because there is no way for any of the links ($S \rightarrow a$), ($a \rightarrow e$), ($e \rightarrow j$), or ($j \rightarrow K$) to be replaced with its own single latency link.

PipeRoute uses this iterative multi-latency search technique to replace Dijkstra's algorithm in PathFinder. It maintains PathFinder's iterative routing scheme and cost formulation in an outer loop to gradually resolve congestion. Unfortunately, this approach has some subtle yet serious limitations. Although the authors prove that their 1-delay router is optimal, their definition of an optimal path allows a route to cross over itself. For example, on the left side of Figure 7.6, if S is both the source and sink, PipeRoute realizes that the shortest one-latency path is the route (S, d, e, f, d, S). Unfortunately, this path visits a register and then doubles back onto itself. This is clearly not a valid physical route since one node must simultaneously carry a value from the current clock cycle and the previous one.

The authors justify their definition of an optimal path by indicating that since they use PathFinder in their outer loop, its natural congestion avoidance will resolve these problems over multiple routing iterations. Unfortunately, PathFinder may not be able to discourage this type of path self-intersection on many common architectures. First, present sharing cost, or the p_n term in Equation 4.7, cannot play any role in preventing self-intersection regardless of architecture design. This is because, as seen in Figure 4.4, PathFinder does not update the present sharing of any node until it has found a complete route from a source to sink. Thus, an exploration will not feel the effects of present sharing between the phase zero and phase one routes until after it has already completed the search. Furthermore, PathFinder cannot update present sharing during a routing exploration itself since a phase one search has no efficient way of distinguishing between when it is wrapping back onto itself versus attempting to explore a node that was used at latency zero by an unrelated exploration. Likewise, this problem cannot be resolved by history cost, the h_n term in Equation 4.7. Consider a symmetrical architecture as shown in the right side of Figure 7.6. Here, the self-intersecting problem will simply alternate between the top and bottom loop, never realizing that a valid alternative exists.

This characteristic makes PipeRoute unsuitable for many FPGA architectures. First, the interconnect flexibility of modern devices will encourage the self-intersecting path problem. In other words, the generally high connectivity within the communication fabric allows a routing node to easily re-discover itself after going through a register. If modern FPGAs were purely directional devices (outputs could only drive inputs that were to the right or below a given location, for example), this might not be a problem. Second, the majority of interconnection networks have a great deal of symmetry. One routing track is

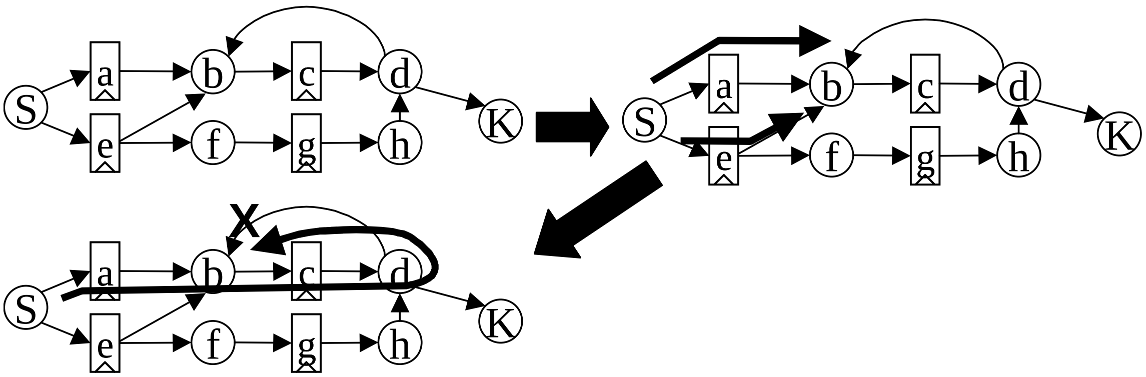


Figure 7.7: QuickRoute and Self-Intersection

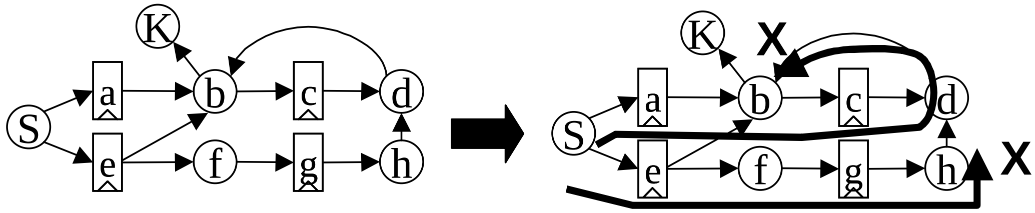


Figure 7.8: QuickRoute and Self Blocking

likely to have the same access to pipelining resources as neighboring tracks. This will encourage explorations to fold all available options back onto themselves and prevent valid non-overlapping routes from being found.

7.2.2: QuickRoute

QuickRoute [23] was the second heuristic to address the *N*-delay routing problem. Like PipeRoute, it also retains an outer loop of PathFinder congestion resolution and simply replaces Dijkstra's algorithm to perform the inner loop searches. However, unlike PipeRoute it attempts to find full *N*-latency paths directly. Although performed for latencies larger than one, it is similar to the Combined-Phased BFS from PipeRoute in that the router must record the phase of an exploration when a node is visited. In QuickRoute, a wave is allowed to explore a given node if the node has been visited by fewer than *k* other waves at the same latency. For example, in the top right figure of Figure 7.7, if the router is trying to find a 2-register path from *S* to *K*, assuming *k*=2, the paths (*S*, *a*, *b*) and (*S*, *e*, *b*) would both be considered. However, unlike PipeRoute, QuickRoute does not allow paths to intersect themselves. To accomplish this, it records the path back to the source for every exploratory wave and does not allow an exploration to revisit a node already used by itself earlier in the search. In the bottom left illustration of Figure 7.7, a path that goes through node *b* will not consider it again for subsequent exploration. This multi-latency search process is continued until the sink is discovered at the appropriate latency.

```
QuickRoute
while(!all signals routed || congestion exists)
    for all nets N
        if N is nota  pipelined net, use PathFinder
        else
            clear N.routing tree
            put source of N into N.routing tree
            sort sinks by non-decreasing latency
            for all sinks of N
                for all nodes in architecture, for all latencies L, set visited[L] = 0
                put all nodes in routing tree into priority queue PQ at cost C, path P, latency L
                while(PQ.head not sink[i] of N && PQ not empty)
                    remove head of PQ H at cost C, path P, latency L
                    if(H.visited[L] < k)
                        set H.cost[L] to C
                        add H to P
                        increment H.visited[L]
                        if (neighbor of H is  not register && neighbor of H.visited < k && neighbor not in P)
                            put neighbor into PQ at cost C + neighbor cost + edge cost, path P, latency L
                        else if (neighbor of H is register && neighbor of H.visited < k && neighbor not in P)
                            put neighbor into PQ at cost C + neighbor cost + edge cost, path P, latency L+1
                        end if
                    end if
                end while
                if(PQ is empty)
                    net unroutable, exit
                else if(PQ.head is sink[i] of net N)
                    mark sink found
                    add new parts of P to N.routing tree
                    clear PQ
                    update cost of congested nodes
                end if
            end for
        end if
    end for
    update critical path delay and sink criticalities
end while
```

Figure 7.9: Pseudo-Code for QuickRoute

Of course, since the problem is still NP-Complete, QuickRoute cannot guarantee a solution. For example, if a slight modification is made to the routing graph, as in Figure 7.8, the router will run into problems. Assuming *k*=1 and the router needs to go from *S* to *K* accumulating two registers, it will fail to find a solution. This is because node *b* is initially used by the doomed route through (*S*, *a*, *b*, *c*, *d*) that, in turn, prevents the correct route through (*S*, *e*, *f*, *g*, *h*) from exploring node *d*. Unfortunately, no matter how large *k* is made, it is possible to construct a routing graph that will cause QuickRoute to self-block by adding additional 1-register paths between *b* and *d*.

However, QuickRoute still holds multiple advantages over PipeRoute. Not only does QuickRoute defend itself from the self-intersection problem of PipeRoute, it has the flexibility to improve its routing ability by increasing the *k* factor. Pseudocode for QuickRoute is shown in Figure 7.9.

7.3: Timing-Driven Pipelined Routing

Although both PipeRoute and QuickRoute do address the basic *N*-Delay routing problem, they also share a critical shortcoming: neither implements timing-driven routing. This is surprising for two reasons. First, registers are generally added to a netlist because the application developer seeks better critical path delay. Thus, not considering the timing concerns of an application during the routing process can nullify much of the advantags these registers might provide. Second, as discussed in Chapter 4, PathFinder already has a timing-driven mode that simultaneously balances congestion and delay very nicely. However, PipeRoute and QuickRoute cannot leverage PathFinder's timing-driven formulation because of multiple differences between the conventional routing problem and the pipelined routing problem. The following sections will discuss the nature of these issues and introduce some new solutions.

7.3.1: Determining Link Criticality

Returning to Equation 4.8, Pathfinder determines the cost of a path based upon A_{ij}, the criticality of the source/sink pair as found during the last routing iteration. One key problem that prevents previous pipelined routing algorithms from using PathFinder's timing-driven methodology stems from the fact that, in the classical CAD sense, they continuously change the very nature of the netlist during the routing process. This makes using the criticality information from one routing iteration in the next unreliable.

As shown in top illustration of Figure 7.10, conventional CAD tools map registers to logic block locations. Since the placement process determines the location of all the blocks before routing begins, it can achieve relatively consistent iteration-to-iteration net criticality. This allows the classical PathFinder cost formulation to function well. In this example, the placement tool has decided that CLB *a* must route to CLB *b* before going to CLB *c*. As routing progresses, Pathfinder can use the criticality of the last route found to determine the next route. In this way, PathFinder relies on the fact that the routing will not drastically change between iterations. In other words, it assumes that it is unlikely that consecutive routing iterations will choose vastly faster or slower routes from *a* to *b* or *b* to *c*. However, if this does occur, the router will over or under-penalize the congestion versus delay contribution to the overall path cost. For example, if the last routing iteration resulted in a timing-critical path for the link from *a* to *b*, but the present routing iteration manages to find a much faster path, the cost of the route will greatly over-penalize delay while erroneously ignoring congestion.

Pipelined routing differs strongly from classical routing because it must find the location of registers in the netlist during the routing process. These registers are not locked into position by the placement tool. While this is a hard problem in itself, it also presents a completely new issue for timing optimization. Since registers are the start and end points of a clock cycle, their placement is naturally very important to the timing of the nets to which they are attached. However, since a pipelined router determines the placement of at least some portion of the registers in the netlist during routing, the timing significance of a given net can change dramatically depending on the location chosen by a given routing iteration. Looking at the pipelined routing problem from the standpoint of conventional routing, it is as if the placement of all the

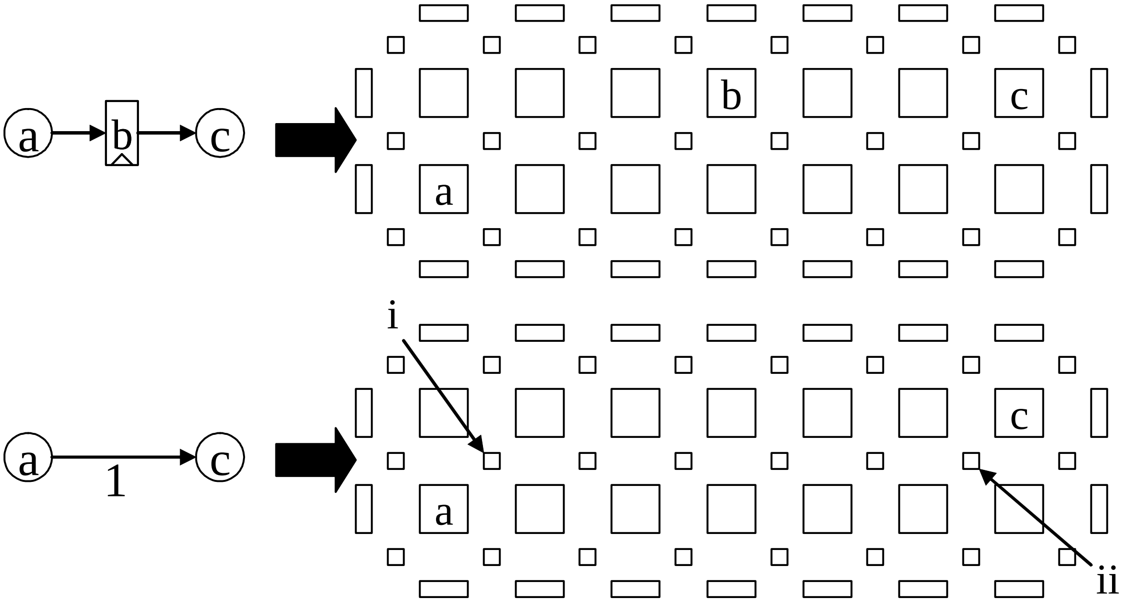


Figure 7.10: Timing Implications for Conventional Routing Versus Pipelined Routing

registers that need to be found during routing can change every routing iteration. This makes it very easy to use the wrong criticality value and over or under-penalize the congestion versus delay contribution of the overall path cost.

Consider the same netlist used before, but in a pipelined routing framework. This is shown in bottom illustration of Figure 7.10. Notice that the register has been replaced by a latency annotation on the edge between *a* and *c*. In this situation, LUT *a* must be connected to LUT *c* by a single latency link, but the router must find the register as part of the routing process itself. However, the criticality of the individual links between *a* and the register and the register and *c* will heavily depend upon the registering location that is chosen. The relative criticality of these links will change completely if the router chooses to register at *i* versus *ii*. However, the system cannot anticipate this change between routing iterations, so it can only follow the classical PathFinder methodology and forward criticality information calculated in one iteration for use in the next. However, potential inaccuracy regarding the criticality of the nets will result in possibly grossly miscalculating the true cost of a path. Ultimately, this will lead to timing oscillations as opposite sides of a register along critical or nearly critical paths vie for dominance.

If the first iteration chooses to register at *i*, the second iteration will choose to register at *ii*, despite that fact that it would be more advantageous, from a timing standpoint, to select a register closer to the center of the array. This problem occurs because the pre-register link will have a very low criticality, making delay on this segment during the next routing iteration very inexpensive. Conversely, the post-register link will have

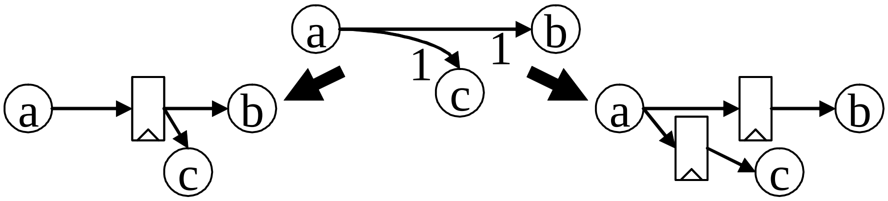


Figure 7.11: Multi-Terminal Criticality Problem

a very high criticality, making delay very costly during the next routing iteration. Thus, ignoring congestion for the moment, the post-register link will want to become as short as possible at the expense of the pre-register link. For similar reasons, a third routing iteration will return to the register at *i*. This means that the router will alternately select equally poor register locations and never find a better solution.

Essentially, this type of behavior occurs because the router utilizes old, and dramatically incorrect net criticality information to determine future routes. The criticality of a link to a register used in one routing iteration has little relevance in the next if the router selects a different register. Notice that the mismatch that occurs between the real criticality of a link and the criticality used for calculating the cost of a path is very reminiscent of the problem encountered during the placement of registered netlists discussed in Section 5.4. For that matter, the fundamental cause of this problem is also the same: the technique that conventional timing-driven routers use implicitly assumes that the criticality of any connection in the system will not change significantly between routing iterations. However, if the criticality does change significantly, the algorithm can produce degenerate solutions.

This problem becomes even further complicated considering multi-terminal and multi-latency nets. As shown in Figure 7.11, there are certain situations in which sinks may want to share registers to reduce congestion. However, depending upon their relative placements and if this net becomes critical or near critical, each sink might wish to use a separate register. Unfortunately, it becomes unclear what criticality to assign any of the nets to allow these “zipped” and “unzipped” paths to exist in consecutive iterations and still produce high-quality results. Should the criticality of all latency-*N* segments be averaged? Should the worst criticality of any segment define the criticality of all links? This becomes an issue because the router can fundamentally changing the nature of the netlist during routing. Similar to before, from the viewpoint of a conventional router it is as if a limited form of logic synthesis or, at the very least, register duplication can be performed between every routing iteration.

7.3.2: Assumed Criticality Searching

Clearly, if a pipelined router is to obtain high quality results, it cannot use criticality information gleaned from previous routing iterations to guide future exploration. However, PathFinder has shown that there still needs to be some mechanism to allow more timing-significant links to trade higher congestion for lower delay, and less important signals to trade additional delay for less congestion. A potential solution? Allow each exploration to discover its own criticality.

While the router would normally obtain the timing importance of the signal from the previous routing iteration, this cannot be done for pipelined signals. One possible alternative is for an exploration to build its own criticality based upon the delay it has seen thus far. In this scenario, the router would start with a very low criticality at the source when the exploration has not accumulated any delay, and gradually increase the timing significance as the search continues and paths becomes slower. Unfortunately, while this may work for low and mid-criticality links, this will not perform well on high criticality segments. This is because the early portion of searches may meander to avoid congestion. As the path becomes longer, the search will opt for more direct routes to the sink. Unfortunately for critical nets, the damage has already been done and they will never obtain the congestion-blind routes that they should.

Instead, it is possible for an exploration to decide the proper criticality for a route at the only point that the decision can actually be made – when it arrives at a sink. In this formulation, the router starts *AC* independent waves from the source, each assuming the net has a different criticality, ranging from 1/*AC* to 1.0. In this manner, the system will have multiple simultaneous searches that each emphasizes delay versus congestion in a slightly different way. The first exploration to reach the sink will be the least expensive and, thus, represent approximately the proper balance of congestion versus delay. Furthermore, the router can trade runtime for further timing accuracy or vice versa by adjusting *AC*. This technique is called an *Assumed Criticality Search*.

However, assumed criticality searching could still lead to grossly incorrect routing. Looking back at Equation 4.8, this is because high criticality nets always emphasize low delay and low criticality nets always emphasize low congestion. This relationship makes it possible for assumed criticality searches to degenerate to always selecting either the lowest or highest assumed criticality for all nets. For example, if the delay values along most paths from the source to the sink are coincidentally smaller in magnitude than their congestion counterparts, searches that assume a criticality of 1.0 will always be the cheapest, regardless as to whether they are truly timing critical. A similar situation occurs for the minimum assumed criticality if the relative values are reversed. While this problem could be addressed by ensuring that the delay and congestion values are always balanced, this is not a feasible solution as the congestion values must be able to grow as the routing progresses – PathFinder relies on gradually escalating congestion costs to resolve sharing.

To deal with this problem the assumed criticality router needs to incorporate the real criticality of a path back into the cost calculation. This can be accomplished by using the assumed criticality values to calculate the cost of route up to, but not including, the sink or register. Then, just as the router reaches this node, it can determine the real criticality of the route that it actually found. At this point the router can re-calculate the cost of the path based upon the actual criticality. This will ensure that a sink is only pushed into the search queue with the true cost of the path. This will prevent the scenario in which low assumed criticality searches rush ahead along uncongested, but slow links and form an unnecessarily high criticality path. This is because just as these searches are about to reach the sink or register, they will calculate the real criticality of the paths found. The cost of these searches will then rise dramatically to reflect their newly revealed high delay and high criticality. This will allow higher assumed criticality searches, which will presumably find faster, slightly more congested paths, to catch up and have the opportunity to form a more appropriate mid-criticality link.

The complete assumed criticality search methodology, as seen in Figure 7.12, has several attractive features. First, it solves the problem of routing inaccuracy due to iteration-to-iteration variance in path criticality. Second, this approach does not dramatically increase the computational effort of routing. Obviously, if the router conducted *AC* completely independent searches for each source/sink pair, this would only invoke PathFinder's inner loop *AC*-1 additional times. However, the router can also easily run all of these searches simultaneously and prune non-productive explorations along the way. Of course, once one search has reached the sink, the router can end all exploration. However, it can even prune incomplete

```
Assumed Criticality Breadth-First Search
for i = 1 to AC
    put source into priority queue PQ at cost = 0, crit= i/AC
end for
while(PQ.head not sink && PQ not empty)
    remove head of PQ H at cost C, crit CR, previous node P
    if(H not visited at crit[CR])
        mark H visited at crit[CR]
        set H.cost[CR] to C
        set H.previous node[CR] to P
        for each neighbor of H
            if neighbor is not sink
                if CR != 1.0 && neighbor.delay > (CR + 1/AC) * critical path
                    continue
                else
                    put unvisited neighbor of H into PQ at cost C + neighbor cost + edge cost, crit CR,
                        previous node H
                end if
            else if neighbor is sink
                calculate actual criticality of current path
                recalculate cost of path
                put sink into PQ at updated cost, crit CR, previous node H
            end if
        end for
    end if
end while
if(PQ is empty)
    sink is unroutable, exit
else if(PQ.head is sink)
    add path net's routing tree
end if
```

Figure 7.12: Assumed Criticality Searching

searches. For example, for *AC*=5 the router will launch five explorations with criticalities (0.2, 0.4, 0.6, 0.8, 1.0). If the current critical path is 10, paths with a delay of 4 or more do not need to be explored by the 0.2 assumed criticality wave. Those paths will be better serviced by the 0.4 assumed criticality exploration. Thus, with the exception of the highest criticality wave, the router can prune a search when the current path delay would make the exploration's criticality larger than the next higher assumed criticality search.

7.3.3: New Cost Formulation

Another issue that appears concerns the congestion versus timing cost formulation itself. As mentioned in the previous section during the discussion of the potential pitfalls of the assumed criticality methodology, the lowest cost path obtained by using Equation 4.8 heavily depends upon the relative values of an architecture's delay and congestion costs. Unfortunately, this can cause further undesirable behavior when considering pipelined routing.

Consider the scenario in Figure 7.13. Here, there are two potential one-latency paths from *S* to *K*. If the notation in the figure is (*delay cost*:*congestion cost*), and the cost of a path is considered to be the some of the congestion and delay costs of all of it links, both paths have the same total congestion and delay. However, the top path is a comparatively poor choice because the post-register path is both highly critical and highly congested. Using Equation 4.8 as a cost function, the cost of the top and bottom paths are shown in Equations 7.1 and 7.2, respectively. For mathematical simplicity, the critical path delay of the system is assumed to be 10*d* for the moment. The effect of system critical path delay will be further examined in Section 7.6.

$$0.1(d) + 0.9(c) + 0.9(9d) + 0.1(9c) = 8.2d + 1.8c \qquad (7.1)$$

$$0.5(5d) + 0.5(5c) + 0.5(5d) + 0.5(5c) = 5d + 5c \qquad (7.2)$$

Based on these equations, the selection of balanced versus unbalanced paths depends entirely upon the relative values of c and d, an architecture's average base congestion and delay cost. In this example, the more balanced path is only selected if c < d. However, maintaining this relationship is very difficult. Even if the router were to scales the base cost of all routing nodes so that it initially selected more balanced paths, the natural congestion cost escalation of PathFinder will cause later iterations to tend toward worse selections. Not only do these unbalanced paths create a more difficult timing problem, they actually work contrary to PathFinder's own attempts at congestion resolution. This is because as the router enters the later stages of routing, the average congestion cost will rise to resolve sharing. However, based upon the observation here, the router will actually tend towards more extreme congestion options.

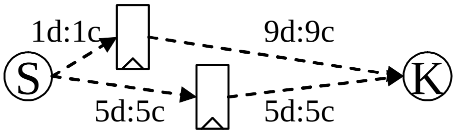


Figure 7.13: Congestion vs. Timing Concerns For Pipelined Routing

This problem occurs because the delay and congestion contributions to the overall path cost are linked. While the A_{ij} versus (1-A_{ij}) terms guarantee that paths can trade delay for congestion and vice-versa, this intertwines the two components, making their relative values very sensitive. To address this issue, a subtle change can be made that removes this vulnerability. Equation 7.3 is obtained by dividing both sides of Equation 4.8 by (1-A_{ij}).

$$\frac{C_n}{1-A_{ij}} = \frac{A_{ij}d_n}{1-A_{ij}} + \frac{(1-A_{ij})c_n}{1-A_{ij}} = \frac{A_{ij}}{1-A_{ij}} d_n + c_n \qquad (7.3)$$

While this change scales all path costs by 1/(1-A_{ij}), since all explorations compete with each other simultaneously, this likely does not change path selection for conventional non-pipelined routing. However, this does change the behavior for pipelined signals. Revisiting the example from Figure 7.13 but substituting the new cost formulation, the cost of the unbalanced top path and balanced bottom path are shown in Equations 7.4 and 7.5, respectively.

$$\frac{0.1}{0.9}(d) + c + \frac{0.9}{0.1}(d) + 9c = 0.11(d) + c + 9(9d) + 9c = 81.11d + 10c \qquad (7.4)$$

$$\frac{0.5}{0.5}(5d) + 5d + \frac{0.5}{0.5}(5d) + 5d = 1(5d) + 5c + 1(5d) + 5c = 10d + 10c \qquad (7.5)$$

Since both the congestion and delay costs are necessarily positive numbers, more balanced paths are now always selected over unbalanced paths without the need to meticulously adjust the relative values of an architecture's congestion and delay costs. However, the router still has the option of selecting the unbalanced path should this path become less congested in future routing iterations.

One concern that might arise regarding Equation 7.3 is that the criticality of a connection, A_{ij}, is divided by 1 minus the criticality, or (1-A_{ij}). This term could become undefined for connections that are along the critical path since A_{ij} is 1.0, resulting in a division by zero. However, this does not occur because timing-driven routers generally cap the criticality used to calculate routing costs to 0.99 [2]. Looking back at Equation 4.8, the reason that the criticality is only allowed to reach a maximum of 0.99 is because routers do not want to create situation in which paths can entirely ignore congestion. If A_{ij} were equal to 1.0, (1-A_{ij}) would equal zero, allowing a path to solely focus on delay with absolutely no concern for congestion. Thus, two critical paths that fight over a single resource would never be able to resolve their conflict. This cap on a path's criticality helps the router better resolve congestion.

This limit also has an effect on routes slower than the current critical path. For example, it is possible that in an attempt to resolve congestion, the system considers a slower route for some connections. Without a cap on the criticality of a connection, the assumed criticality methodology could find a route with a criticality larger than 1.0, causing the congestion term to become negative. This might actually cause the system to use highly congested paths, just to receive the cost benefit. However, with the limit in place, the router can still feel the effects of slower paths since the delay term is larger, but without potentially creating a problem for congestion resolution.

7.4: Armada

The assumed criticality search technique and the new cost function can be integrated into the QuickRoute algorithm. This new pipelined routing algorithm is called Armada [10]. As shown in Figure 7.14, Armada launches a series of multi-criticality searches from the source. In this example the router would like to find a one-latency path between S and K. The first series of searches expand from the source. When one of these waves encounters a register, it recalculates the path cost based upon the real criticality required to reach the register along the given path. When the cheapest path to the register is popped from the priority queue, it launches a new series of assumed criticality searches of its own at latency one. Notice that although all zero-latency searches may reach the register and push it into the priority queue, only one path will be deemed the least expensive and, thus, the best way to use this particular register. Only this path will continue on with one-latency explorations.

However, this example brings up the issue of defining the cost of a multiple latency route. In Figure 7.14, eventually both registers in the architecture will launch their own set of one-latency explorations. As they near K, the router needs to determine which path best balances not only the congestion and delay of their zero and one-latency paths individually, but the combination of the two. Since each time the router encounters a register it determines the actual criticality of the link, the cost of an L-latency path can be

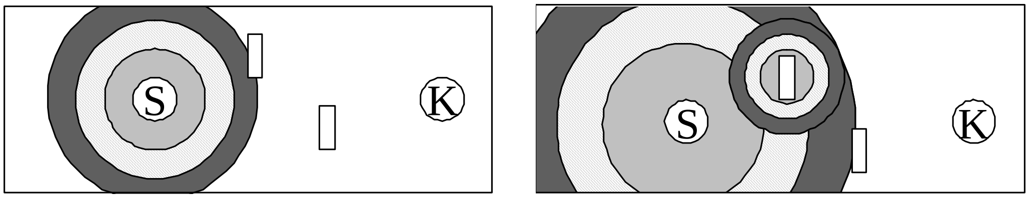


Figure 7.14: QuickRoute with Assumed Criticality Searching

defined as the total of the timing and congestion costs of all zero to L-latency segments. This is shown in Equation 7.6.

$$C = \sum_{i=0}^{L} (\text{timingCost}_i + \text{congestionCost}_i) \tag{7.6}$$

Furthermore, as seen in line 6 of the pseudocode in Figure 7.16, Armada borrows a concept from QuickRoute and sorts the sinks of each net it is responsible for routing. To give priority to higher criticality links, it sorts each net's sinks first by non-decreasing order of latency (# of registers required on the path), then by non-increasing order of maximum link criticality found in the previous routing iteration. In this way, the most timing-critical sinks with the fewest chances to amortize path delay over multiple clock cycles determine the earliest stages of the routing tree.

To build successive multi-terminal routes, Armada must also define how pre-existing routes should initialize the priority queue. As seen in Figure 7.15, after the router has found a one-latency route to *K*, the router pushes this existing route into the priority queue to reflect all of the possible routing options to the 2-latency sink *J*. This can be seen in lines 9-17 of the pseudocode in Figure 7.16. While building a link from *b* would allow for the maximum register sharing and will likely cause the minimum congestion impact, developing a wholly new path may offer some timing benefits. Borrowing a concept from timing-driven PathFinder, Armada considers existing routes to be free in terms of congestion, and it only consider their delay impact on further sinks. Based upon the model discussed in Equation 7.6, Armada pushes nodes along existing routes into the priority queue by summing only the timing cost of all upstream zero to L-latency segments. For the example in Figure 7.15, to combine this concept with the assumed criticality searching technique, all nodes along *a* would be pushed into the priority queue *AC* times using different assumed criticalities to determine their timing cost. While all nodes along *b* would also be added to the priority queue *AC* times, they would all share some common portion of their cost – the zero-latency timing cost incurred along *a*.

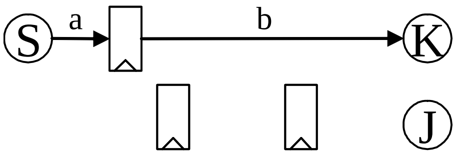


Figure 7.15: Re-initializing PQ for Multi-Terminal Nets

```
Armada
while(!all signals routed || congestion exists)
    for all nets N
        if N is not pipelined net, use PathFinder
        else
            clear N.routing tree
            put source of N into N.routing tree
            sort sinks by non-decreasing latency, non-increasing criticality
            for all sinks of N
                for all nodes in architecture, for all latencies L, for all assumed criticalities CR set visited[L][CR] = 0
                Initialize priority queue PQ with existing routing tree
                for all CR = 1/AC to 0.99
                    for all nodes X in routing tree
                        if CR != 0.99 && X.delay > (CR + 1/AC) * critical Path
                            continue            // prune search for starting points
                        else
                            put X into PQ at cost C, path P, latency L, assumed criticality CR
                        end if
                    end for
                end for
                while(PQ.head notsink[i] of N && PQ not empty) // search for L-latency route to sink
                    remove head of PQ H at cost C, path P, latency L, assumed criticality CR
                    if(H.visited[L][CR] < k)
                        set H.cost[L][CR] to C
                        add H to P
                        increment H.visited[L][CR]
                        for each neighbor of H
                            if neighbor is not sink
                                if CR != 1.0 && neighbor.delay > (CR + 1/AC) * critical path
                                    continue            // prune searches
                                else if neighbor of H.visted ≥ k || neighbor in P
                                    continue            // don’t explore visited or loopback neighbors
                                else if (neighbor of H is  not register)
                                    put neighbor into PQ at cost C + neighbor cost + edge cost, path P,
                                        latency L,  assumed criticality CR
                                else if (neighbor of H is register)
                                    calculate actual criticality of current path
                                    recalculate cost of path
                                    put neighbor into PQ at updated cost + neighbor cost + edge cost, path P,
                                        latency L+1, assumed criticality CR
                                end if
                            else if neighbor is sink
                                calculate actual criticality of current path
                                recalculate cost of path
                                put sink into PQ at updated cost, path P, latency L, assumed criticality CR
                            end if
                        end for
                    end if
                end while
                if(PQ is empty)
                    net unroutable, exit
                else if(PQ.head is sink[i] of net N)
                    mark sink found
                    add new parts of P to N.routing tree
                    clear PQ
                    update cost of congested nodes
                end if
            end for
        end if
    end for
    update critical path delay
end while
```

Figure 7.16: Pseudo-Code for Armada Timing-Driven Pipelined Routing

7.5: Testing and Results

As described in [34], the pipeline routing problem was first inspired by the RaPiD [9] architecture. Thus, to determine the effectiveness of the Armada algorithm it was tested with RaPiD architectures and RaPiD netlists. RaPiD is a coarse-grain, one-dimensional reconfigurable array with a word-width interconnect network. As seen in Figure 7.17, logic blocks populate the top of the array with a mixture of short and long-distance routing wires below. Although short wires cannot be concatenated to make longer routes and are not connected to specialized interconnect registers, long wires can be concatenated for up to chip-wide routes and can acquire between zero and three register latencies at each switchpoint, also known as a *bus connector*. Bus connectors are represented with small squares between long wire segments. Furthermore, multiple RaPiD cells can be abutted side by side to construct larger arrays.

In the existing RaPiD toolflow, a high-level language compiler produces a retimed netlist that must be mapped to a device given specific latency requirements on each connection. Although RaPiD architectures contain a wealth of register locations, any specific bus connector can only communicate with the wires immediately to its left and right. Because of this connectivity, register assignment cannot be performed during placement. This is because, like the registered track-graph architecture in [38], deciding exactly which registers should be used for a given signal also mostly determines the detailed routing for that net. Unfortunately, deferring register assignment until routing also presents a problem since it is not obvious how to find routes that contains exactly the correct number of pipelining registers. A conventional router cannot be used because the architecture has limited pipelining resources that determine the overall characteristics of each path. For example, logic blocks that are placed physically close to each other may not be able to be connected via the most direct route. If the connection between these blocks requires multiple pipelining delays, the router may need to take a more circuitous path to acquire sufficient registering.

Testing was performed using nine RaPiD netlists that represent a wide range of pipeline register requirements. These netlists, detailed in Appendix A, were mapped to three different RaPiD architectures: the original architecture that contains 16 logic blocks per cell, length-4 short wires, length-16 long wires, and three optional registers at each bus connector, and two other architectures that are similar, but substitute long wires of length 8 and 4.

The Armada router was compared to both PipeRoute and QuickRoute. PipeRoute was represented by a slightly augmented version from [32] that added a rudimentary timing-driven formulation to the original PipeRoute algorithm. In the new PipeRoute methodology, the maximum criticality encountered by any link between a given source and sink determined the overall net criticality during the following routing iteration. Of course, this technique introduces some inaccuracies into the system. Not only does this

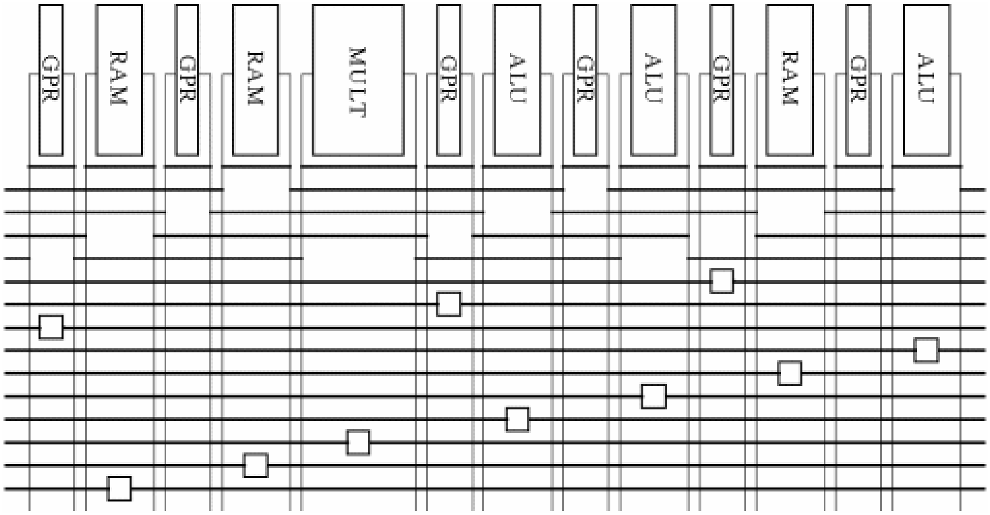


Figure 7.17: Illustration of a RaPiD Cell

methodology suffer from the problem associated with determining the correct relative cost between congestion and timing that inspired the modified cost formulation, it also suffers from the false link criticality predictions that was addressed with the assumed criticality approach. As for QuickRoute's *k* term, as suggested by [23], *k* = 1 was used. Armada also used *k* = 1 and arbitrarily set AC = 10 for the initial round of testing.

Before the quality of these routers could be evaluated, they required the benchmark netlists to be placed. All nine netlists were placed using the placement tool built into PipeRoute [33]. This provided a fixed, pipelining-aware placement as a starting point for all three algorithms. While conventional placement tools always attempt to group interconnected blocks as closely as possible, this is not necessarily favorable on architectures that require pipelined routing such as RaPiD. This is because, as mentioned earlier, high latency connections may need to take a circuitous route if there are not enough pipelining resources between the logic blocks to acquire the appropriate registering. The PipeRoute placer attempts to take this into account by explicitly placing both logic blocks and registers during annealing. However, unlike a conventional placement tool, the placement of the registers in the system is not binding and new register locations are determined during the routing process.

Testing began with the original RaPiD architecture. Six independent PipeRoute placement and routing runs were performed, and the placement with the lowest routed critical path delay as found by PipeRoute was passed on to evaluate the other routers. These placements were routed using congestion-driven QuickRoute, the Armada algorithm, and the Armada algorithm with the original PathFinder cost formulation substituted in.

In Table 7.1 to Table 7.3, the *Best Track Count* results are the average normalized track requirements, circuit timing and router runtime when each tool searched separately for the minimum routable architecture for each of the nine netlists. Notice that this is slightly different than the testing used to evaluate the placement tools from the previous chapters, but provides good insight into the true quality of the routing algorithms. *Match PipeRoute Track Count* results were obtained when each tool was given the same number of tracks that PipeRoute required for a given netlist. *Match QuickRoute Track Count* results were obtained when each tool was given the maximum number of tracks required by any of the QuickRoute-derivative tools (QuickRoute, Armada or Armada with PathFinder's cost function) for a given netlist. *Match QuickRoute Track Count* results do not include results for PipeRoute as the available codebase does not allow the placement and routing steps to be separated. Given a different architecture, PipeRoute will also change the placement.

Although a precise relationship cannot be made due to the wide range of benchmark complexity, these tables also include un-normalized average router runtime to give a general sense of algorithm effort. All results were gathered on 3.2GHz Intel Xeon machines with 2GB of RAM. Unfortunately, runtime is only reported for the three QuickRoute-derivative routers because differences in code execution prevented meaningful comparisons to be made with the PipeRoute codebase. This said, the original QuickRoute algorithm is likely to perform as fast or faster than PipeRoute since it does not perform multiple piecewise searches.

As seen in Table 7.1, the first surprise is that the original congestion-driven QuickRoute algorithm actually achieves nearly the same critical path delay as the improved timing-driven PipeRoute formulation. QuickRoute produced a normalized critical path delay of 1.64x while PipeRoute's critical path delay was somewhat faster with a 1.56x critical path delay. Although based upon the tests performed in [23] one would expect QuickRoute to provide marginally better track counts than PipeRoute, the very similar timing results indicate that the technique used to make PipeRoute timing-driven is largely ineffective. As predicted, it is likely that inaccuracies within the timing-driven formulation itself greatly limit its ability for optimization.

In contrast, though, Armada finds vastly superior timing results with slightly better routability. PipeRoute produced 1.56x worse critical path with 1.09x worse track count and QuickRoute produced 1.64x worse critical path delay with 1.04x worse track count. This improvement in track count is likely due to the fact that the timing-driven cost formulation provides additional direction to the QuickRoute-like searches, avoiding some occurrences of self-blocking. However, as expected given the *AC* = 10 factor, Armada runs approximately 10x slower than QuickRoute. Furthermore, it is also clear that the new timing-driven cost formulation functions as intended. When PathFinder's cost function is substituted back into the Armada algorithm, it produces 1.18x worse critical path delay.

Of course, it may be unfair to compare the critical path delay of netlists mapped to architectures with different track counts. Thus, as seen in the bottom two sections of Table 7.1, testing was repeated using the same architecture for all of the routing algorithms. However, the results are largely the same – Armada still produces vastly superior critical path delay compared with all of the other approaches, but requires approximately 10x the runtime of QuickRoute.

Table 7.1. Normalized Results for Length-16 Long Wire Architecture

Best Track Count	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
PipeRoute-TD	1.09	1.56	-	-
QuickRoute	1.04	1.64	0.10	133 s
Armada	**1.00**	**1.00**	1.00	1721 s
Armada, PathFinder Cost	1.03	1.18	7.65	7982 s
Match PipeRoute Track Count	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
PipeRoute-TD	1.09	1.56	-	-
QuickRoute	1.09	1.75	0.08	113 s
Armada	**1.09**	**1.00**	0.94	1752 s
Armada, PathFinder Cost	1.09	1.19	5.21	5329 s
Match QuickRoute Track Count	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
QuickRoute	1.05	1.73	0.11	138 s
Armada	**1.05**	**0.99**	1.05	1826 s
Armada, PathFinder Cost	1.05	1.20	7.45	7705 s

All results normalized to the Armada results with the smallest track count

Table 7.2. Normalized Results for Length-8 Long Wire Architecture

Best Track Count	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
PipeRoute-TD	1.00	1.66	-	-
QuickRoute	0.96	1.65	0.10	66 s
Armada	**1.00**	**1.00**	1.00	1357 s
Armada, PathFinder Cost	1.02	1.30	3.11	3068 s
Match QuickRoute Track	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
QuickRoute	1.03	1.71	0.08	45 s
Armada	**1.03**	**1.00**	0.88	841 s
Armada, PathFinder Cost	1.03	1.31	3.12	3075 s

All results normalized to the Armada results with the smallest track count

Table 7.3. Normalized Results for Length-4 Long Wire Architecture

Best Track Count	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
PipeRoute-TD	1.01	1.59	-	-
QuickRoute	1.02	1.54	0.11	76 s
Armada	**1.00**	**1.00**	1.00	2637 s
Armada, PathFinder Cost	1.05	1.21	2.75	2976 s
Match QuickRoute Track	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
QuickRoute	1.05	1.55	0.10	41 s
Armada	**1.05**	**0.99**	0.84	1593 s
Armada, PathFinder Cost	1.05	1.21	2.75	2976 s

All results normalized to the Armada results with the smallest track count

As seen in Table 7.2 and Table 7.3, this trend continues when the netlists are mapped to architectures that present a more difficult pipelined routing problem. The testing methodology used on the original RaPiD architecture was repeated on architectures with double and quadruple the number of pipelined switch opportunities. On the length-8 architectures, PipeRoute and QuickRoute produce 1.66x and 1.65x worse critical path delay respectively. On the length-4 architectures, PipeRoute and QuickRoute produce 1.59x and 1.54x worse critical path delay respectively. As a note, since the gap between the track counts of PipeRoute and the QuickRoute-derivatives mostly closes, the *Match PipeRoute Track Count* results are no longer shown.

Although this testing proved that Armada produces significantly better pipelined routing results than its predecessors, there are two other outstanding questions regarding its effectiveness. First, as mentioned earlier, the maximum visitation factor used in this initial testing was suggested by the original QuickRoute paper (k = 1). Even though the routing algorithm is still operating within the same architectural framework, the timing-driven nature of the Armada approach might make more thorough explorations attractive. As seen in Table 7.4, there is some correlation between larger values of k and higher quality results, but the change is relatively minor. The small potential improvement in critical path delay (up to 0.95x) or track count (up to 0.98x) is likely not worth the increase in algorithm runtime. However, since larger values of k primarily help combat self-blocking, this behaviour is probably highly architecture-specific.

The second issue is that the number of assumed criticality searches that were performed in the initial round of testing was completely arbitrarily chosen (*AC*=10). Since the assumed criticality entirely controls how paths weigh congestion versus delay for the majority of a given route, it is likely that the quality of the critical path timing heavily depends upon the granularity of the assumed criticality searches. However, looking at Table 7.5, although there is a marked runtime improvement, dramatically decreasing the number of assumed criticality searches does not necessarily affect the overall quality of the routing. In fact, there is no real decline in quality even if the number of searches is reduced to merely two (only assume criticalities of 0.5 and 0.99). With *AC* = 2, the critical path delay for the original length 16 long wire architecture is 0.97x better with the same track count and a 0.36x shorter runtime, the critical path delay for the length 8 long wire architecture is only 1.04x worse with the same track count and a 0.29x shorter runtime, and the critical path delay for the length 4 long wire architecture is only 1.01x worse with a 1.04x worse track count and a 0.31x shorter runtime.

Although this may seem counter-intuitive, examining the routed results found by Armada more closely, this is likely an artifact of the RaPiD architecture's design philosophy. In almost all cases, the critical path reaches some architectural limit – two to three bus connector-to-bus connector delays or less. Considering that RaPiD was built to be an architecture for heavily pipelined netlists, this should not be particularly

Table 7.4. Normalized Results for Armada, *k*=1, 2, 4

Length-16 Architecture	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
***k* = 1**	1.00	1.00	1.00	1721 s
***k* = 2**	0.99	0.95	1.67	2454 s
***k* = 4**	0.98	1.02	3.24	5634 s
Length-8 Architecture	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
***k* = 1**	1.00	1.00	1.00	1357 s
***k* = 2**	0.99	1.00	1.29	1757 s
***k* = 4**	0.98	1.00	6.48	21725 s
Length-4 Architecture	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
***k* = 1**	1.00	1.00	1.00	2637 s
***k* = 2**	1.01	1.00	1.77	4414 s
***k* = 4**	1.00	0.98	3.93	9452 s

All results normalized to the Armada results with the smallest track count

Table 7.5. Normalized Results for Armada, *AC*=10, 8, 6, 4, 2

Length-16 Architecture	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
***AC* = 10**	1.00	1.00	1.00	1721 s
***AC* = 8**	0.98	1.04	0.96	1168 s
***AC* = 6**	1.00	0.97	0.67	785 s
***AC* = 4**	0.99	1.02	0.52	573 s
***AC* = 2**	1.00	0.97	0.36	354 s
***AC* = 1**	1.11	1.20	0.35	300 s
Length-8 Architecture	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
***AC* = 10**	1.00	1.00	1.00	1357 s
***AC* = 8**	0.98	1.01	0.79	910 s
***AC* = 6**	0.98	1.01	0.43	576 s
***AC* = 4**	1.00	1.01	0.56	692 s
***AC* = 2**	1.00	1.04	0.29	307 s
AC = 1	1.08	1.37	0.32	217 s
Length-4 Architecture	**Tracks**	**Crit. Path Delay**	**Runtime**	**Avg. Runtime**
***AC* = 10**	1.00	1.00	1.00	2637 s
***AC* = 8**	1.01	0.99	0.75	1802 s
***AC* = 6**	0.99	1.00	1.05	1803 s
***AC* = 4**	0.99	0.98	0.60	1014 s
***AC* = 2**	1.04	1.01	0.31	342 s
AC = 1	1.60	1.51	0.67	1080 s

All results normalized to AC=10 values

surprising. Because of this, Armada merely finds exactly the types of routes that the original designers had anticipated. When the router achieves such an extremely low critical path delay, all signals actually become either 50% or 100% critical, making *AC* = 2 work exceedingly well. It is only when AC is reduced to 1 and all signals are considered critical that the router is not accurate. However, as with determining *k*, the *AC* behaviour is also likely highly architecture dependent. The majority of FPGAs do not have the extremely predictable routing characteristics of the RaPiD architecture. Thus, more conventional FPGAs are likely more sensitive to the number of assumed criticality searches.

7.6: Conclusions and Future Research

This chapter delved into the details of a relatively new CAD problem: pipelined routing. FPGA architectures that contain a large number of registers often limit the input and output connectivity of many of them due to area concerns. This architectural characteristic makes efficiently using these registers somewhat difficult since the placement tool cannot assign flip-flops in a netlist to these registers in the traditional manner. This is because mapping a flip-flop to a register with very limited connectivity also largely determines the routing needed to connect this register to the rest of the circuit. This can make the subsequent routing problem much more difficult, particularly if a netlist requires a large number of registers.

One manner of dealing with this issue is to assign register locations during the routing process itself. However, this fundamentally changes the nature of the routing problem because signals must find paths that satisfy an additional constraint – valid paths must traverse a very specific number of registers. This new routing problem is called the *N*-Delay Routing problem. Although there have been two prior research efforts to address the *N*-Delay Routing problem, neither of these heuristics can effectively implement timing-driven routing. Primarily, the timing-driven *N*-Delay Routing problem is difficult because, from the viewpoint of conventional CAD tools, it contains aspects of both register placement and physical re-synthesis that must be solved simultaneously within the normal timing-driven routing problem. Attempting to apply conventional timing-driving methodologies can lead to poor solutions, largely because the criticality of registered connections can change dramatically between different routing iterations.

This chapter suggested two new techniques that address some of the instabilities that can form during the timing-driven pipelined routing process. First, this chapter presented an approach that allows the router to determine the criticality of a given connection without any a priori knowledge. Second, this chapter introduced a new timing-driven cost formulation that guides the router towards better pipelined paths. These two techniques were combined with aspects from previous routers to form the Armada timing-driven pipelined routing algorithm. On three different architectures this algorithm was shown to provide roughly 0.6x better average critical path delay without compromising routability. While more computationally intensive than previous pipelined routing algorithms, Armada remains competitive, especially given the large improvement in circuit timing.

Although these results are promising, looking into the future there is still room for improvement. One concern is the quality obtained using the new cost function. A large portion of Section 7.3.3 was devoted to analyzing the routing problem in Figure 7.13. In this example, the traditional PathFinder cost function was shown to potentially favor paths that had both highly critical and highly congested links over paths that had lower criticality and less congested connections. Which paths were selected largely depended upon the relative cost of an architecture’s average base congestion and delay cost. The new cost function suggested in this chapter was shown to remove this dependency and favor more balanced paths. However, the behavior of this new cost function can change depending upon the critical path delay of the system found during the last routing iteration. It turns out that the new cost function can prefer less balanced paths under certain conditions. Of specific interest is what occurs when the critical path delay of the last routing iteration was relatively low, because this provides some idea of what happens when the router encounters congestion and begins exploring slower paths.

As shown in Table 7.6 and Table 7.7, the same calculations as performed in Equations 7.4 and 7.5 for the example in Figure 7.13 can be repeated for different system critical path delays. The criticality and (1-criticality) terms of the expanded equations on the left sides of 7.4 and 7.5 can be found in Table 7.6 in the CPD = 10*d* row. Similarly, the *d* and *c* multiplier terms on the right side of Equations 7.4 and 7.5 can be found in Table 7.7 in the CPD = 10*d* row.

While, as expected, the new cost function causes the router to always prefer the balanced bottom path when the previous critical path delay was greater than 6*d*, the router will always prefer the unbalanced top path when the previous critical path delay was between 2 and 5 inclusively. Most troubling, this means that when the previous critical path was 5, the router will not find the balanced path that maintains this critical path delay during the next routing iteration. Rather, it will find the unbalanced route that will make the

Table 7.6: Capped Link Criticality of Connections in Figure 7.13

	Unbalanced Top Path				**Balanced Bottom Path**			
	Pre Register		**Post Register**		**Pre Register**		**Post Register**	
CPD	**Crit**	**1-Crit**	**Crit**	**1-Crit**	**Crit**	**1-Crit**	**Crit**	**1-Crit**
1*d*	0.99	0.01	0.99	0.01	0.99	0.01	0.99	0.01
2*d*	0.50	0.50	0.99	0.01	0.99	0.01	0.99	0.01
3*d*	0.33	0.67	0.99	0.01	0.99	0.01	0.99	0.01
4*d*	0.25	0.75	0.99	0.01	0.99	0.01	0.99	0.01
5*d*	0.20	0.80	0.99	0.01	0.99	0.01	0.99	0.01
6*d*	0.17	0.83	0.99	0.01	0.83	0.17	0.83	0.17
7*d*	0.14	0.86	0.99	0.01	0.71	0.29	0.71	0.29
8*d*	0.13	0.88	0.99	0.01	0.63	0.38	0.63	0.38
9*d*	0.11	0.89	0.99	0.01	0.56	0.44	0.56	0.44
10*d*	0.10	0.90	0.90	0.10	0.50	0.50	0.50	0.50
11*d*	0.09	0.91	0.82	0.18	0.45	0.55	0.45	0.55
12*d*	0.08	0.92	0.75	0.25	0.42	0.58	0.42	0.58

CPD refers to the critical path delay of the system during the last routing iteration.

Table 7.7: Effect of Critical Path Delay on Revised Cost Function

	Unbalanced Top Path		**Balanced Bottom Path**	
CPD	**Delay Term * *d***	**Congestion Term * *c***	**Delay Term * *d***	**Congestion Term * *c***
1*d*	990.00	10.00	990.00	10.00
2*d*	892.00	10.00	990.00	10.00
3*d*	891.50	10.00	990.00	10.00
4*d*	891.33	10.00	990.00	10.00
5*d*	891.25	10.00	990.00	10.00
6*d*	891.20	10.00	50.00	10.00
7*d*	891.17	10.00	25.00	10.00
8*d*	891.14	10.00	16.67	10.00
9*d*	891.12	10.00	12.50	10.00
10*d*	81.11	10.00	10.00	10.00
11*d*	40.60	10.00	8.33	10.00
12*d*	27.09	10.00	7.14	10.00

CPD refers to the critical path delay of the system during the last routing iteration.

critical path delay 9. Although the current cost formulation seemed to function well enough in the testing performed thus far, removing this vulnerability may improve Armada's results. While solving this problem will require more extensive investigation, one area that may be worth looking into as a possible solution is re-evaluating the way the existing system saturates net criticality at 0.99.

An additional concern is that the Armada algorithm has only been tested on RaPiD architectures. Unfortunately, RaPiD's routing structure is considerably simpler than more conventional FPGAs. Both the overall number of wires and the interconnect flexibility of the system as a whole is much lower than a traditional island-style FPGA. This leads to several concerns looking into the future, primarily revolving around the runtime of the algorithm.

Although the testing that has been performed so far showed that the assumed criticality search technique was computationally efficient on the RaPiD architecture, this was only because the simple routing resources allowed the use of relatively few different assumed criticalities while still obtaining high quality results. The routability and achievable critical path delay did not truly change even when using only two independent explorations. Although unproductive searches are pruned when possible, the number of independent searches launched has a nearly linear relationship with algorithm runtime. Furthermore, since the assumed criticality completely controls how paths weigh congestion versus delay for the majority of a given route, it is likely that the quality of the router on most FPGA architectures will heavily depend upon the granularity of the assumed criticality searches. Thus, Armada may need to launch far more searches to get similar critical path timing improvement on an architecture with a more sophisticated communication structure. To avoid creating a computationally intractable problem, several alternatives can be explored to lower the computational needs of the system as a whole.

The first possibility is to launch fewer, but more relevant searches. The current algorithm divides the spectrum of criticalities used for exploration into *AC* evenly spaced pieces. However, it is possible that it is sufficient to merely split signals into groups of those that are significant in terms of timing, and those that are not. Thus, instead of launching twenty searches with criticalities evenly spaced from 0.05 to 1.0, it is possible that four searches, perhaps at 1.0, 0.95, 0.5 and 0.1 may be enough to capture the timing and congestion needs of the system.

The second manner of reducing the router's computational needs is to avoid or reduce the size of the pipelined routing problem whenever possible. Even without the assumed criticality methodology, QuickRoute itself is already computationally demanding. Unlike Dijkstra's algorithm, it can visit each node in the graph multiple times – k times at each latency between 1 and L. Thus, the computational needs of the router can be considerably reduced if either the use of Armada is limited outright, or, at the very least, the latency depth of the searches is made smaller.

The use of Amada can be eliminated on some nets entirely. While using conventional placement and fixing the location of some of the registers in the architecture can lead to potential routability problems, many circuits may only have congestion problems in certain local regions of the device. Rather than ignoring the placement of registers throughout the system, it may be possible to analyse the congestion profile of a placement and only use Armada for nets that need to traverse potentially sensitive areas. The remaining nets could be routed using PathFinder since the placement of the registers outside of these regions can be fixed before routing begins.

Furthermore, the number of registers that need to be found on a given net can also be reduced. Although it is likely that the best results will be obtained by using Armada for a full *L* latency path, high latency connections could be broken into shorter, lower latency links. The runtime of most routers is highly correlated with the distance between the source and sink. This is because the searches expand in a wave-like manner and the number of nodes within the search radius for most architectures generally goes up quadratically as the radius is increased. This is particularly important for Armada because the runtime of the router is also affected by the target latency of the sink. Each node between the source and sink can be visited separately by each latency between 0 and *L*. However, if high latency routes are split into multiple sections by fixing the placement of some of the registers along the way, this would create “waypoints” for the router and considerably decrease the runtime. For example, for an 11 register path, the placement of registers 4 and 8 could be fixed. Rather than finding a single, long 11-register path, Armada would only have to find three shorter 3-register paths.

Perhaps the most vital aspect that affects the demands placed on the router is the architecture itself. Pipelined routing can be required on architectures that limit the connectivity of the registers it provides. However, as will be discussed in the next chapter, if a large number of highly connected registers could be efficiently introduced into an FPGA, the need for pipelined routing can be eliminated or drastically reduced.

Chapter 8: Register-Enhanced Architectures

While the previous chapters focused on improving FPGA CAD tools, the target device itself ultimately determines how fast applications will run and how much silicon area they will require. Thus, although pipelining, retiming and C-slowing can greatly improve the performance of an application, this can be largely dependent upon how well the underlying FPGA supports netlists with a large number of registers.

This chapter will focus on how future FPGAs can efficiently incorporate additional registers. This discussion begins by examining how the area requirements and performance profile of a netlist change as it is pipelined or C-slowed. This chapter continues with some background on several previous research efforts that attempted to increase the density of registering resources within FPGA architectures and a discussion of the potential drawbacks of these systems. This will lead to an analysis of the underlying components of existing FPGAs and a discussion of the potential benefits of adding registers to both the interconnect network and the logic blocks.

8.1: Scaling of CLB Requirements and Performance

As seen in Figure 8.1 and Figure 8.2, the critical path delay of a netlist roughly scales linearly with the amount of pipelining or C-slowing performed on the circuit. The vertical axis of these two figures show the post-routing critical path delay when a circuit was placed and routed onto the four 4-LUT, four flip-flop, length-4 wire architecture used in Chapter 6. The horizontal axis indicates the logical depth of the netlist. Each of the 11 combinational and 11 sequential MCNC netlists were pipelined/C-slowed and then Leiserson/Saxe retimed such that the maximum logical depth of the circuit ranged from the original logic depth of the MCNC netlist to at least three registers following each LUT. Thus, the rightmost point of each line represents the original MCNC netlist, and the leftmost point represents the depth = 0.33 netlist used in Chapter 6. Although the slope of the line differs slightly for each netlist, the impact of additional registering on the achievable critical path delay is relatively clear.

Although this performance gain is encouraging, Figure 8.3 and Figure 8.4 show this does come at a price. Specifically, as more registers are introduced into the various netlists, the number of required CLBs also rises to accommodate the extra registers. As seen in Figure 8.3, the area overhead is relatively low for the majority of originally purely combinational circuits when the logic depth of the circuit is 1 or greater. These netlists generally require less than 1.5x the number of CLBs required by the unpipelined circuit. These benchmarks can be efficiently handled because the target architecture has one optional flip-flop per LUT inside each logic block. As seen in Figure 8.5a and Figure 8.5b, the registers in these moderately pipelined combinational circuits can largely piggyback on the flip-flops that are on the output of each LUT. However, as seen in Figure 8.5c, this area overhead can become very large when the logic depth of the

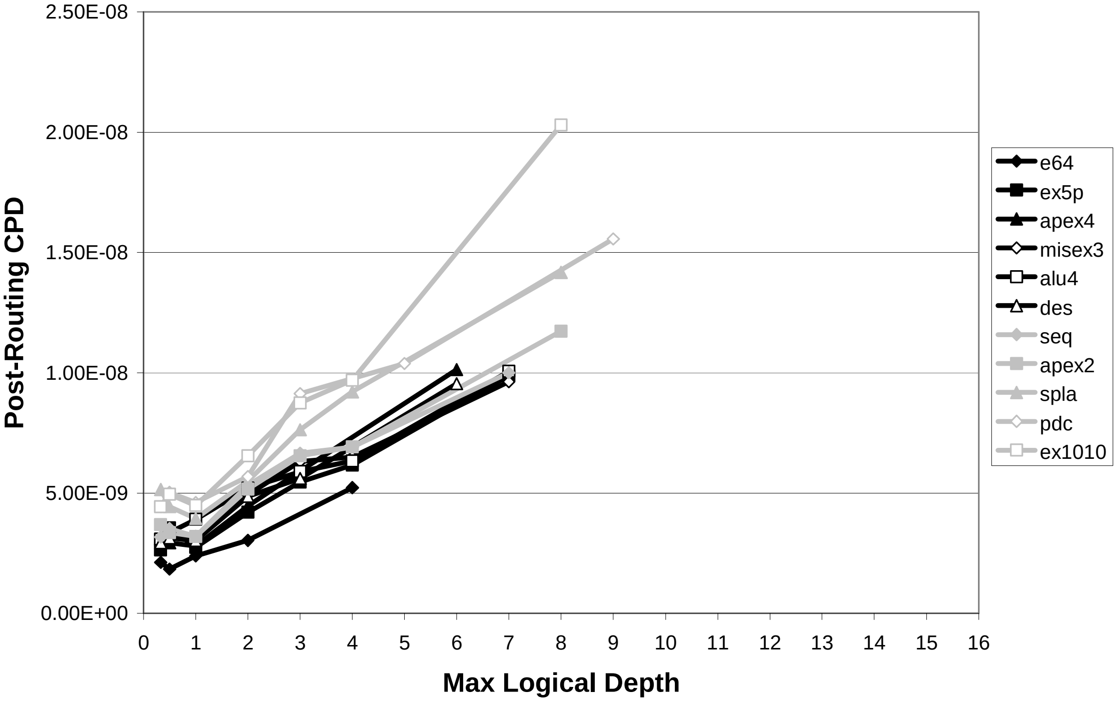


Figure 8.1: Combinational MCNC Netlists Critical Path Delay

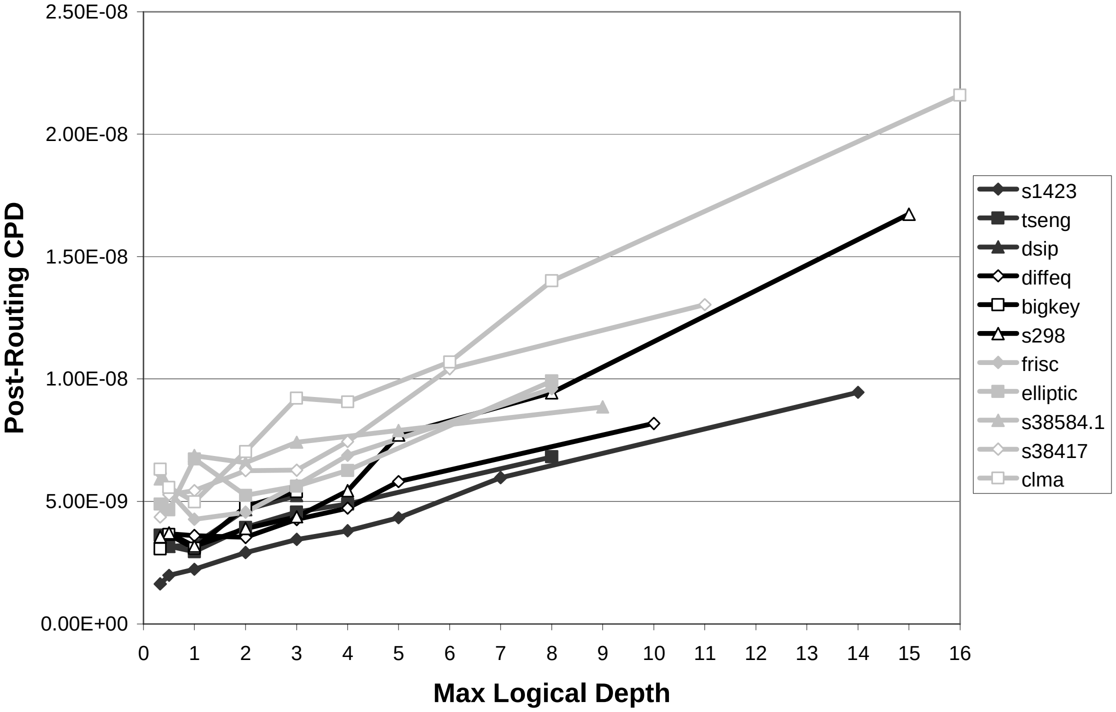


Figure 8.2: Sequential MCNC Netlists Critical Path Delay

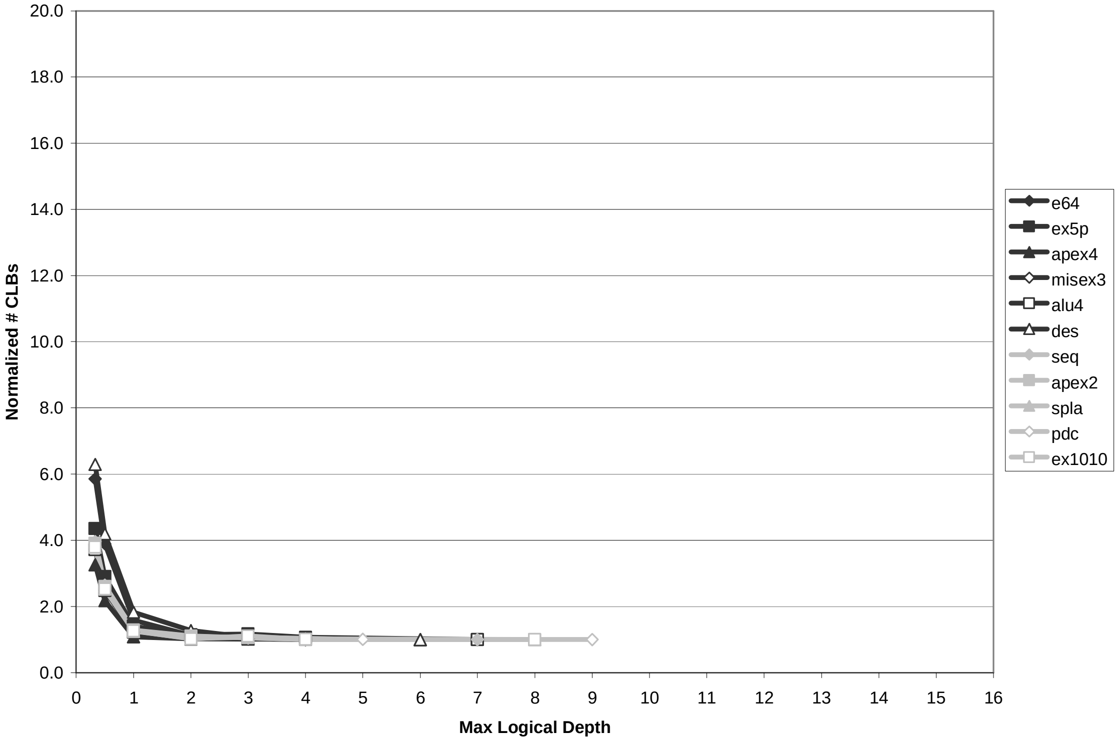


Figure 8.3: Combinational MCNC Netlists CLB Requirements

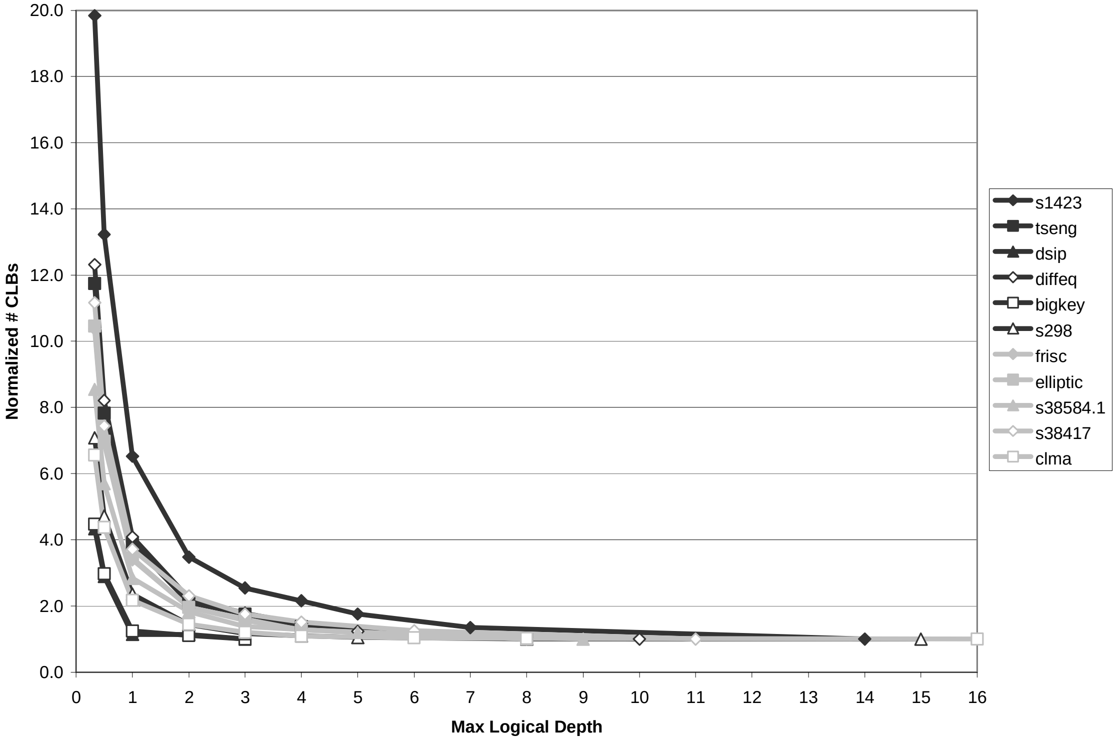


Figure 8.4: Sequential MCNC Netlists CLB Requirements

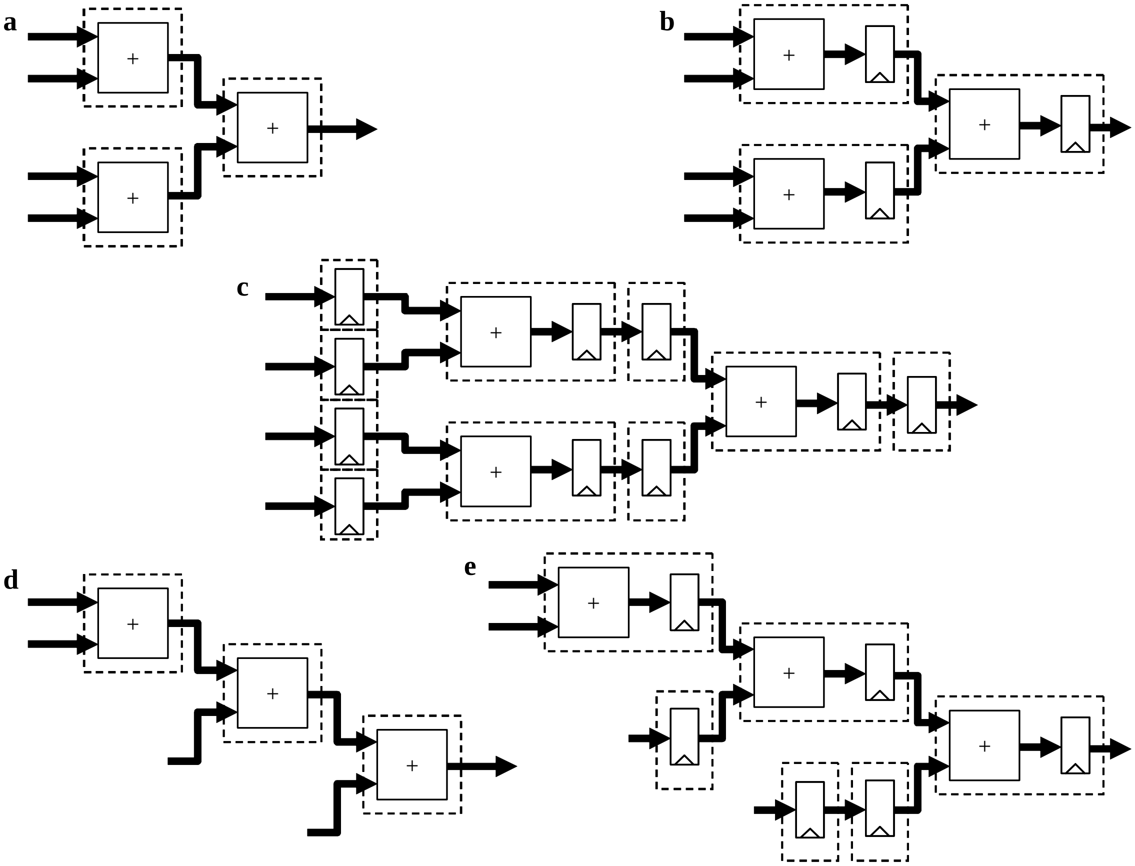


Figure 8.5: Effect of Pipelining and Netlist Topology on CLB Requirement

circuit dips below 1. This is because although each new register added beyond one per LUT allows the system to better pipeline potential interconnect delay, it also requires an additional BLE. Thus, a depth = 0.33 netlist will require approximately 3x the number of CLBs as a depth = 1 implementation.

However, pipelining or C-slowing more sophisticated circuits to a depth of even one LUT can potentially require a large number of additional CLBs. As seen in Figure 8.4, the area overhead is much higher when adding registers to the sequential MCNC netlists. For these circuits, pipelining or C-slowing to a depth of one LUT generally requires 2-4x the number of CLBs as the original circuit. This is because, to pipeline or C-slow a circuit without changing the functionally, all paths through the system must add the same number of registers. However, as seen in Figure 8.5d and Figure 8.5e, if the original circuit has a non-uniform logical depth, some connections will accumulate more registers than others to match the latency of the longest path. These additional registers require new BLEs.

The CLB overhead associated with pipelining or C-slowing these netlists matches the behavior reported in previous research. The radio cross-correlator in [41] indicated that the best circuit found by hand-pipelining and hand-placing registers required 4x the number of CLBs as the unpipelined circuit. This means that at least ¾ of the CLBs in the system were being used only for their flip-flops and not for their logical resources. Thus, while heavily registering a circuit can considerably boost performance, this can require a significantly larger FPGA. Furthermore, a large portion of the LUTs in the underlying fabric may sit idle as these netlists require a much larger ratio of registers to logic than commercial architectures typically provide.

8.2: Previous Register-Rich FPGAs

Multiple research groups have noticed that conventional FPGA architectures can have trouble implementing heavily registered applications. Thus, several research efforts have attempted to address these concerns by increasing the number of registering resources inside the logic blocks and embedding pipelining resources within the interconnect network itself. Unfortunately, all of the systems suggested so far restrict the types of circuits mapped to these devices or have significant overheads inappropriate for many applications.

Although they differ in several key ways, HSRA [40] and SFRA [42] both provide vast pipelining resources in each logic block. Each input of the LUTs in these architectures has a large bank of optional flip-flops. In addition, some fraction of the programmable switchpoints inside their routing switchboxes have optional registers. Although these resources allow for very fast, fixed-frequency operation, these devices provide so many registers that they also suffer a 2-4x area penalty compared with conventional FPGAs. This kind of overhead is unacceptable for applications that cannot make use of these resources.

Furthermore, these architectures also require extremely high levels of C-slowing or pipelining. The authors of [40] and [42] needed to pipelined or C-slowed their applications somewhere between five to 67 times in order for them to be suitable amenable to these architectures. However, as mentioned in Chapter 3, even applications that could potentially be sped up by some pipelining or C-slowing typically cannot be that deeply registered due to their input and output protocols. Thus, while an architecture such as HSRA or SFRA can be useful for some very specific applications, the area overhead and registering requirements are likely far too large for a mass-market FPGA.

Alternatively, some systems have been developed that, while not insisting that their mappings be heavily registered, provide support for such computations by adding registers to the interconnect. Again, although they differ in several key ways, RaPiD [9] and CHESS [27] both offer optional registers in their switchboxes. Even though computations do not need to be pipelined to be efficiently implemented, the opportunity exists if application developers desire. Unfortunately, both of these systems are also optimized to very specific types of computation. The underlying logic and interconnect resources that they provide can make implementing more generic computation very difficult.

All of these architectures significantly compromise their general-purpose use. This largely goes against one of the guiding philosophy of FPGAs themselves: provide a versatile and cheap alternative to ASICs. The limited widespread appeal of existing register-centric systems hampers their ability to leverage economies of scale and Moore's Law, ultimately restricting their quality and availability.

8.3: New Potentials for Increasing Register Capabilities

Looking into the future, one question is how to improve the performance of FPGAs for heavily registered applications while not seriously affecting the area and performance characteristics of the device for more classical applications. No matter what the advantages are for specialized deeply pipelined and C-slowed netlists, it is difficult to justify changes that can significantly degrade the area or timing profile of an architecture for netlists that cannot use these resources. Thus, rather than drastically changing the well-established characteristics of current FPGAs by completely revamping their organization, it may be better to make minimally invasive architectural changes that, while offering significant benefit to suitable register-rich circuits, will disrupt the general-purpose use of the device as little as possible.

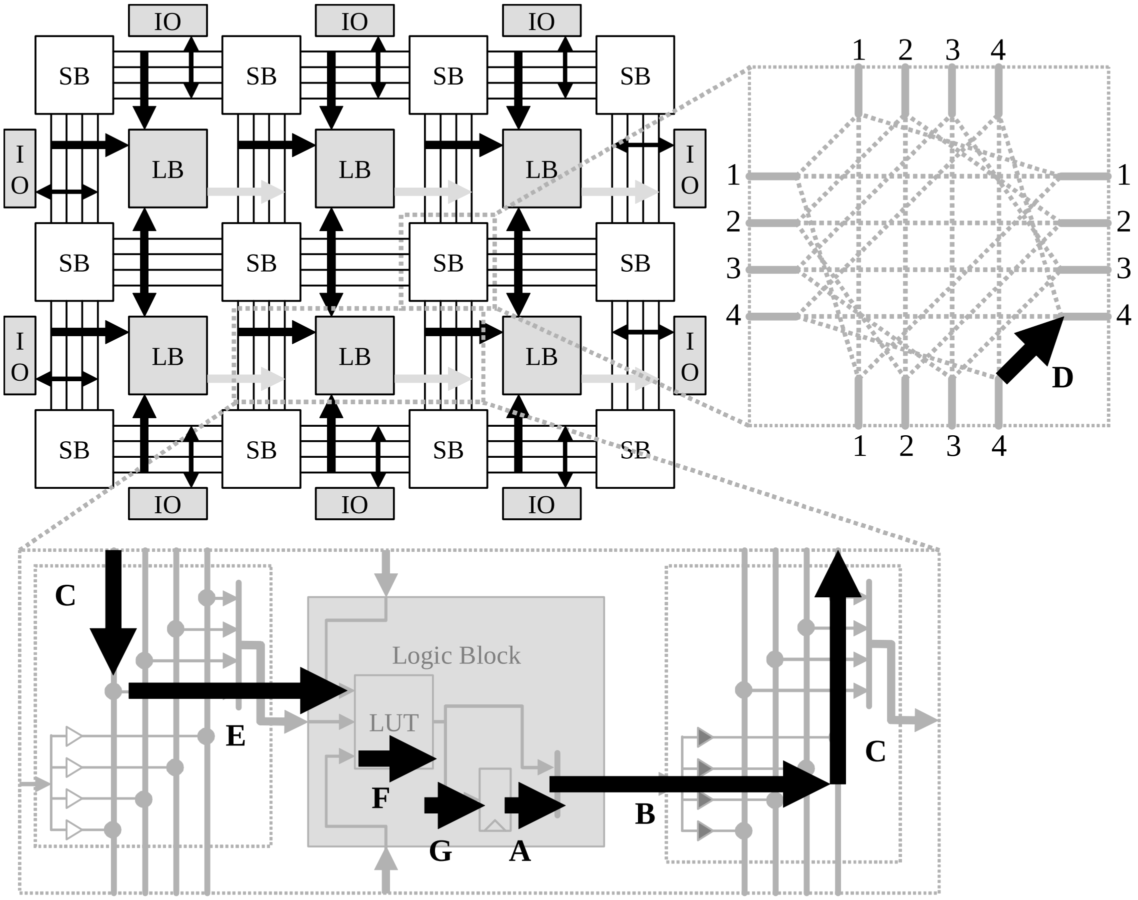
The following sections will investigate the potential advantages and disadvantages of introducing additional register resources into modern island-style FPGAs. This will begin with an analysis of architectures with registers in the interconnect network and will continue with a discussion of the feasibility of adding registers into the logic blocks.

8.3.1: Potential of Registered Switchboxes

Research efforts such as the registered-track graph FPGA in [38] and RaPiD [9] have suggested embedding registers with limited connectivity within interconnect switchboxes. These registers are attractive because they can be introduced with relatively little additional area and can pipeline long wires without adding the delay associated with entering and exiting a CLB. However, as described in Chapter 7, the CAD tools necessary to efficiently map flip-flops to these registers may not be entirely straightforward. More importantly, as will be shown in this section, the potential critical path delay advantages for heavily registered applications on these types of architectures may be relatively small.

Ignoring signals that are associated with the I/O pins of a device, the critical path in any FPGA design will either begin at a flip-flop and end at another flip-flop, or begin at a flip-flop, pass though one or more LUTs and end at another flip-flop. Since this dissertation is primarily concerned with heavily registered circuits, for simplicity, signals that perform computation will assume to be pipelined or C-slowed such that they only pass though a single LUT.

In a conventional FPGA, flip-flops are only available inside logic blocks. This means that the critical path of a circuit will begin at one CLB, go through some number of interconnect wires and switchboxes and end at another CLB. The delay required by such a signal can be broken up into multiple pieces. Although the precise area and performance numbers of commercial architectures are not publicly available, the architecture files provided by the VPR [3] toolflow and the toolflow itself can be mined for some reasonable information regarding a modern 0.65nm FPGA with four 4-LUT BLEs per CLB and length-4 wires. As seen in Figure 8.6, there are seven numbers that are particularly significant: the clock to output delay of a flip-flop, the delay required to exit a CLB and enter a wiring channel, the delay through a single wire segment, the switching delay between wire segments, the delay required to enter a CLB through the



A	FF clock to Q delay	1.261E-10
B	CLB demultiplexer delay	6.562E-11
C	1 wire segment delay	1.390E-9
D	Switchbox delay	6.562E-11
E	CLB input multiplexer delay	2.478E-10
F	LUT propagation delay	1.679E-10
G	FF setup time	1.280E-10

Figure 8.6: Significant Delay Numbers for an Island-Style FPGA
Information taken from 65nm four 4-LUT, length-4 wire FPGA architecture

input multiplexers, the propagation delay of a LUT and the setup time for a flip-flop. These delay numbers can be used to perform some rough calculations and estimate the anticipated critical path delay of a mapped circuit.

The delay for a signal on a conventional architecture that goes between one flip-flop and another without passing though a LUT is shown in Equation 8.1. Here, the signal will exit a flip-flop, exit the CLB, traverse *N* wire segments and (*N* - 1) switchboxes, enter a CLB and finish at another flip-flop. Similarly, the delay of a signal on a conventional architecture that goes from one flip-flop to another through one LUT is shown in Equation 8.2. In this case, the signal will exit a flip-flop, exit the CLB, traverse *N* wire segments and (*N* - 1) switchboxes, enter a CLB, pass through a LUT and finish at another flip-flop.

$$CPD = A + B + (N * C) + [(N-1) * D] + E + G \tag{8.1}$$

$$CPD = A + B + (N * C) + [(N-1) * D] + E + F + G \tag{8.2}$$

Since the only difference between these two equations is the *F* term, when traveling through an equal number of wires, a signal that uses a LUT will be slightly slower. Of course, for the application to actually perform computation, some signal in the circuit must use a LUT. Thus, Equation 8.2 will likely be the critical path.

As shown in Figure 8.7, an architecture with interconnect registers has flip-flops in both the logic blocks and the switchboxes. The hope is that these additional registers can make the system faster by removing the time to exit/enter a CLB (delay *B* and *E* in Figure 8.6) and reduce the number of wires between registers (the *N* terms in the previous equations). There are eight possible scenarios for the critical path on these kinds of devices.

The first two possibilities are identical to the situation in an architecture without switchbox registers, that a signal begins at a flip-flop inside a CLB, either goes through or does not go though a LUT and ends at another flip-flop inside a CLB. Thus, the delay on these signals will be the same as described in Equations 8.1 and 8.2.

The second two possibilities are that a signal begins at a flip-flop inside a switchbox, either goes through or does not go though a LUT, and ends at a flip-flop inside a CLB. Assuming that the delay associated with the output demultiplexer on a register embedded inside a switchbox is the same as the delay of the switch between two wire segments, the delay of a signal that begins at a switchbox register and ends at a CLB

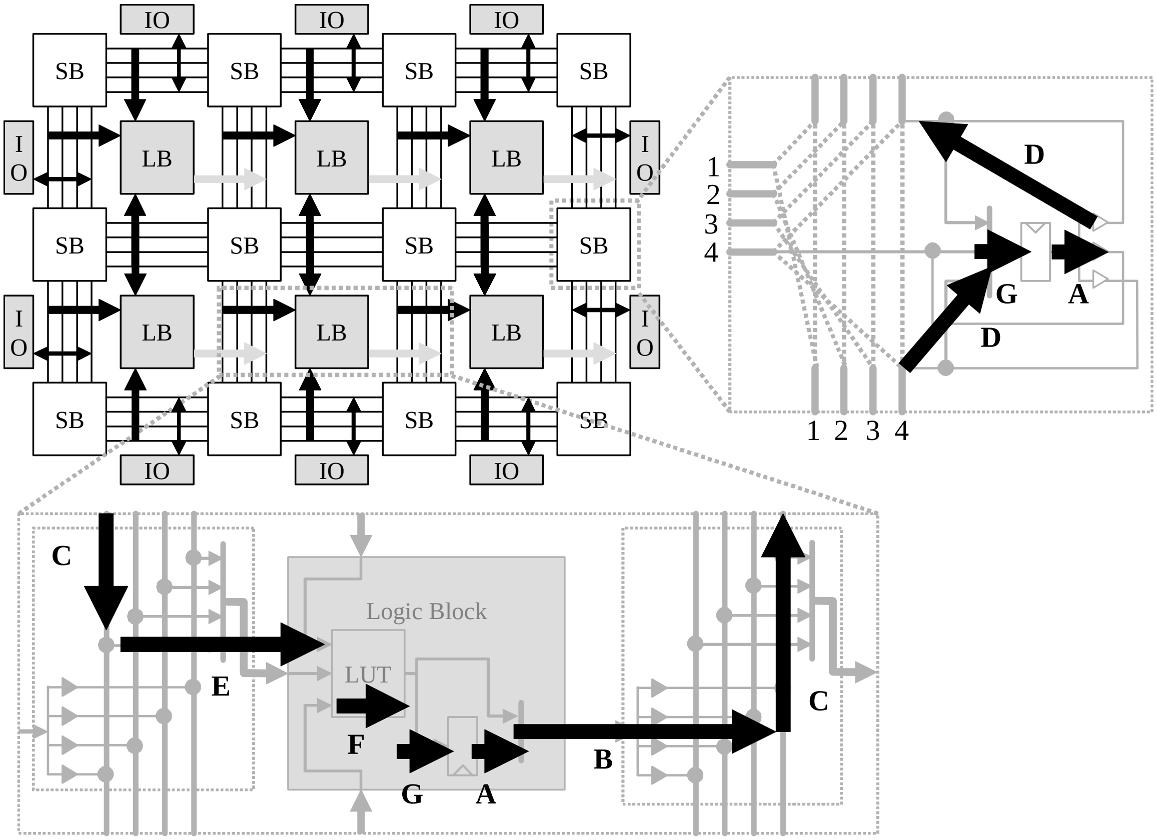


Figure 8.7: Significant Delay Numbers for an Island-Style FPGA with Registered Switchboxes

register without passing though as LUT is shown in Equation 8.3. The delay of a similar signal that goes through a LUT is shown in Equation 8.4.

$$CPD = A + D + (N * C) + [(N - 1) * D] + E + G \tag{8.3}$$

$$CPD = A + D + (N * C) + [(N - 1) * D] + E + F + G \tag{8.4}$$

The third two possibilities on an architecture with registered switchboxes are that a signal begins at a flip-flop inside a CLB, either goes through or does not go though a LUT, and ends at a flip-flop inside a switchbox. Assuming that the delay associated with the input multiplexer on a register embedded inside a switchbox is the same as the delay of the switch between two wire segments, the delay of a signal that begins at a CLB and ends at a switchbox register without passing though as LUT is shown in Equation 8.5. The delay of a similar signal that goes though a LUT is shown in Equation 8.6. Notice that Equation 8.6 is optimistic in that assumes that a flip-flop in one BLE can directly feed a LUT in another BLE within the same logic block. Since most LUTs will require multiple inputs, it may not be possible to register all of the incoming signals within the same CLB as the actual computation.

$$CPD = A + B + (N * C) + [(N - 1) * D] + D + G \tag{8.5}$$

$$CPD = A + F + B + (N * C) + [(N - 1) * D] + D + G \tag{8.6}$$

The last two possibilities for this device are that a signal both begins and ends at a flip-flop inside a switchbox, either going through or not going though a LUT. Using the same conventions as before, the delay of a signal that does not go through a LUT is shown in Equation 8.7. The delay of a similar signal that goes though LUT is shown in Equation 8.8.

$$CPD = A + D + (N * C) + [(N - 1) * D] + D + G \tag{8.7}$$

$$CPD = A + D + (N * C) + [(N - 1) * D] + E + F + B + D + G \tag{8.8}$$

Looking at these eight possible situations, many of them can be eliminated from consideration. For example, while a signal that begins and ends at registers inside a CLB could indeed be the critical path of a circuit mapped to an architecture that has interconnect registers, this does not use the primary feature of the system. Erring on the optimistic side, the hope is that the tools will be able to use the interconnect registers available and these situations will not be the critical path of a mapped application. By the same token, the delay of a signal shown in Equation 8.8 that begins at a switchbox register, passes though a LUT and ends at another switchbox register is slower than a similar length signal on an architecture that does not have switchbox registers. Largely, this is because such a path still enters and exits a CLB, but also must contend with the input and output multiplexing on switchbox registers. This leaves five possible scenarios, three paths that do not pass though a LUT (Equations 8.3, 8.5, and 8.7) and two paths that do (Equations 8.4 and 8.6). Between these different possibilities, Equations 8.4 and 8.6 have the largest likelihood of being on the critical path since they perform computation.

These three equations (Equations 8.2, 8.4, and 8.6) can be used to compare the potential critical path delay of netlists mapped to both FPGAs that only have registers inside CLBs and architectures that have registers inside both CLBs and switchboxes. Table 8.1 applies the delays shown in Figure 8.6 to Equation 8.2 for values of *N* between 1 and 4 wire segments. This shows the critical path delay and resulting maximum clock frequency for an application mapped to a device with only CLB registers. Similarly, Table 8.2 shows the results of Equations 8.4 and 8.6 on an FPGA with interconnect registers when the critical path either goes from a register inside a switchbox to a register inside a CLB (left side) or from a register inside a CLB to a register inside a switchbox (right side).

Table 8.1: Estimated Critical Path Delay of Conventional FPGA

# Wire Segments on Critical Path	Critical Path Delay	MHz
1	2.128E-9	470.02
2	3.585E-9	278.91
3	5.043E-9	198.29
4	6.501E-9	153.82

Table 8.2: Estimated Critical Path Delay of Island-Style FPGA with Registered Switchboxes

	Switchbox Reg to CLB Reg			CLB Reg to Switchbox		
# Wire Segments on Critical Path	**Critical Path Delay**	**MHz**	**Norm. Speed**	**Critical Path Delay**	**MHz**	**Norm. Speed**
1	2.128E-9	470.02	1.000	1.945E-9	514.02	0.914
2	3.585E-9	278.91	1.000	3.403E-9	293.84	0.949
3	5.043E-9	198.29	1.000	4.861E-9	205.72	0.964
4	6.501E-9	153.82	1.000	6.319E-9	158.26	0.972

Comparing these results, when the critical path goes from a register inside a switchbox to a register inside a CLB, for any given value of N wire segments, an application mapped to an architecture with interconnect registers is no faster than on an architecture without interconnect registers. This is because, comparing Equations 8.2 and 8.4, the only difference between these two arrangements is that B is traded for D. Stated another way, the delay through a CLB demultiplexer is replaced by the delay through a switchbox register demultiplexer. However, since D equals B in the VPR model, the architecture with interconnect registers is no faster. Even if delay though a switchbox register demultiplexer were reduced to zero ($D \rightarrow 0$), this would only remove 6.562E-11 seconds of delay from the critical path. In the best case, where there is one wire segment on the critical path, this would make an architecture with interconnect registers only ([2.128E-9 - 6.562E-11] / 2.128E-9 = 0.969x) faster.

Furthermore, when the critical path goes from a register inside a CLB to a register inside a switchbox, for any given value of N wire segments, an application mapped to an architecture with interconnect registers is only marginally faster than on an architecture without interconnect registers. Comparing Equations 8.2 and 8.6, the only difference is that E is traded for D, or that the delay through a CLB input multiplexer is replaced by the delay though a switchbox register input multiplexer. However, this only represents a saving of (2.478E-10 - 6.562E-11 = 1.8218E-10) seconds. As shown on the right side of Table 8.2, at best this results in an architecture with interconnect registers being 0.914x faster. Unfortunately, this may not be a large enough performance benefit to justify modifying the architecture and opening the door for problems with the CAD tools. For perspective, according to Xilinx's datasheets [45], the performance difference between only one device speed grade is approximately 0.91x. On top of this, the advantage also quickly decreases as the number of wire segments along the critical path is increased. At two wire segments, the registered switchbox architecture is only 0.949x faster. This is particularly concerning since two wire segments are often required even in the most heavily registered circuits to allow the system to turn a corner and connect logic blocks or switchboxes in different rows or columns.

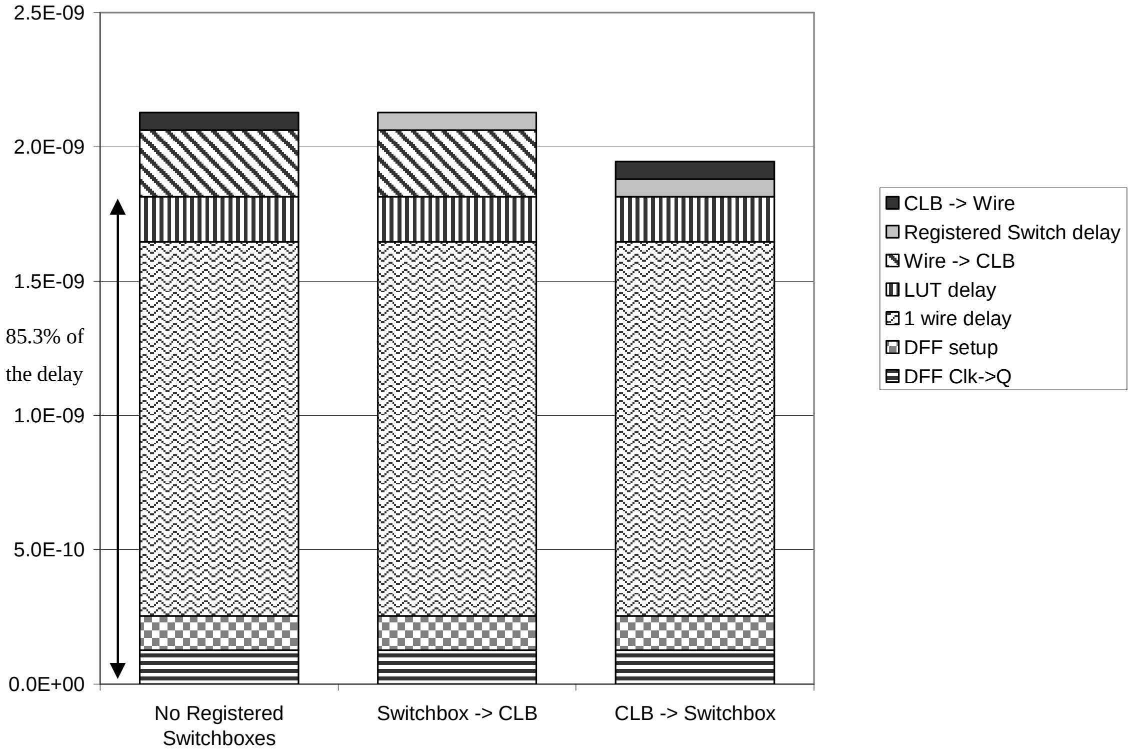


Figure 8.8: Delay Contribution of Best-Case Scenarios (1 Wire Segment) for Registered Switchboxes

Looking at Figure 8.8, it is clear why architectures with registered switchboxes have such a small performance advantage over more conventional devices. Regardless of the architecture, there is very little that can be done about four components of the critical path delay: the clock to Q delay of a flip-flop, the setup time of a flip-flop, the delay through a single wire segment and the delay of a LUT. These portions alone comprise over 85% of the critical path delay, even in the best case of a single wire segment between the source and sink registers. Thus, even if the overhead associated with getting in or out of a register inside a switchbox were reduced to zero, such an architecture is limited to an approximately 15% performance improvement. However, it should be noted that the largest portion of this “unavoidable” delay is caused by the delay through the wire segments themselves. If an FPGA were to use very high strength drivers or some other technique to drastically reduce the delay of the wires, the potential advantage of these kinds of architectures might go up. That said, it is expected that the delay through wires will only become a larger portion of overall delay in future process generations.

Of course, though, this analysis makes one critical assumption: that the number of wire segments along the critical path of a netlist mapped to the two architectures is the same. In practice, this may not be the case since registered switchboxes increase the ratio of registers to logic in the architecture. A heavily registered netlist mapped to a register-enhanced architecture will likely be more densely arranged than when mapped to a conventional device. This is because the implementation mapped to a classical FPGA will probably need to spread out over more CLBs for all of the signals to accumulate the necessary flip-flops. At the very least, this could mean that the average wirelength of the nets in the circuit will be longer, if not the wirelength along the critical path.

To get any notable speedup, architectures that have registered switchboxes must rely on the fact that the additional registers in the system can generally reduce the number of wire segments along the critical path. Without this feature, these architectures are not really intrinsically faster. However, as mentioned earlier, these architectures also contain registers with very limited input and output connectivity, changing the fundamental problem presented to placement and routing tools. As will be explored in the next section, it may be better to find a way to adding inexpensive but highly connected registers to the system. These can be incorporated into the CLBs. This would allow an architecture to obtain short wires without creating a problem for the CAD tools.

8.3.2: Enhancing Logic Blocks with Additional Registers

Although incorporating registers into the interconnect network of an island-style FPGA may not provide a large performance benefit, additional registers need to be placed somewhere in the architecture to improve the support for heavily registered applications and keep the number of wires along the critical path relatively low. Although increasing the register capacity of the logic blocks is the obvious alternative, this must be done relatively carefully to avoid seriously affecting the area or performance of netlists that are lightly registered. This problem is made even more difficult since it is highly preferable that any additional registers have the same high connectivity as existing registers to prevent issues with CAD tools.

From a practical standpoint, there are two different issues regarding how heavily registered netlists map to conventional architectures. Although these problems are somewhat intertwined, the nature of these issues can be largely separated and addressed independently. The first issue is that a large portion of the silicon resources in a conventional FPGA architecture cannot be used when mapping a heavily registered application. While heavily registered circuits require a large number of additional BLEs, the LUTs in the majority of these blocks are entirely ignored and only the flip-flops are used. These unused LUTs actually contain a large amount of registering resources that could be made available with some relatively minor architectural modifications. The second issue is that the register density of conventional architectures may not be high enough to efficiently map heavily registered circuits. Pipelining or C-slowing a circuit can cause a netlist to spread out so that the necessary registers can be accumulated. However, this can also cause the circuit to slow down because the average wirelength of each net may go up. Thus, for the system to increase the operational frequency of an application beyond a certain point, it is likely necessary to increase the number of available registers in computationally dense regions. While enhancements to improve one of these issues can improve the other as a side effect, the nature of the architectural changes that will be suggested are distinct.

8.3.2.1: Using LUTs as Shift Registers

Not being able to use the majority of the LUTs in an FPGA is wasteful in two ways. First, CLBs devote a significant amount of area to multiplexing the inputs and outputs of their LUTs. Second, as discussed in Chapter 2, LUTs are actually built from small memories. Putting these two characteristics together, each BLE actually has the basic building blocks to potentially register multiple signals if the LUT is not needed for a logic function. However, conventional architectures only provide the capability to register one signal using the flip-flop.

Some commercial FPGAs already unlock a portion of this potential. As shown on the left of Figure 8.9, a conventional 4-LUT consists of 16 individual memory cells. The content of these cells is programmed when a circuit is downloaded to the device. During normal operation, the values held in these cells are selected through a multiplexer to implement a logic function. However, modern Xilinx devices expose the underlying memory cells within half of their LUTs to allow them to be used either as a conventional LUT or as a 1 to 16-bit shift register. Although the exact mechanism Xilinx uses to provide this functionality is not publicly known, the illustration on the right of Figure 8.9 shows one possibility. This shift register capability can be provided by adding a small number of additional components to each BLE: twenty 2:1 multiplexers and four memory cells. 16 of the 2:1 multiplexers are added to the input of each of the original memory cells to control whether the value written into the cell comes from the programming logic or, when forming a shift register, from the previous memory cell. The remaining four multiplexers are added to the address lines of the 16:1 multiplexer to control whether the address comes from the outside world, to implement a logic function, or from a static address that is defined when the BLE is programmed as a shift register.

This modification adds a huge raw number of registers into the architecture because every previously unused LUT in the device can implement up to sixteen registers. For example, if a purely combinational circuit requires N LUTs, this can be mapped to N BLEs. A deeply pipelined version of this circuit may add $4N$ registers. On an architecture that offers one LUT and one flip-flop in each BLE, this will require $(N + 3N = 4N)$ BLEs. The first N registers can be packed into a BLE with a LUT, but the other $3N$ registers must be assigned to their own BLEs. The same circuit on an architecture that offers one flip-flop and one LUT that can be turned into a 1 to 16-bit shift register in each BLE could theoretically only require $(N + 3/17\ N \approx 1.18\ N)$ BLEs. This is because every BLE beyond the original N does not use the LUT for logic. This makes it available for use as a 16-bit shift register. Combined with the flip-flop, this allows each BLE with an unused LUT to implement 17 registers.

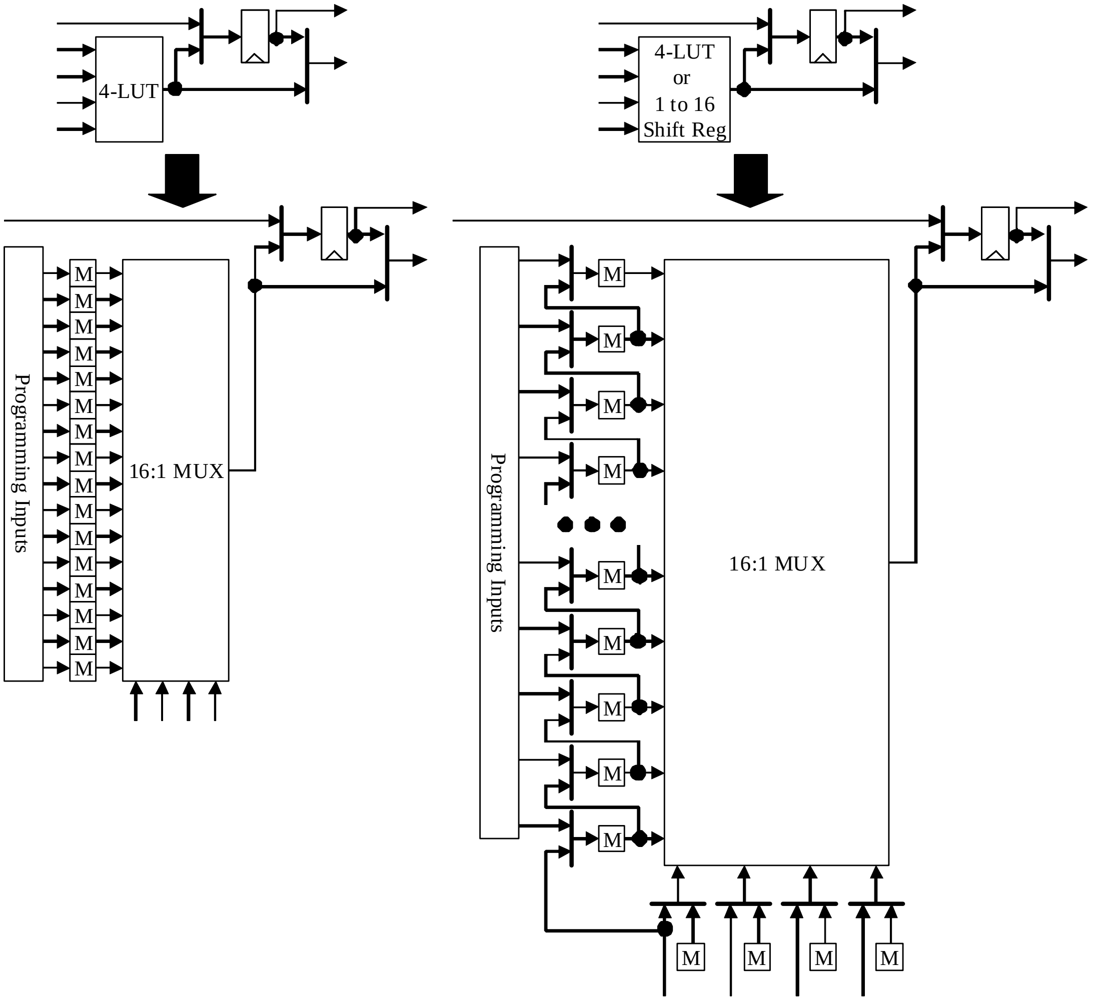


Figure 8.9: Conventional BLE (left) and LUT/16-bit Shift-Register BLE (right)
"M" denotes a memory cell

However, the number of truly useful registers is likely considerable lower. This is because although heavily registered netlists require a large number of pipelining resources, the distribution of these demands is relatively even throughout the circuit. For example, in the heavily pipelined and C-slowed benchmarks in [40], 99% of signals require eight or fewer registers while 95% require four or fewer. Thus, the majority of nets simply cannot use deep, monolithic register banks. For that matter, even if a net requires a large number of registers between the source and sink, it is unlikely that it is a good idea to group all of the registers in a single location from a performance standpoint. This is because one of the primary advantages of adding registers into a netlist is the capability to break very long paths into smaller parts. Thus, shift registers will only be used for one or two registers rather than the full 16. If the system is able to map an average of 1.5 registers to each shift register, the number of required BLEs would be ($N + 3/2.5\,N = 2.2\,N$).

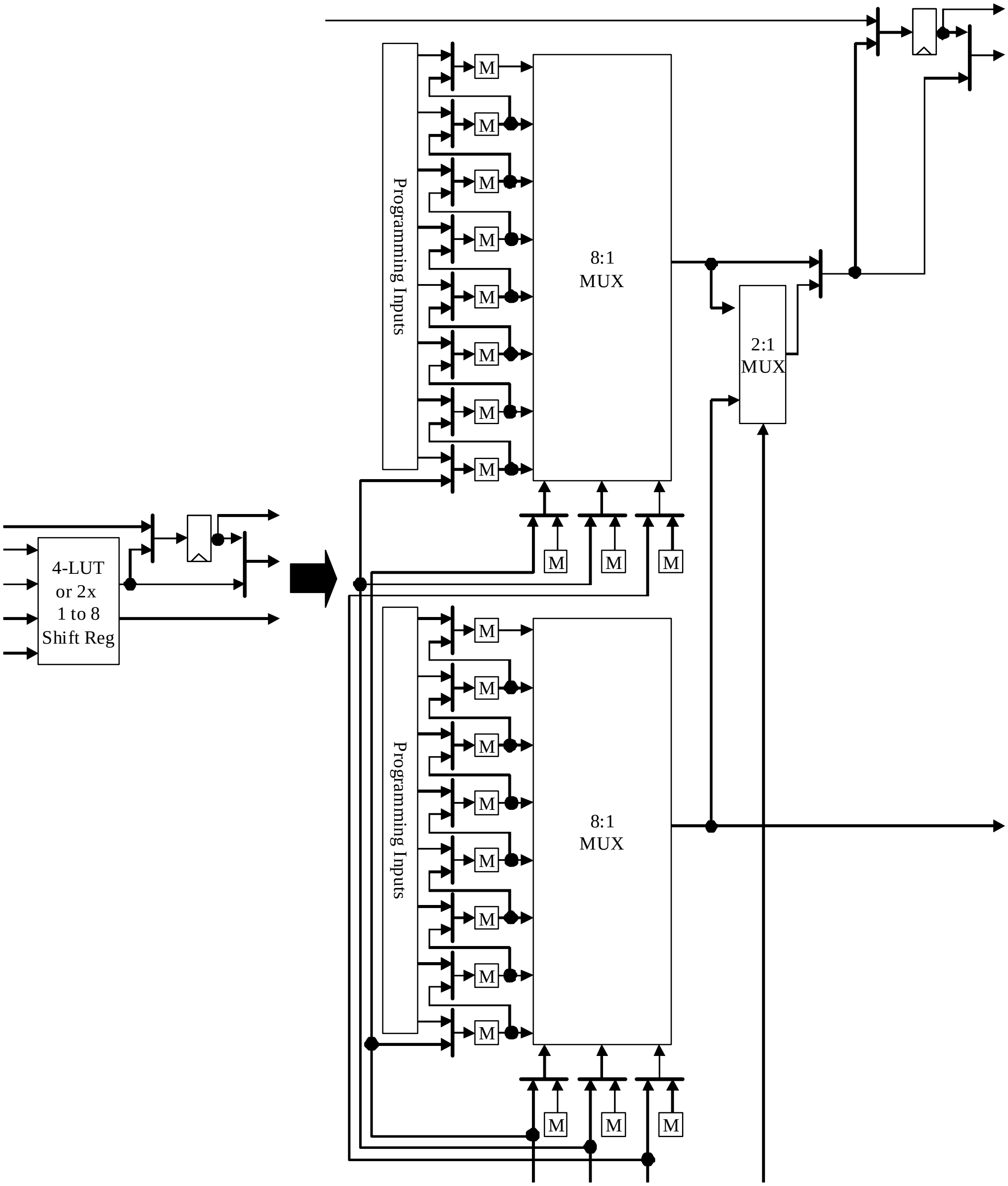


Figure 8.10: LUT/Two 8-bit Shift-Register BLE
"M" denotes a memory cell

The large number of low latency signals in typical netlists indicates that it is likely more useful to add the capability to register multiple different signals by a smaller amount rather than a single signal by a large amount. As shown in Figure 8.10 and Figure 8.11, a single large shift register can be split into two or four

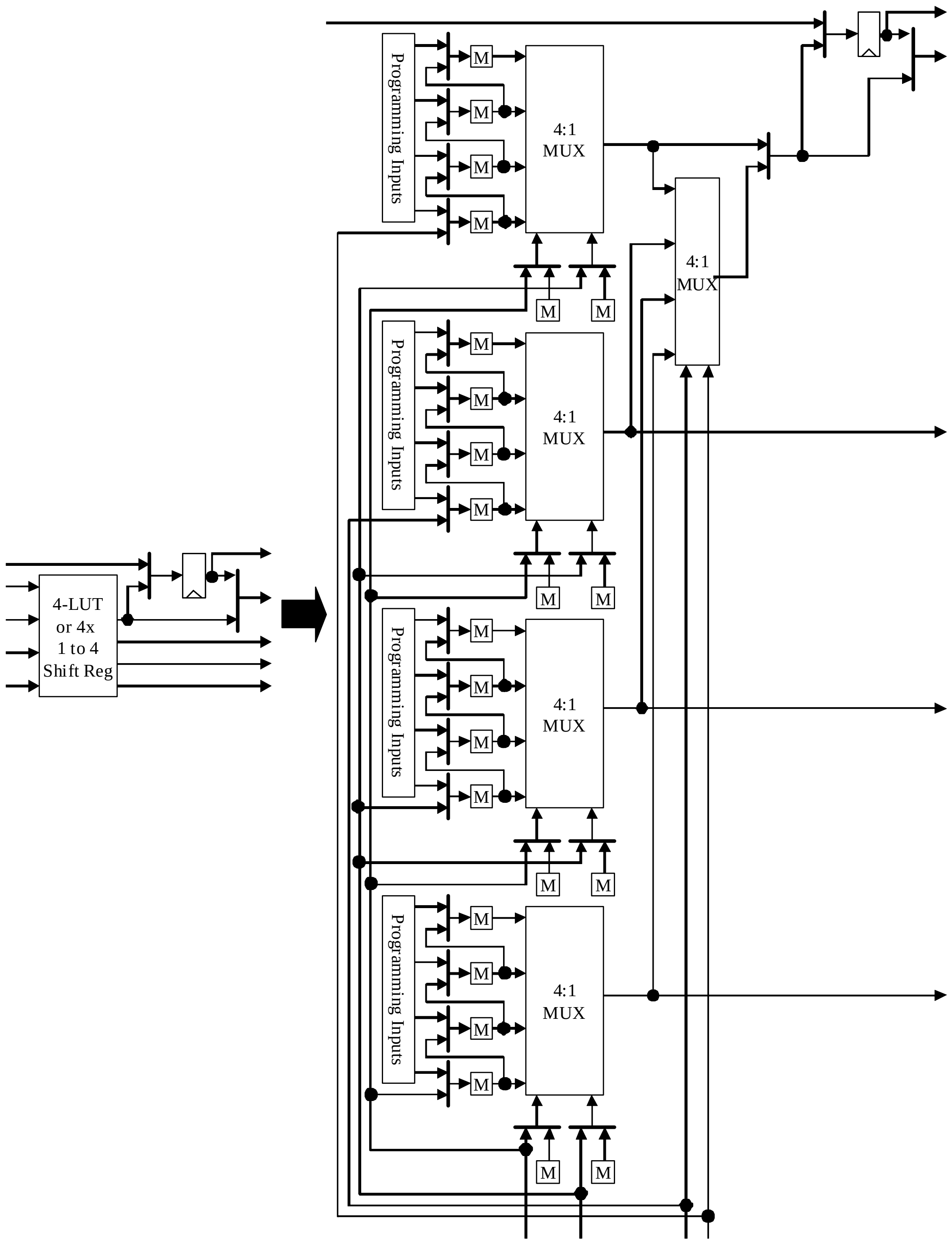


Figure 8.11: LUT/Four 4-bit Shift-Register BLE

"M" denotes a memory cell

smaller shift registers. Assuming that the 16:1 multiplexer in Figure 8.9 can be broken into smaller multiplexers with little to no overhead, splitting a 16-bit shift register into two 8-bit shift registers will likely require two additional 2:1 multiplexers and additional two memory cells. Similarly, splitting a 16-bit shift register into four 4-bit shift registers will require four additional 2:1 multiplexers and four additional memory cells.

That said, there is another cost associated with splitting a LUT into smaller shift registers. Adding the capability of turning a LUT into a single 1 to 16-bit shift register likely incurs relatively little overhead. This is because although additional multiplexers and memory cells are needed, the input and output connectivity of the larger CLB does not require any changes – the input and output of the shift register simply borrow the connections already needed to use the LUT for logic. However, splitting a shift register into two or four smaller shift registers requires one or three additional outputs, respectively. Providing full connectivity for these new outputs to the external channel wires could significantly affect the area of the CLB and associated connection blocks.

8.3.2.2: Adding Independent Flip-Flops

The fact that registers are generally evenly distributed throughout circuits also contributes to the second issue that this section would like to address – that the achievable clock frequency of a circuit may be limited by the number of registers within specific areas of the device. This phenomenon can be most easily seen in Figure 8.1 and Figure 8.2 for circuits with a logical depth below one LUT. While the critical path delay of most of the circuits goes down as more registers are added, the critical path delay for some of the circuits stays constant or even goes up. This likely occurs because these very heavily registered circuits require a higher density of registers than the architecture provides in order to improve the critical path delay.

Taking a step back for a moment, consider a different scenario. When a purely combinational circuit is mapped to an FPGA, each of the individual logic blocks must fight with the others to be as close as possible to the other logic blocks to which they are connected. Assuming that routing congestion is not an issue, the operational frequency of the resulting implementation is largely determined by how close these logic blocks are able to get. Thus, it is expected that a larger combinational circuit will have a higher critical path delay, even if the logical depth is the same as the smaller circuit. This is because more logic blocks will interfere with each other in the larger circuit and prevent them from getting as close to the logic blocks to which they are connected. However, this can be mitigated by increasing the logical density of the architecture. For example, if this netlist is mapped to an FPGA that has twice as many LUTs in each CLB, each LUT will be able to be “near” twice as many other LUTs. Ignoring for a moment the effect this architectural change might have on the speed of the interconnect wires, this will allow the circuit to run faster.

A similar situation occurs for the registers in a netlist. Beginning with a purely combination circuit, each LUT will try to be as close as possible to the other LUTs to which it is connected. As registers are gradually added to the circuit, they are able to make the system faster because the logical depth of the circuit is reduced. Furthermore, these registers are relatively easy to find. Assuming that the netlist has a uniform logic depth across all paths, until the maximum logic depth of the circuit dips below one, these registers can simply be placed in the same BLE as their source LUT. Thus, the placement of the LUTs in the system is still very dense and the critical path delay is improved dramatically. However, once the logic depth of the circuit dips below one LUT, additional registers require additional BLEs. Since the registers are relatively evenly distributed throughout the circuit, the LUTs in the system must spread out to make room for the required registers. This can nullify any potential advantage of adding the registers in the first place.

This phenomenon is shown in the example in Figure 8.12. As shown in Figure 8.12a, there are two paths in the circuit that must cross each other. The first inverter takes an input from the left of the device and sends the output to the right. The second inverter takes an input from the right of the device and sends the output to the left. For simplicity, this circuit is initially mapped to an architecture with one LUT and one flip-flop per CLB and unit-length interconnect wires. As shown in Figure 8.12b, pipelining this circuit once improves the critical path delay. Rather than traversing one wire segment, propagate through a LUT, and traversing three more wire segments, the new critical path delay is the time required to traverse three wire segments. However, as seen in Figure 8.12c, further pipelining does not improve the critical path delay because the entire netlist must be spread out to make room for these additional registers. The only way to reduce the critical path delay to one wire segment is to increase the number of registers in each CLB. As shown in Figure 8.12d, this can be accomplished by mapping the circuit to an architecture with two independently accessible flip-flops per CLB. Figure 8.13 shows the BLE of such an architecture.

Notice that adding additional independent flip-flops to each BLE affects the system differently than allowing LUTs to be used as shift registers. While both modifications increase the total number of potentially available registers in the architecture as a whole, since shift registers are built from unused LUTs, inserting them into a computationally dense area still requires the placement tool to spread out the LUTs in a netlist to provide whitespace. On the other hand, introducing additional flip-flops into each CLB increases the number of registers in computationally dense regions without any need to change the density of mapped LUTs. Thus, while shift registers can potentially provide denser register resources, additional independent flip-flops are more likely to improve critical path delay.

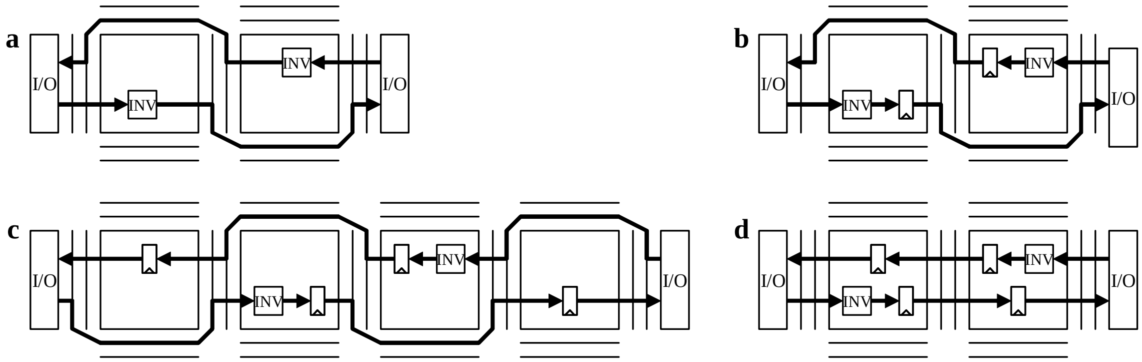


Figure 8.12: Registered Netlists and Effect of Architecture Register Density on Critical Path Delay

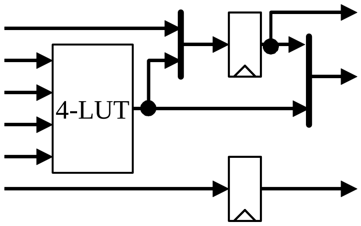


Figure 8.13: BLE with Two Independent Flip-Flops

That said, adding flip-flops to a CLB also requires more inputs and outputs. The shift register architecture suggested in Figure 8.9 added up to 16 registers without requiring any additional inputs or outputs. The split shift registers in Figure 8.10 and Figure 8.11 also added up to 16 registers and require 1 and 3 additional CLB outputs, respectively. However, each additional independently accessible flip-flop added to a CLB requires its own input and output. This means that although independently accessible flip-flops might be more flexible, they are also potentially more expensive.

The difference between independently accessible flip-flops and shift registers can also be seen in how they provide registers to applications. The number of BLEs that a netlist requires can be estimated using Equation 8.9.

$$If\left(L \geq \frac{R}{IndFF}\right), \#\, of\ \ required\ BLEs = L, otherwise = L + \left\lceil \frac{R-(IndFF * L)}{ULReg} \right\rceil \quad (8.9)$$

This equation assumes that each BLE consists of one LUT that may or may not have the capability to implement one or more shift registers (if it is not needed for logic), along with some number of independently accessible flip-flops. *L* is the number of LUTs in the netlist, *R* is the number of registers in the netlist, *IndFF* is the average number of independently accessible flip-flops available in each BLE in the target architecture and *ULReg* is the average number of registers that can be mapped to each BLE with an unused LUT.

Since each BLE contains a single LUT and *IndFF* number of independently accessible flip-flops, if the number of registers in a netlist is relatively low, the number of LUTs defines the minimum number of BLEs needed to implement the circuit. However, if the number of registers in the circuit is large enough, the registers will determine the number of required BLEs. Since each LUT will require its own BLE, the first (*L*IndFF*) flip-flops in the netlist can use the registers in BLEs occupied by a LUT. However, registers beyond this number will require additional BLEs. Each of these extra BLEs with an unoccupied LUT will be able to implement *ULReg* registers. As shown in Equation 8.10, the average number of registers that can be mapped to a BLE with an unoccupied LUT equals the average number of independent flip-flops in each BLE plus the average number of registers can be implemented using the LUT as one or more shift registers. The average number of registers can be implemented using an unoccupied LUT is *ANSR*, the average number of shift registers per BLE, multiplied by *RegPerSR*, the average number of registers that can be mapped to each shift register.

$$ULReg = IndFF + (ANSR * RegPerSR) \tag{8.10}$$

8.4: Evaluation and Results

These equations can be used to gain some basic insight regarding how efficiently heavily registered applications can be mapped to different register-enhanced architectures. As shown in Table 8.3, the initial round of testing investigated how effectively shift register reduce the number of required BLEs compared to independently accessible flip-flops. This testing included fifteen architectures. The first set of five all contain one independently accessible flip-flop per LUT. Architecture *I-0A* is the basic four 4-LUT, four flip-flop architecture described in previous chapters. Since each BLE in this architecture contains one flip-flop, *IndFF* = 1. Since the LUTs in this architecture cannot be used as shift registers, *ANSR* = 0. This makes *ULReg* = 1, since all the BLEs in the systems can provide one register, regardless as to whether or not the LUT is occupied.

Architecture *I-0B* adds the capability for one of the four 4-LUTs in each CLB to be used as a 1 to 16-bit shift register (abbreviated as SR-16). Since each BLE in this architecture still contains one independently accessible flip-flop, *IndFF* = 1. Since one in four of the LUTs in the system can be used as a shift register, *ANSR* = 0.25. At this point, the average number of registers that can be mapped to a shift register becomes important. While each shift register can be used to implement at least one register, the hope is that at least some of the shift registers will be able to be filled with more registers. Although this is largely netlist dependent, 1, 1.5 and 2 seem to be reasonable estimates for *RegPerSR*. These values for *RegPerSR* result in three values for *ULReg*: 1.25, 1.375, and 1.5 respectively. *RegPerSR* = 1 results in *ULReg* = 1.25 because each BLE with an unused LUT has one independently accessible flip-flop and 0.25 shift registers that can be used to implement one register on average. *RegPerSR* = 1.5 results in *ULReg* = 1.375 because each BLE with an unused LUT has one independently accessible flip-flop and 0.25 shift registers that can be used to implement 1.5 registers on average. Architectures *I-0C* through *I-0E* add more hardware to allow two through all four of the 4-LUTs in each CLB to be used as SR-16s. As mentioned earlier, modern Xilinx devices provide one independent flip-flop per BLE and allow half of the LUTs in the system to be used as SR-16s. This is similar to the resources provided by architecture *I-0C*.

Table 8.3: Architectures Used in Testing Phase I – Adding Independent Flip-Flops and 1 to 16-bit Shift Registers

Arch	Description – Contents of Each CLB
I-0A	4x normal LUTs, 4x FFs (default architecture)
I-0B	3x normal LUTs, 1x LUT with SR-16 mode, 4x FFs (one LUT in each CLB can be used as a 1 to16-bit shift register)
I-0C	2x normal LUTs, 2x LUTs with SR-16 mode, 4x FFs (two LUTs in each CLB can be used as a 1 to 16-bit shift register)
I-0D	1x normal LUT, 3x LUTs with SR-16 mode, 4x FFs (three LUTs in each CLB can be used as a 1 to 16-bit shift register)
I-0E	4x LUTs with SR-16 mode, 4x FFs (all four LUTs in each CLB can be used as a 1 to 16-bit shift register)
I-2A	4x normal LUTs, 5x FFs (adds 1 additional independent FF per CLB)
I-2B	3x normal LUTs, 1x LUT with SR-16 mode, 5x FFs (one LUT in each CLB can be used as a 1 to 16-bit shift register and adds 1 additional FF per CLB)
I-2C	2x normal LUTs, 2x LUTs with SR-16 mode, 5x FFs (two LUTs in each CLB can be used as a 1 to 16-bit shift register and adds 1 additional FF per CLB)
I-2D	1x normal LUT, 3x LUTs with SR-16 mode, 5x FFs (three LUTs in each CLB can be used as a 1 to 16-bit shift register and adds 1 additional FF per CLB)
I-2E	4x LUTs with SR-16 mode, 5x FFs (all four LUTs in each CLB can be used as a 1 to 16-bit shift register and adds 1 additional FF per CLB)
I-4A	4x normal LUTs, 6x FF (adds 2 additional independent FF per CLB)
I-4B	3x normal LUTs, 1x LUT with SR-16 mode, 6x FFs (one LUT in each CLB can be used as a 1 to 16-bit shift register and adds 2 additional FFs per CLB)
I-4C	2x normal LUTs, 2x LUTs with SR-16 mode, 6x FFs (two LUTs in each CLB can be used as a 1 to 16-bit shift register and adds 2 additional FFs per CLB)
I-4D	1x normal LUT, 3x LUTs with SR-16 mode, 6x FFs (three LUTs in each CLB can be used as a 1 to 16-bit shift register and adds 2 additional FFs per CLB)
I-4E	4x LUTs with SR-16 mode, 6x FFs (all four LUTs in each CLB can be used as a 1 to 16-bit shift register and adds 2 additional FFs per CLB)

Arch	Normal LUTs /CLB	SR-16 LUTs/CLB	FF/CLB	IndFF	ANSR	RegPerSR	ULReg	Additional IO Pins
I-0A	4	0	4	1	0	-	1	0
I-0B	3	1			0.25	1,1.5, 2	1.25, 1.375, 1.5	
I-0C	2	2			0.5	1, 1.5, 2	1.5, 1.75, 2	
I-0D	1	3			0.75	1, 1.5, 2	1.75, 2.125, 2.5	
I-0E	0	4			1	1, 1.5, 2	2, 2.5, 3	
I-2A	4	0	5	1.25	0	-	1.25	2
I-2B	3	1			0.25	1, 1.5, 2	1.5, 1.625, 1.75	
I-2C	2	2			0.5	1, 1.5, 2	1.75, 2, 2.25	
I-2D	1	3			0.75	1, 1.5, 2	2, 2.375, 2.75	
I-2E	0	4			1	1, 1.5, 2	2.25, 2.75, 3.25	
I-4A	4	0	6	1.5	0	-	1.5	4
I-4B	3	1			0.25	1, 1.5, 2	1.75, 1.875, 2	
I-4C	2	2			0.5	1, 1.5, 2	2, 2.25, 2.5	
I-4D	1	3			0.75	1, 1.5, 2	2.25, 2.625, 3	
I-4E	0	4			1	1, 1.5, 2	2.5, 3, 3.5	

Architecture *I-2A* returns to the basic four 4-LUT architecture but adds one additional flip-flop per CLB, bringing the total to five independently accessible flip-flops. This requires 2 additional I/O pins to each CLB. Since each BLE in this architecture contains 1.25 flip-flops, *IndFF* = 1.25. Since the LUTs in this architecture cannot be used as shift registers, *ANSR* = 0. This makes *ULReg* = 1.25. As with the *I-0B* through *I-0E* architectures, architectures *I-2B* through *I-2E* add the capability for some of the LUTs in each CLB to be used as an SR-16. In a similar manner, architecture *I-4A* adds two additional flip-flops per CLB to the basic architecture and architectures *I-4B* through *I-4E* allow some of the LUTs in the device to be used as SR-16s.

The architectures used in this testing stop at two additional flip-flops per CLB because this requires four additional CLB I/O pins. All of the architectures discussed in this chapter add four or fewer additional CLB inputs or outputs because the basic architecture requires 28 CLB inputs and outputs (16 inputs/4 outputs for the four 4-LUTs and 4 inputs/4 outputs for the four flip-flops). An additional four CLB I/O pins result in 14% more signals for the connection blocks to deal with. Since the fundamental philosophy this chapter began with tries to limit the impact of any architectural modifications, limiting the number of additional CLB inputs and outputs to 14% will likely cover all of the architectures that are of interest.

The efficiency of these fifteen architectures in mapping heavily registered netlists was tested by applying Equations 8.9 and 8.10 to the 22 depth = 1 netlists and 22 depth = 0.33 netlists used in the previous chapters. Figure 8.14 and Figure 8.15 show the geometric mean number of BLEs that these netlists require when mapped to each of the various architectures. All of these values are normalized to the number of BLEs required by the depth = N netlists.

Although all of these architectures reduce the number of BLEs that these netlists require, adding the capability of using some of the LUTs in the device as SR-16s has a much larger effect than adding additional flip-flops. As shown in Figure 8.15, adding SR-16 capabilities to all four LUTs in each CLB (architecture *I-0E*) halves the number of BLEs that are required by the basic system for the depth = 0.33 netlists (3.133x versus 6.172x the number of BLEs required by the original netlists represents a 0.508x improvement, assuming that an average of 1.5 registers can be mapped to each shift register). This is a feat that even adding 2 additional flip-flops per CLB (architecture *I-4A*) cannot achieve. The *I-4A* architecture only reaches a 0.667x improvement over the basic *I-0A* architecture. Furthermore, adding SR-16 capabilities does not require adding any additional I/O pins. Largely, this behavior occurs because although the *I-0E* architecture has a smaller *IndFF* value than the *I-4A* architecture (1 versus 1.5), it has a larger *ULReg* value (between 2 and 3 versus 1.5). Since the depth = 0.33 netlists have so many BLEs with an unoccupied LUT, the *ULReg* value has a much larger impact on the number of BLEs that is required.

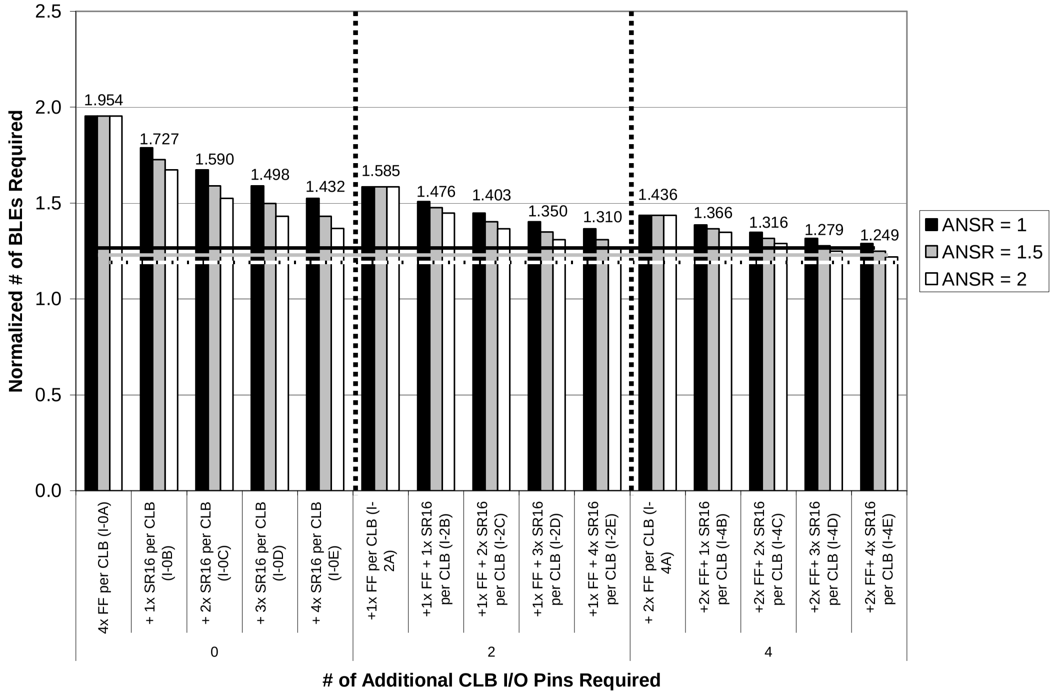


Figure 8.14: Testing Phase I – Effect of Additional Independent Flip-Flops and Shift Registers on Depth = 1 Netlists

Numbers provided indicate *ANSR* = 1.5 results

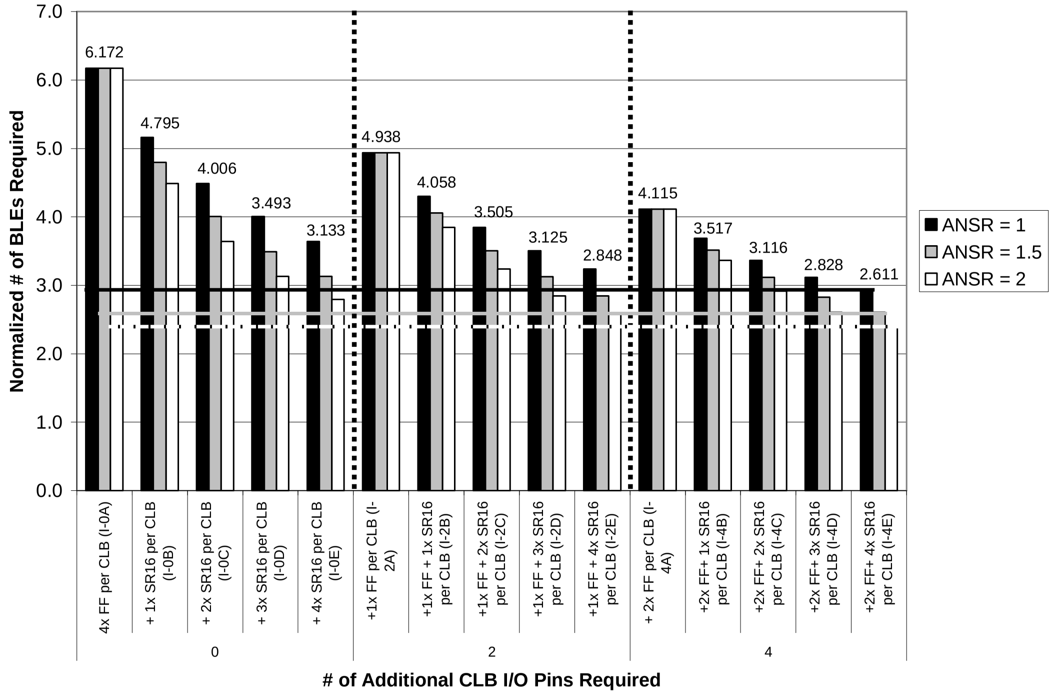


Figure 8.15: Testing Phase I – Effect of Additional Independent Flip-Flops and Shift Registers on Depth = 0.33 Netlists

Numbers provided indicate *ANSR* = 1.5 results

This phenomenon can also be seen elsewhere in Figure 8.15. For the depth = 0.33 netlists, adding the capability for one of the LUTs in each CLB to be used as a SR-16 (architecture *I-0B*) produces approximately the same results as adding one additional flip-flop (architecture *I-2A*). Again assuming that an average of 1.5 registers can be mapped to each shift register, the *I-0B* architecture requires an average of 4.795x the number of BLEs in the original netlists while the *I-2A* architecture requires an average of 4.938x. Furthermore, adding the capability for two of the LUTs in each CLB to be used as SR-16s (architecture *I-0C*) produces approximately the same results as adding two additional flip-flops (architecture *I-4A*). The *I-0C* architecture requires an average of 4.006x the number of BLEs in the original netlists while the *I-4A* architecture requires an average of 4.115x. This makes sense because for the BLEs in the system with an unoccupied LUT, having one SR-16 is essentially the same as having one extra flip-flop, even assuming the worse case where *RegPerSR* = 1. Both architectures have *ULReg* = 1.25. Similarly, having two SR-16s in each CLBs is essentially the same as having two extra flip-flops. Again, even assuming the worse case, both architectures have *ULReg* = 1.5.

However, this characteristic only applies to BLEs that have an unoccupied LUT. The depth = 1 netlists shown in Figure 8.14 require far fewer additional BLEs. Thus, these netlists have a much smaller ratio of BLEs with unoccupied LUTs. This affects the results. For these netlists, to produce approximately the same results as adding one additional flip-flop (architecture *I-2A*), two of the LUTs in each CLB need SR-16 capabilities (architecture *I-0C*). Assuming that an average of 1.5 registers can be mapped to each shift register, the *I-0C* architecture requires an average of 1.590x the number of BLEs in the original netlists while the *I-2A* architecture requires an average of 1.585x. Similarly, to produce approximately the same results as adding two additional flip-flops (architecture *I-4A*), all four of the LUTs in each CLB need SR-16 capabilities (architecture *I-1E*). The *I-0E* architecture requires an average of 1.432x the number of BLEs in the original netlists while the *I-4A* architecture requires an average of 1.436x. More SR-16s are required to match the results produced by adding additional flip-flops because the number of BLEs with unoccupied LUTs in the depth = 1 netlists no longer dwarfs the number of BLEs with an occupied LUT. This makes the *ULReg* value less important and the *IndFF* value more significant. That said, the benefit of allowing LUTs to be used as SR-16s is still impressive, particularly because this does not increase the number of CLB I/O pins.

Adding independent flip-flops or allowing LUTs to be used as SR-16s are not the only ways of adding additional register resources. Although it requires addition CLB output pins, it is also possible to allow each LUT to be used as two 1 to 8-bit shift registers (SR-8s) or four 1 to 4-bit shift registers (SR-4s). Thus, the next phases of testing investigated the efficiency of architectures that had these types of resources. This testing was divided into 3 separate parts. The second phase of testing assumed that each CLB could support converting one LUT in each CLB into one SR-16, two SR-8s or four SR-4s. The third and fourth phases assumed that each CLB could support converting two or four of the LUTs in each CLB, respectively, into shift registers. Again, each of these LUTs could be used as one SR-16, two SR-8s or four SR-4s. The testing was divided in this manner because adding the basic components required to convert a LUT into any kind of shift register might be costly. This may be the reason that Xilinx devices only allow half of their LUTs to be used as SR-16s. As mentioned earlier, adding the capability to turn a 4-LUT into even a SR-16 could require twenty additional 2:1 multiplexers and four additional memory cells. Depending upon the structure used for the basic LUT and the necessary transistor sizing that is required, this may be significant. Thus, each of the subsequent phases of testing explored what could be accomplished by adding shift registers and additional flip-flops to the system while limiting the number of modified LUTs.

As shown in Table 8.4, the second phase of testing compared the mapping efficiency of seven architectures. Four of the architectures are from the first phase of testing: *I-0A*, *I-0B*, *I-2B*, *I-4B*. Listed first is the default architecture, *I-0A*. Next comes all of the architectures that can be made that do not require any additional I/O pins and have exactly one modified LUT per CLB. One architecture can be made that has one LUT in each CLB that can be used as a SR-16, architecture *I-0B* from the first round of testing. Next comes all of the architectures that can be made that require one additional I/O pin and have exactly one modified LUT per CLB. Only one architecture can be made, *II-1A*, an architecture that allows one of the LUTs in each CLB to be used as two SR-8s. This process continues for two, three and four additional I/O pins per CLB.

As seen in Figure 8.16 and Figure 8.17, for architectures in which one unoccupied LUT can be converted into one or more shift registers, increasing the number of I/O pins in each CLB is relatively compelling. While adding the capability of converting one LUT per CLB into a SR-16 provides some benefit (0.884x improvement over the default architecture for the depth = 1 netlists and 0.777x improvement over the default architecture for the depth = 0.33 netlists), a larger improvement comes from mapping the circuits to architectures that have CLBs with additional I/O pins (up to a 0.699x improvement over the default architecture for the depth = 1 netlists and up to a 0.508x improvement over the default architecture for the depth = 0.33 netlists). However, for the depth = 1 netlists, the benefits of creating more sophisticated architectures largely drops off at architecture *II-3A*, a device with one additional flip-flop and one LUT that can be converted into two SR-8s per CLB. Architecture *II-3B*, a system with one LUT that can that be converted into four SR-4s per CLB is less compelling. While architecture *II-3A* improved the number of BLEs required by 0.718x compared to the default architecture (1.403x versus 1.954x, again assuming that an average of 1.5 registers can be mapped to each shift register), the *II-3B* architecture only improve the number of BLEs by 0.733x (1.432x versus 1.954x). This is largely because the depth = 1 netlists do not require a huge number of BLEs with unoccupied LUTs. Thus, having a larger *IndFF* is preferable to a larger *ULReg*. (when *RegPerSR* = 1, *IndFF* = 1.25 and *ULReg* = 1.75 versus *IndFF* = 1 and *ULReg* = 2)

However, for the depth = 0.33 netlists, architecture *II-3B* provides better performance than architecture *II-3A*. Architecture *II-3B* improved the number of required BLEs by 0.508x while architecture *II-3A* only improved the number of required BLEs by 0.568x. This is because the depth = 0.33 netlists require more registers, creating a larger fraction of BLEs with unoccupied LUTs. That said, architecture *I-4B* can be essentially disregarded. This is because while it requires more additional I/O pins than either *II-3A* or *II-3B*, it does not produce dramatically better results than *II-3A* for the depth = 1 netlists (1.366x versus 1.403x) and produces worse results than *II-3B* for the depth = 0.33 netlists (3.517x versus 3.133x).

As shown in Table 8.5, the third phase of testing compared the mapping efficiency of ten architectures. Again, four of the architectures are from the first phase of testing: *I-0A*, *I-0C*, *I-2C*, *I-4C*. The remaining six were generated using the same methodology as in the second testing phase: all of the architectures that

Table 8.4: Architectures Used in Testing Phase II –
Adding Independent Flip-Flops and Shift Registers, 1 Modified LUT/CLB

Arch	Description – Contents of Each CLB
I-0A*	4x normal LUTs, 4x FFs (default architecture)
I-0B*	3x normal LUTs, 1x LUT with SR-16 mode, 4x FFs (one LUT in each CLB can be used as a 1 to 16-bit shift register)
II-1A	3x normal LUTs, 1x LUT with 2x SR-8 mode, 4x FFs (one LUT in each CLB can be used as two 1 to 8-bit shift registers)
I-2B*	3x normal LUTs, 1x LUT with SR-16 mode, 5x FFs (one LUT in each CLB can be used as a 1 to 16-bit shift register and 1 additional FF per CLB is added)
II-3A	3x normal LUTs, 1x LUT with 2x SR-8 mode, 5x FFs (one LUT in each CLB can be used as two 1 to 8-bit shift registers and 1 additional FF per CLB is added)
II-3B	3x normal LUTs, 1x LUT with 4x SR-4 mode, 4x FFs (one LUT in each CLB can be used as four 1 to 4-bit shift registers)
I-4B*	3x normal LUTs, 1x LUT with SR-16 mode, 6x FFs (one LUT in each CLB can be used as a 1 to 16-bit shift register and 2 additional FFs per CLB are added)

Arch	Normal LUTs /CLB	SR-16 LUTs /CLB	2x SR-8 LUTs /CLB	4x SR-4 LUTs /CLB	FF /CLB	IndFF	ANSR	RegPerSR	ULReg	Extra IO Pins
I-0A*	4	0	0	0	4	1	0	-	1	0
I-0B*	3	1	0	0	4	1	0.25	1	1.25	
								1.5	1.375	
								2	1.5	
II-1A	3	0	1	0	4	1	0.5	1	1.5	1
								1.5	1.75	
								2	2	
I-2B*	3	1	0	0	5	1.25	0.25	1	1.5	2
								1.5	1.625	
								2	1.75	
II-3A	3	0	1	0	5	1.25	0.5	1	1.75	3
								1.5	2	
								2	2.25	
II-3B	3	0	0	1	4	1	1	1	2	
								1.5	2.5	
								2	3	
I-4B*	3	1	0	0	6	1.5	0.25	1	1.75	4
								1.5	1.875	
								2	2	

* Denotes architecture from Phase I testing

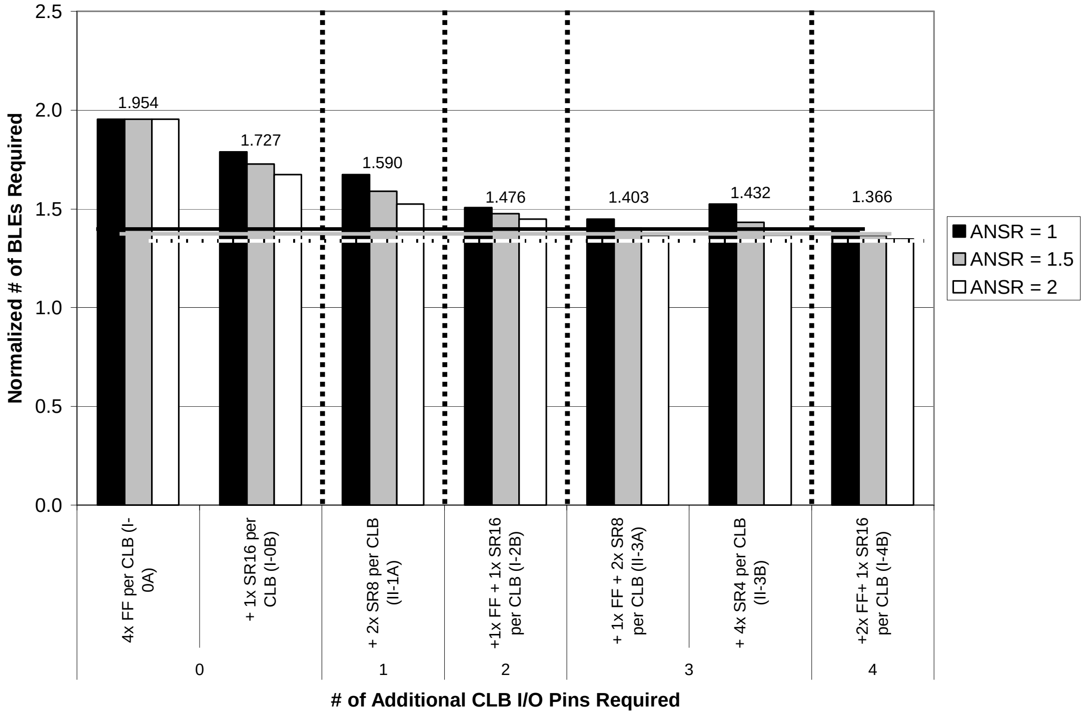


Figure 8.16: Architecture Exploration on Depth = 1 Netlists, 1 LUT/CLB has Shift Register(s)

Numbers provided indicate *ANSR* = 1.5 results

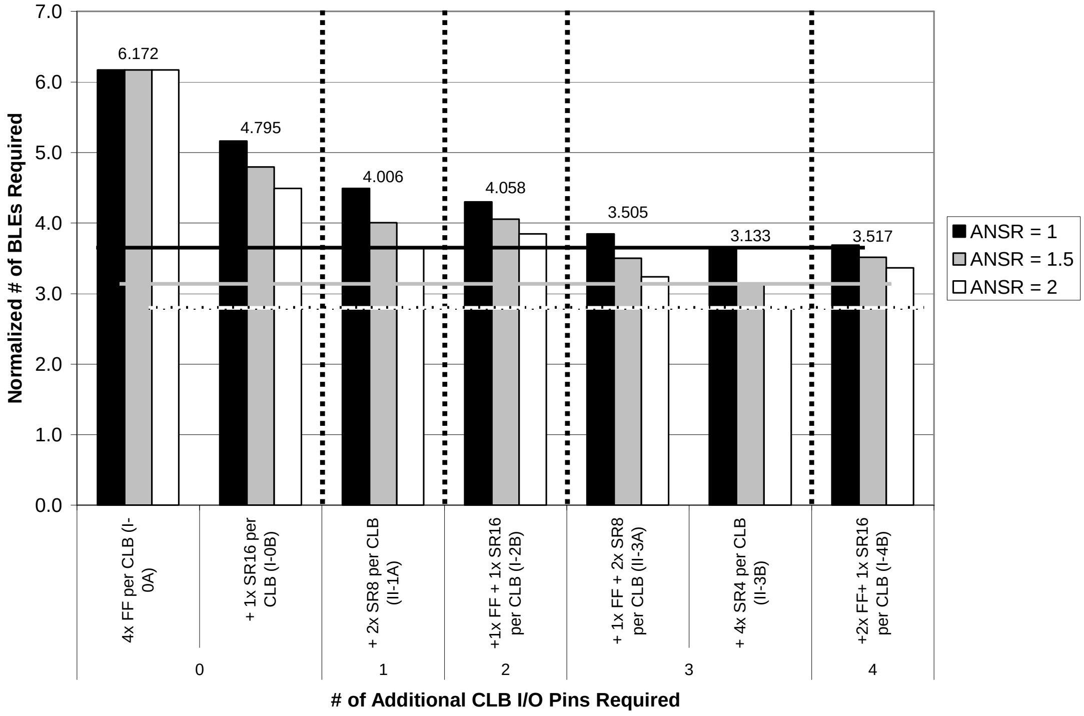


Figure 8.17: Architecture Exploration on Depth = 0.33 Netlists, 1 LUT/CLB has Shift Register(s)

Numbers provided indicate *ANSR* = 1.5 results

require one through four additional I/O pins that have exactly two modified LUTs per CLB were investigated.

As seen in Figure 8.18 and Figure 8.19, for architectures in which two unoccupied LUTs can be converted into two or more shift registers, increasing the number of I/O pins in each CLB is still relatively compelling. Although adding the capability of converting two of the LUTs in each CLB into SR-16s provides the largest benefit (by itself a 0.814x improvement for the depth = 1 netlists and a 0.649x improvement for the depth = 0.33 netlists), mapping to architectures that have CLBs with additional I/O pins also produces a relatively significant improvement (up to 0.670x improvement for the depth = 0.33 netlists and up to 0.431x improvement for the depth = 0.33 netlists). That said, as with the architectures in the previous round of testing, the benefits of creating more sophisticated architectures largely drops off at three additional I/O pins. All three architectures with two modified LUTs and four additional I/O pins per CLB can likely be eliminated from consideration since they do not provide enough benefit to justify the additional I/O pins required.

The architectures that worked best in this testing also showed the same netlist dependence as in the previous round of testing. While the depth = 1 netlists preferred architectures *I-2C* and *III-3A* for devices with 2 and 3 additional I/O pins respectively, the depth = 0.33 netlists preferred architectures *III-2A* and *III-3B*. The *I-2C* and *III-3A* architectures provided a 0.718x and 0.691x improvement, respectively, for the depth = 1 netlists and the *III-2A* and *III-3B* architectures provided a 0.508x and 0.464x improvement, respectively, for the depth = 0.33 netlists. Again, this is because the depth = 1 netlists prefers architectures that trade a slightly higher *IndFF* for a slightly lower *ULReg* while the depth = 0.33 netlists prefer architectures that trade a slightly higher *ULReg* for a slightly lower *IndFF*.

As shown in Table 8.6, the last phase of testing compared the mapping efficiency of twelve architectures that had four modified LUTs per CLB. As seen in Figure 8.20 and Figure 8.21, these architectures showed the most improvement by simply allowing all four of the LUTs in each CLB to be used as SR-16s. By itself, this represented a 0.733x improvement over the default architecture for the depth = 1 netlists and a 0.508x improvement for the for the depth = 0.33 netlists. However, as with the previous two tests, both the depth = 1 and depth = 0.33 netlists showed marked improvement for architectures that added up to three additional I/O pins. Again, architectures with four modified LUTs and four additional I/O pins per CLB can likely be eliminated from consideration since they do not provide a sizeable benefit over the architectures that require three additional I/O pins. Furthermore, also like the previous round of testing, the architectures that showed the best performance for the depth = 1 netlists had a slightly higher *IndFF* and the architectures that showed the best performance for the depth = 0.33 netlists had a slightly higher *ULReg*.

Table 8.5: Architectures Used in Testing Phase III –
Adding Independent Flip-Flops and Shift Registers, 2 Modified LUTs/CLB

Arch	Description – Contents of Each CLB
I-0A*	4x normal LUTs, 4x FFs (default architecture)
I-0C*	2x normal LUTs, 2x LUTs with SR-16 mode, 4x FFs (two LUTs in each CLB can each be used as a 1 to 16-bit shift register)
III-1A	2x normal LUTs, 1x LUT with SR-16 mode, 1x LUT with 2x SR-8 mode, 4x FFs (one LUT in each CLB can be used as a 1 to 16-bit shift register and one LUT in each CLB can be used as two 1 to 8-bit shift registers)
I-2C*	2x normal LUTs, 2x LUTs with SR-16 mode, 5x FFs (two LUTs in each CLB can each be used as a 1 to 16-bit shift register and 1 additional FF per CLB is added)
III-2A	2x normal LUTs, 2x LUTs with 2x SR-8 mode, 4x FFs (two LUTs in each CLB can each be used as two 1 to 8-bit shift registers)
III-3A	2x normal LUTs, 1x LUT with SR-16 mode, 1x LUT with 2x SR-8 mode, 5x FFs (one LUT in each CLB can be used as a 1 to 16-bit shift register, one LUT in each CLB can be used as two 1 to 8-bit shift registers, and 1 additional FF per CLB is added)
III-3B	2x normal LUTs, 1x LUT with SR-16 mode, 1x LUT with 4x SR-4 mode, 4x FFs (one LUT in each CLB can be used as a 1 to 16-bit shift register and one LUT in each CLB can be used as four 1 to 4-bit shift registers)
I-4C*	2x normal LUTs, 2x LUTs with SR-16 mode, 6x FFs (two LUTs in each CLB can each be used as a 1 to 16-bit shift register and 2 additional FFs per CLB are added)
III-4A	2x normal LUTs, 2x LUTs with 2x SR-8 mode, 5x FFs (two LUTs in each CLB can each be used as two 1 to 8-bit shift registers and 1 additional FF per CLB is added)
III-4B	2x normal LUTs, 1x LUT with 2x SR-8 mode, 1x LUT with 4x SR-4 mode, 4x FFs (one LUT in each CLB can be used as two 1 to 8-bit shift registers and one LUT in each CLB can be used as four 1 to 4-bit shift registers)

Arch	Normal LUTs /CLB	SR-16 LUTs /CLB	2x SR-8 LUTs /CLB	4x SR-4 LUTs /CLB	FF /CLB	IndFF	ANSR	RegPerSR	ULReg	Extra IO Pins
I-0A*	4	0	0	0	4	1	0	-	1	0
I-0C*	2	2	0	0	4	1	0.5	1 1.5 2	1.5 1.75 2	
III-1A	2	1	1	0	4	1	0.75	1 1.5 2	1.75 2.125 2.5	1
I-2C*	2	2	0	0	5	1.25	0.5	1 1.5 2	1.75 2 2.25	2
III-2A	2	0	2	0	4	1	1	1 1.5 2	2 2.5 3	
III-3A	2	1	1	0	5	1.25	0.75	1 1.5 2	2 2.375 2.75	3
III-3B	2	1	0	1	4	1	1.25	1 1.5 2	2.25 2.875 3.5	
I-4C*	2	2	0	0	6	1.5	0.5	1 1.5 2	2 2.25 2.5	4
III-4A	2	0	2	0	5	1.25	1	1 1.5 2	2.25 2.75 3.25	
III-4B	2	0	1	1	4	1	1.5	1 1.5 2	2.5 3.25 4.0	

* Denotes architecture from Phase I testing

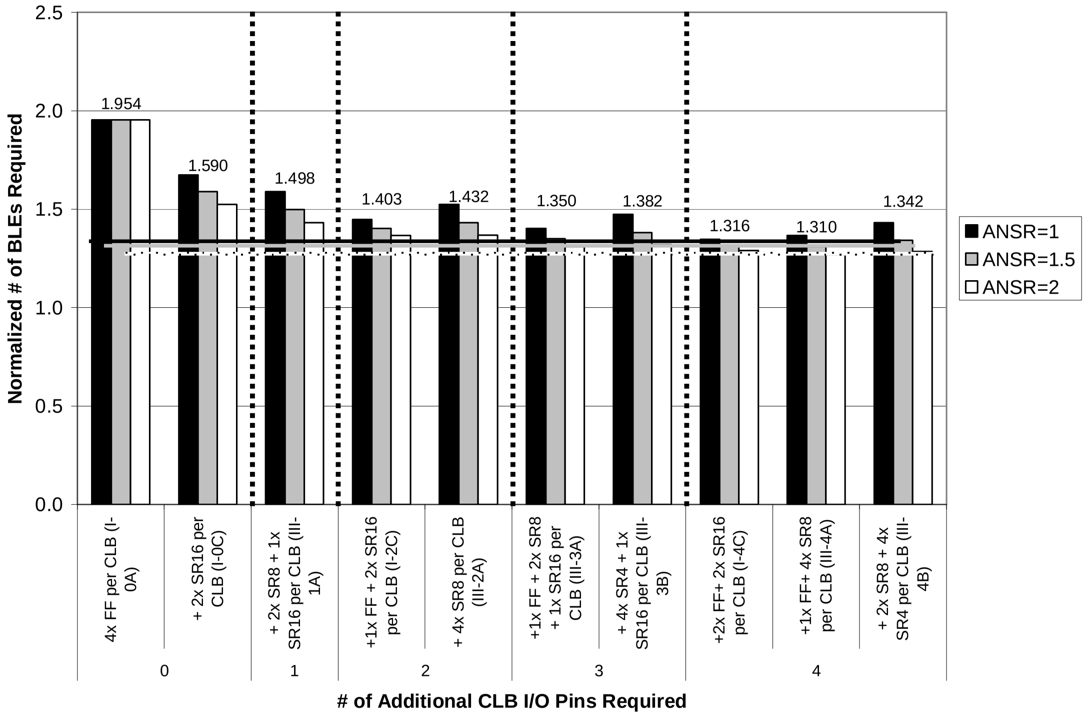


Figure 8.18: Architecture Exploration on Depth = 1 Netlists, 2 LUTs/CLB have Shift Register(s)
Numbers provided indicate *ANSR* = 1.5 results

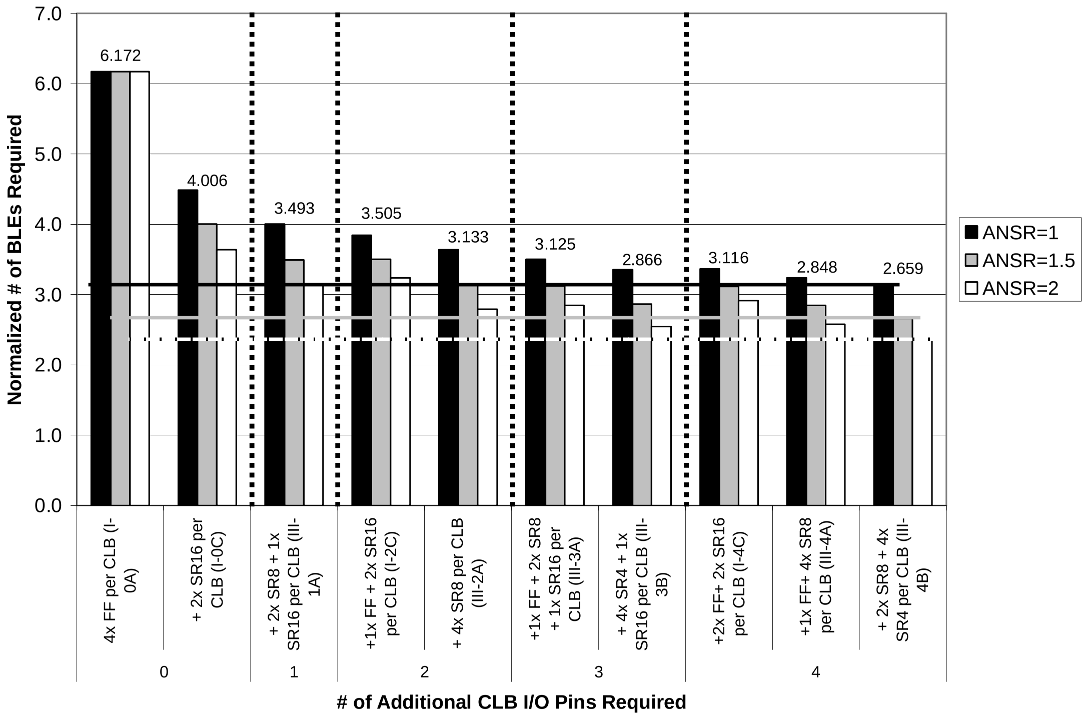


Figure 8.19: Architecture Exploration on Depth = 0.33 Netlists, 2 LUTs/CLB have Shift Register(s)
Numbers provided indicate *ANSR* = 1.5 results

Table 8.6: Architectures Used in Testing Phase IV – Adding Independent Flip-Flops and Shift Registers, 4 Modified LUTs/CLB

Arch	Description – Contents of Each CLB
I-0A*	4x normal LUTs, 4x FFs (default architecture)
I-0E*	4x LUTs with SR-16 mode, 4x FFs (all four LUTs in each CLB can be used as a 1 to 16-bit shift register)
IV-1A	3x LUTs with SR-16 mode, 1x LUT with 2x SR-8 mode, 4x FFs (three LUTs in each CLB can be used as a 1 to 16-bit shift register and one LUT in each CLB can be used as two 1 to 8-bit shift registers)
I-2E*	4x LUTs with SR-16 mode, 5x FFs (all four LUTs in each CLB can be used as a 1 to 16-bit shift register and 1 additional FF per CLB is added)
IV-2A	2x LUTs with SR-16 mode, 2x LUTs with 2x SR-8 mode, 4x FFs (two LUTs in each CLB can be used as a 1 to16-bit shift register and two LUTs in each CLB can be used as two 1 to 8-bit shift registers)
IV-3A	3x LUTs with SR-16 mode, 1x LUT with 2x SR-8 mode, 5x FFs (three LUTs in each CLB can be used as a 1 to 16-bit shift register, one LUT in each CLB can be used as two 1 to 8-bit shift registers, and 1 additional FF per CLB is added)
IV-3B	3x LUTs with SR-16 mode, 1x LUT with 4x SR-4 mode, 4x FFs (three LUTs in each CLB can be used as a 1 to 16-bit shift register and one LUT in each CLB can be used as four 1 to 4-bit shift registers)
IV-3C	1x LUTs with SR-16 mode, 3x LUT with 2x SR-8 mode, 4x FFs (one LUT in each CLB can be used as a 1 to 16-bit shift register and three LUTs in each CLB can be used as two 1 to 8-bit shift registers)
I-4E*	4x LUTs with SR-16 mode, 6x FFs (all four LUTs in each CLB can be used as a 1 to 16-bit shift register and 2 additional FFs per CLB are added)
IV-4A	2x LUTs with SR-16 mode, 2x LUTs with 2x SR-8 mode, 5x FFs (two LUTs in each CLB can be used as a 1 to16-bit shift register, two LUTs in each CLB can each be used as two 1 to 8-bit shift registers and 1 additional FF per CLB is added)
IV-4B	2x LUTs with SR-16 mode, 1x LUT with 2x SR-8 mode, 1x LUT with 4x SR-4 mode, 4x FFs (two LUTs in each CLB can be used as a 1 to16-bit shift register, one LUT in each CLB can be used as two 1 to 8-bit shift registers and one LUT in each CLB can be used as four 1 to 4-bit shift registers)
IV-4C	4x LUTs with 2x SR-8 mode, 4x FFs (all four LUTs in each CLB can be used as two 1 to 8-bit shift registers)

Arch	Normal LUTs /CLB	SR-16 LUTs /CLB	2x SR-8 LUTs /CLB	4x SR-4 LUTs /CLB	FF /CLB	IndFF	ANSR	RegPerSR	ULReg	Extra IO Pins
I-0A*	4	0	0	0	4	1	0	-	1	0
I-0E*	0	4	0	0	4	1	1	1 1.5 2	2 2.5 3	
IV-1A	0	3	1	0	4	1	1.25	1 1.5 2	2.25 2.875 3.5	1
I-2E*	0	4	0	0	5	1.25	1	1 1.5 2	2.25 2.75 3.25	2
IV-2A	0	2	2	0	4	1	1.5	1 1.5 2	2.5 3.25 4.0	
IV-3A	0	3	1	0	5	1.25	1.25	1 1.5 2	2.5 3.125 3.75	3
IV-3B	0	3	0	1	4	1	1.75	1 1.5 2	2.75 3.625 4.5	
IV-3C	0	1	3	0	4	1	1.75	1 1.5 2	2.75 3.625 4.5	
I-4E*	0	4	0	0	6	1.5	1	1 1.5 2	2.5 3 3.5	4
IV-4A	0	2	2	0	5	1.25	1.5	1 1.5 2	2.75 3.5 4.20	
IV-4B	0	2	1	1	4	1	2	1 1.5 2	3 4 5	
IV-4C	0	0	4	0	4	1	2	1 1.5 2	3 4 5	

* Denotes architecture from Phase I testing

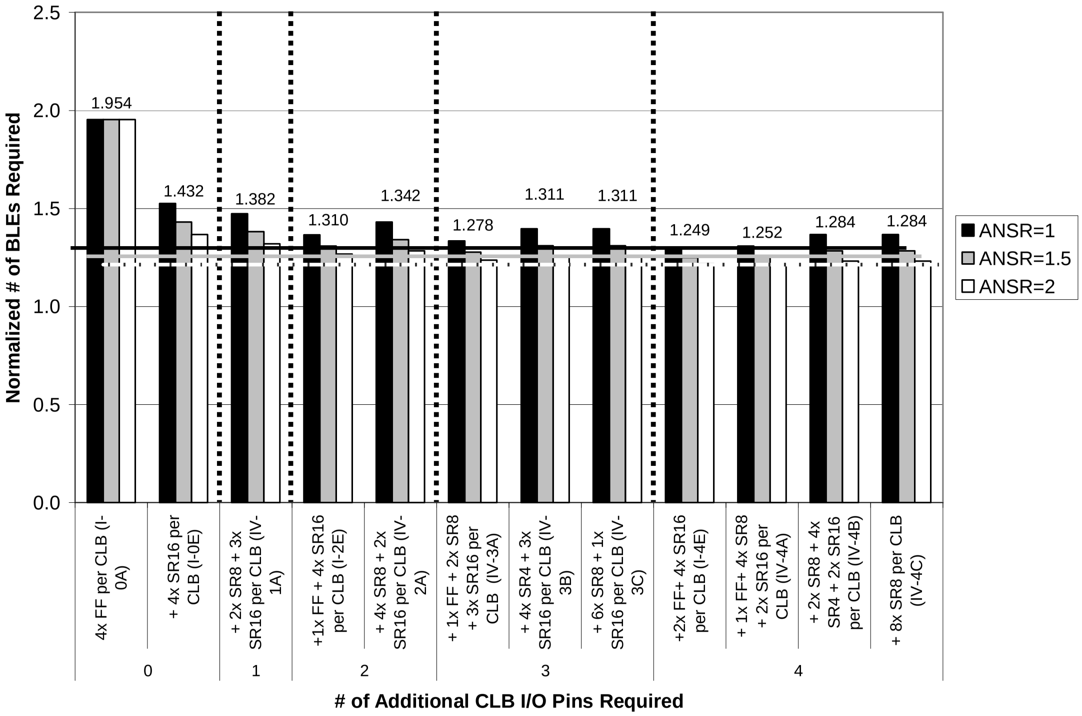


Figure 8.20: Architecture Exploration on Depth = 1 Netlists, 4 LUTs/CLB have Shift Register(s)
Numbers provided indicate *ANSR* = 1.5 results

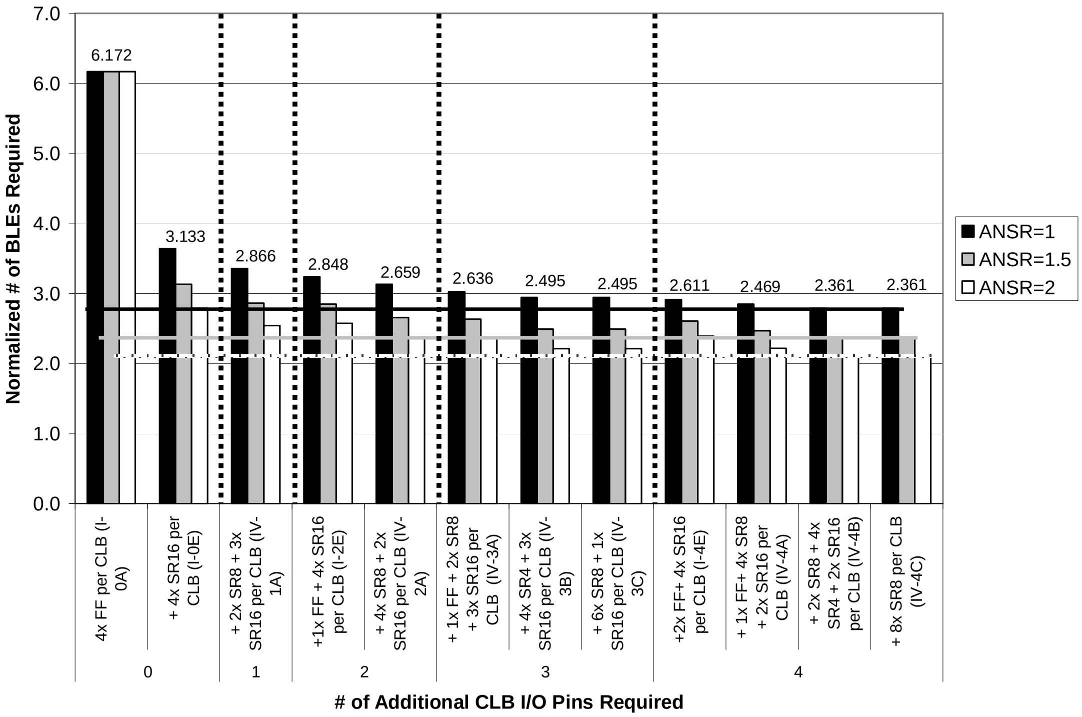


Figure 8.21: Architecture Exploration on Depth = 0.33 Netlists, 4 LUTs/CLB have Shift Register(s)
Numbers provided indicate *ANSR* = 1.5 results

8.5: Conclusions and Future Research

This chapter examined the potential benefits of increasing the number of registers that FPGAs provide. Heavy pipelining and C-slowing can add a huge number of registers into a netlist. Although this can improve the critical path delay significantly, this also considerably increases the number of BLEs that are required. Since conventional architectures offer BLEs with one LUT and one flip-flop, circuits that have a uniform logic depth can be pipelined or C-slowed to a logic depth of one LUT and not require a large number of additional logic blocks. This is because the registers in these circuits can be put into the same BLEs as the logic. However, many circuits have some paths that go though several layers of logic while the rest of the circuit goes through relatively few. Since the number of registers added to all paths must be the same to maintain the functionality of the circuit, deeply pipelining or C-slowing these non-uniform netlists may tremendously increase the number of BLEs that are needed. This is because many registers will have to be added to the short paths in order to fully pipeline the long paths. For that matter, FPGA application developers often purposefully add multiple registers to signals in their circuits. This is done to allow the delay through long wires to be broken up across multiple clock cycles.

There have been multiple previous research efforts that have investigated potential ways of increasing the number of registers that FPGAs provide. Unfortunately, these systems typically impose strict limitations on the types of circuits they can implement or have an unacceptably large area overhead when mapping more conventional, lightly-registered applications. Since these architectures are not practical for the majority of users, they are not commercially viable. This chapter addresses the problem of introducing additional registers into an FPGA in a fundamentally different way. Rather than making drastic changes that alter the basic usability of entire system, it is likely more reasonable to add small modifications that are beneficial to heavily registered applications, but largely invisible to more conventional circuits.

There are two basic areas of an FPGA that additional registers can be incorporated: the routing network and the logic blocks. While some previous architectures have added registers into the interconnect by embedding registers inside switchboxes, this really introduces more problems than it solves. Adding registers that are connected to all of the wires that enter and exit a switchbox is not practical due to area concerns. However, introducing registers into the system that have more limited input and output options changes the fundamental nature of the placement and routing problems. As discussed in the previous chapter, this kind of architecture can require pipeline-aware routing algorithms.

From a performance standpoint, embedding registers in the interconnect switchboxes does not make the system considerably faster compared to using registers that might be found in more conventional logic block locations. Although signals that use registers in the interconnect do not incur the delay associated with entering and exiting a CLB, they cannot escape the largest component of delay in modern FPGAs: the delay though the wires themselves. Furthermore, it is likely that wire delay will become a larger part of the overall delay in the system in future process technologies. Using the values provided by VPR for 0.65nm FPGAs, devices with registers embedded in the interconnect not only require more complicated CAD tools, they are likely less than 0.92x faster than more conventional architectures.

However, this is not to say that embedding registers within the routing fabric cannot help the performance of any FPGA, only that it does not make sense to incorporate these kinds of resources in architectures that only have conventional island-style routing wires. For example, in an architecture with dedicated local connections, embedding registers along these wires could be quite helpful. If each CLB has a direct connection to each of its 8 or 24 nearest neighbors, adding a register to split the delay through the wire could make these connections much faster. Alternatively, this could allow the system to use smaller drivers for these connections and still have them keep up with other faster routing resources in the architecture. Furthermore, adding registers to these dedicated connections does not create a problem for the router. This is because the endpoints of these connections are already fixed. Thus, these wires are not shared resources that can run into congestion resolution problems if registers are assigned to these locations during placement.

That said, focusing FPGA architects on incorporating additional registers into the logic blocks themselves is probably a better idea. However, there are multiple ways that this could be accomplished. One option is to simply add more independently accessible flip-flops to each CLB. Another way is to harness the memory cells that already exist within the LUTs themselves. With a few minor modifications, any LUT that is not needed to implement the logic of the circuit can be converted into one 1 to 16-bit shift register, two 1 to 8-bit shift registers or four 1 to 4-bit shift registers.

However, there are several practical concerns that should be kept in mind when evaluating any modifications to the system. First, there is the number of input and output pins that each of these various enhancements add to the CLB in which they are placed. Each additional flip-flop that is added to a CLB also adds one additional input pin and one additional output pin. Furthermore, while any LUT that is converted into an SR-16 can utilize the existing CLB inputs and outputs, LUTs that are converted into two SR-8s or four SR-4s require one or three additional output pins, respectively.

The next issue is the basic usability of shift registers. Although deeply pipelined and C-slowed circuits contain a large total number of registers, they are generally evenly distributed throughout the circuit. The majority of signals in even the most heavily registered netlists require less than four registers. For that matter, for performance reasons these registers generally must be mapped to multiple locations between the source and sink. Thus, each shift register will probably only be used to implement one or two registers.

For this reason, while splitting a LUT into multiple shift registers requires more resources, this might be a good idea since this gives the LUT the capability to provide registers to multiple different signals.

The last issue concerns the difference in local availability between individual flip-flops and shift registers. The ratio of logic to register resources within certain parts of the device likely limits the performance benefits of heavily pipelining or C-slowing a netlist. This is because beyond a certain critical threshold of pipelining or C-slowing, the circuit must spread out to accumulate all of the necessary registers in the netlist. This can counteract the benefits of performing pipelining or C-slowing in the first place. Although shift registers provide very dense register resources, they can only be implemented in LUTs that are not used for logic. Thus, for these to be inserted into tightly knit computational kernels, the LUTs in the circuit must be spread out to provide empty LUT locations. Fundamentally, while shift registers are good at raising the average number of registers in the chip as a whole, they have a hard time increasing the local density of register resources where they might be needed. On the other hand, adding independently accessible flip-flops to CLBs inherently evenly raises the ratio of register resources to logic. This means that although each register might be more expensive, they may be more useful.

These concerns regarding how different types of resources within potential target architectures interact with each other and the characteristics of incoming netlists were captured using a few equations. These equations considered basic attributes, like the number of LUTs and registers required by a circuit, along with more subtle issues such as the average number of registers that an incoming netlist could map to shift registers that might be in the architecture. These equations were used to evaluate how 32 different architectures handled two different sets of heavily registered circuits, the 11 combinational and 11 sequential MCNC netlists pipelined, C-slowed and retimed to logic depth = 1 and logic depth = 0.33.

This testing showed that the largest gains could be achieved by giving as many LUTs as possible the ability to be used as a 1 to 16-bit shift register. For the depth = 1 netlists, allowing half of the LUTs to be used as SR-16s, like modern Xilinx devices, improved the number of BLEs required by 0.814x over the default architecture that did not contain shift registers. Allowing all of the LUTs to be used as SR-16s reduced the number of BLEs by 0.733x. Similarly, for the depth = 0.33 netlists, allowing half of the LUTs to be used as SR-16s improved the number of BLEs required by 0.649x and allowing all of the LUTs to be used as SR-16s reduced the number of BLEs by an enormous 0.508x.

Although adding additional flip-flops to the CLBs or splitting these shift registers into smaller banks could potentially further improve the mapping efficiency of an architecture, the achievable improvements were comparatively much smaller. Furthermore, these kinds of modifications increase the number of I/O pins that each CLB needs, requiring more extensive changes to the system. Even adding four I/O pins to each

CLB could only improve the results obtained by allowing all four LUTs in each CLB to be used as an SR-16 by 0.872x for the depth = 1 netlists and 0.754x for the depth = 0.33 netlists. Thus, from the standpoint of layout and architectural design, the most important aspect to consider during the development process of an FPGA is how to make the necessary modifications needed to use LUTs as SR-16s as cheap as possible. This is far more important than designing the system to make additional CLB I/O pins inexpensive.

That said, the mapping efficiency of an architecture could possibly be justifiably improved if one or two additional I/O pins could be added to each CLB. This would allow the architecture to split one or two of the SR-16s into two SR-8s or add one additional flip-flop per CLB. For the depth = 1 netlists, adding one additional I/O pin could improve the results by 0.942x over only having four SR-16s and adding two additional I/O pins could improve results by 0.882x. For the depth = 0.33 netlists, adding one additional I/O pin could improve the results by 0.915x over only having four SR-16s and adding two additional I/O pins could improve results by 0.849x. Each additional I/O pin represents a 3.5% increase in the number of inputs and outputs needed by a CLB with four 4-LUTs and four independently accessible flip-flops.

This testing also confirmed that how to best use additional I/O pins depends upon the characteristics of the intended applications. Since shift registers can only be implemented in empty LUT locations, as the ratio of registers to LUTs in a netlist goes down, it becomes more attractive to add independently accessibly flip-flops rather than split shift registers. This is because the number of empty LUT locations in these netlists is naturally smaller, allowing fewer of the shift registers to actually be used. For architectures with the same number of I/O pins, allocating the internal resources differently could result in a 1.119x difference in the number of required BLEs. A corollary to this is the observation that adding or splitting shift registers in an architecture can never allow a heavily registered application to map to exactly the same number of BLEs as an unregistered version. This is because additional BLEs are always necessary in order to provide empty LUT locations that can be used to implement shift registers.

While this insight is a good start, there are still many open questions regarding how different architectures affect the mapping of applications. The most pressing issue is that the number of BLEs required by a netlist was the only metric used to evaluate different architectures. However, this information by itself is not enough to constitute a rigorous architecture exploration. Specifically, more precise area and delay values are necessary. That said, getting this information requires a large amount of additional work.

Although the number of BLEs that a netlist requires strongly affects its silicon footprint and this chapter provided some basic analysis of the relative cost of adding or splitting shift registers and adding flip-flops, exact area numbers for the architectures were never given. Largely, this is because accurately estimating the area of a device requires additional information regarding the transistor and wire-level realities of the various architectural options. Although it is impractical to expect entire FPGAs to be laid out while still at the architectural exploration phase, the basic pieces of candidate systems must be implemented in some way to build a meaningful area model. At the very least, transistor schematics must be made for all of components in an FPGA: LUTs, flip-flops, memory cells, multiplexers, etc. These schematics could be used to build a very rudimentary transistor count area model. However, to be truly useful such a model must account for differences in transistor sizing. While the transistors within a memory cell can likely be close to minimum size, the transistors that drive long interconnect wires will probably need to be much larger. Even better, while entire FPGAs cannot be laid out, specific pieces of the system could be laid out to create a relatively accurate area model.

However, applying this area model in a meaningful way also requires netlists to actually be placed and routed on these architectures. This is because the number of logic blocks in an architecture is only one component of the area requirement of the system as a whole. The interconnect network represent somewhere between 50-90% of the area in modern FPGAs. Since changing some of the fundamental characteristics of the underlying architecture will also likely change the channel width that mapped applications require, this can have a large effect on the overall area of the device.

For that matter, placement and routing is also needed to determine the achievable clock frequency of mapped circuits. Architectural modifications can change the critical path delay of applications in three ways. First, the physical length of each interconnect wire will change because the logic blocks will get bigger or smaller as the contents is varied and the switch boxes will get bigger or smaller as the channel width of the system goes up and down. Since longer wires are naturally slower than shorter wires, this can affect the delay of the entire FPGA. Second, the density of mapped circuits will change. Any increase in the amount of register resources in the system will also allow circuits to be mapped to a smaller number of logic blocks. Since the circuits can fit into a smaller region, this may speed up the achievable clock frequency. Third, as touched on earlier, the specific types of register resources provided by the architecture will alter the logic density that netlists can achieve. For example, it is likely that an architecture with more independent flip-flops will allow logic-constrained netlists to run faster than an architecture with shift registers because additional flip-flops allow the LUTs in computational kernels to be placed more closely together.

Unfortunately, altering the architecture itself can also change the demands on the placement and routing tools. Since an architecture can only perform as well as the CAD tools allow, addressing any issues that arise is crucial to getting an accurate idea of the advantages or disadvantages of an architecture. Some of these problems are relatively straightforward to address, but potentially difficult to actually solve. For example, different architectures may need different placement or routing tuning parameters to produce good results. While the techniques needed to test different parameters are obvious, the tests themselves may require a huge amount of computational resources to evaluate multiple architectures across multiple sets of benchmarks.

Other problems present more fundamental issues. For example, it is not obvious how to fairly evaluate the required size of an application when mapped to an architecture that contains shift registers. Heavily registered circuits, such as the depth = 0.33 netlists, may have many signals with multiple registers. These netlists could be mapped to a very small architecture if all of the registers on each signal are mapped to a single shift register. However, this severely limits the capability of the system to distribute delay along long wires because all of the registers in the netlist are packed into dense register resources. The placement tool cannot break these registers out into separate locations because there are so few empty registers available in the device. Thus, while such an implementation is small, it may be very slow. Conversely, the netlist could be mapped to a very large architecture if only one register is initially mapped to each shift register. This gives the placement tool plenty of options to improve critical path delay, but also artificially increases the size of the required device. Of course, the best mixture of size and speed of the application probably lies between these two extremes, but finding this implementation is not obvious. For that matter, it is unclear which of these implementations an FPGA architect should use to compare this architecture to others.

Chapter 9: Conclusions and Future Research

This dissertation provided a detailed look at the potential advantages and disadvantages of heavily pipelining, C-slowing and retiming FPGA-based applications. Heavily registered circuits are important to the future of FPGAs because they can address one of the largest drawbacks that typically plagues today's reconfigurable devices: a relatively low operating frequency. However, these circuits also present special challenges to FPGA CAD tools and put unique demands on the architectures themselves. Finding solutions to these issues is critical because they can dramatically affect the achievable clock rate and area requirements of mapped netlists. Towards this goal, this dissertation focused on four primary problems: how to make timing-driven placement more effective, the implications of packing and retiming registers on placement, how registers can affect routing and how to efficiently incorporate more register resources into existing FPGA architectures.

Chapter 5 provided an in-depth look at timing-driven FPGA placement. The well-established and often-cited technique used by VPR was shown to have a significant shortcoming in the fundamental way that it tracks the timing information of a netlist during the annealing process. Simply put, performing static timing analysis once every thousand or hundred thousand moves is simply not enough to insure that the timing information remains relevant. It was demonstrated that this can lead to disappointing results, particularly for heavily registered netlists since they are inherently more sensitive to changes in the placement. Although forcing the annealer to simply perform static timing analysis more often can improve the results, this comes with some risks. Not only does this dramatically increase placement runtime, it can potentially cause the annealer to fail entirely.

Chapter 5 solved this problem by introducing a new incremental criticality update technique that allowed the annealer to efficiently estimate changes in net criticality between every single annealing move. This approach was paired with a new cost function that enabled the system to take advantage of more up-to-date timing information. For conventional combinational and lightly registered sequential netlists, this technique produced 0.888x faster post-routing critical path delay without affecting wire cost. For heavily registered benchmarks, it generated placements that were 0.581x faster with 0.951x better wire cost.

While this performance benefit is impressive, perhaps more importantly, the timing-driven placement approach suggested in Chapter 5 only requires a few small changes to the basic placement algorithm. Thus, it can likely easily be incorporated into existing placement tools and provide immediate benefits for many different applications across many different FPGA architectures.

Speaking more broadly, this new timing-driven placement technique is interesting because it shows that it is possible to make dramatic improvements to VLSI CAD tools, even in areas that are thought to be essentially mature, solved problems. Since changing the smallest detail can have a huge effect on the performance of an algorithm, hopefully this work will inspire future research to more closely examine classical FPGA CAD tools and techniques. For that matter, this approach also showed that merely estimating the timing of nets during placement is enough to improve post-routing critical path delay. This lends credence to the possibility for much faster CAD tools in the future. Rather than performing costly exact calculations, good estimates may be sufficient to maintain the quality of results and might even lead to significant improvements if applied carefully.

Chapter 6 began by investigating the difficulties in packing heavily registered applications. Packing a netlist that has a large number of registers was shown to be problematic because conventional algorithms, such as T-VPack, assume that a register will always want to be in the same BLE as its source LUT. This limits the options available to the placement tool to use registers to distribute delay along long wires. Furthermore, conventional packing tools simply have no idea what to do when a signal has multiple registers on it and they will likely combine unrelated portions of the circuit together, making the placement problem unnecessarily difficult.

To address these problems, Chapter 6 introduced a new hybrid CLB and flip-flop level placement approach that added the capability to efficiently re-assign registers to new CLBs during the placement process. When targeting a four 4-LUT, four flip-flop architecture, this technique improved critical path delay by 0.870x and wire cost by 0.865x for benchmarks that were pipelined/C-slowed and retimed to have a minimum of one register on the output of each LUT. This approach improved critical path delay by 0.588x and wire cost by 0.682x for benchmarks that were pipelined/C-slowed and retimed to have a minimum of three registers after each LUT.

Since packing is such an ingrained part of the traditional CAD toolflow, like the issue surrounding the accuracy of timing information in conventional placement algorithms discussed in Chapter 5, simply making the observation that packing can be inherently flawed is somewhat of a revelation. Also like the approach in Chapter 5, this technique is particularly valuable because it can easily be incorporated into existing toolflows. Heavily registered applications will likely cause similar packing problems on any FPGA that has multiple BLEs in each CLB. Looking into the future, this problem will probably get worse since the trend in commercial FPGAs is to build devices with larger and larger CLBs.

Although traditional packing was shown to work acceptably for lightly registered applications, and it is probably a necessary part of the toolflow since it significantly reduces the placement problem size, the hope is that this work will encourage FPGA CAD developers to examine their general approach more carefully. For example, rather than packing the entire netlist and forcing the placement tool to either accept the potential limitations or, as with the new technique discussed in this dissertation, discover for itself where the problems are, it may be better to only pack the LUTs and flip-flops in the netlists that have strong relationships. Although packing the entire netlist might be necessary to simply get an initial placement, the packer can forward information to the placer regarding which components have a known reasonable packing versus those that were combined arbitrarily. If this is done, the placement tool will have a much better idea of which registers should be moved independently and which are better to move as an entire CLB. Although this approach requires altering the existing toolflow more extensively, it may achieve even better results than the technique suggested here.

Chapter 6 also looked at how retiming can be incorporated into the netlist compilation process. Retiming can be difficult to apply because when it is performed before placement, the system does not have any information regarding the delay accumulated in the interconnect. On the other hand, since retiming restructures the netlist, applying it after placement can be disruptive and lead to problems with timing closure.

Chapter 6 borrowed concepts from multiple previous research efforts to develop a technique to more fully incorporate retiming into the placement process. This retiming approach gradually introduces new registers into the system and leverages the power of simulated annealing optimization to integrate them into an existing placement. Unfortunately, the results of the testing performed in this chapter seem to indicate that retiming is not essential for circuits mapped to more sophisticated architectures. In the presence of a good placement tool, retiming only improved critical path delay by a few percent on architectures with clustered CLBs and long interconnect wires. This result was largely confirmed by the work in [36].

Basically, sophisticated integrated placement and retiming techniques do not provide a large benefit on these architectures because retiming a netlist before placement is actually very effective. The need for retiming after placement is only partially a CAD problem. It is also a symptom of a larger architectural problem. Specifically, retiming is necessary when a lack of resources in an architecture makes the delay of potentially sensitive nets unpredictable. If the device does not have sufficient fast connectivity between different logic blocks in timing-sensitive regions of a netlist, the placement tool has no choice but to make some of the wires it knows to be timing critical long. This creates a mismatch in the system between nets that the placer could optimize versus those that it could not. However, more sophisticated modern FPGA architectures put quite a bit of effort into providing dense logic blocks and fast interconnect resources. This eases the pressure put on the placement tool and reduces the need for retiming.

However, this is not to say that retiming during or after placement, when more accurate timing information is available, is entirely irrelevant. The MCNC netlists available for testing in this dissertation are quite small by today’s standards and applications will only get larger in the future. Since larger applications naturally have a more complex structure, they could place higher demands on the target architecture. This may make retiming more important. Furthermore, this general trend also places the burden on FPGA architects. They must insure that the interconnect resources provided by their FPGAs stays ahead of the needs of application developers. While FPGA architects already consider the effect that interconnect resources have on routing congestion, the nature of the problem presented by retiming is somewhat different. Rather than routing channel capacity, the concern is the number of logic blocks that can be reached with a certain delay. Of course, there are physical limitations that prevent every logic block from having a fast connection to every other logic block in the device, so the problem of retiming may be unavoidable when devices and applications scale beyond a certain critical threshold.

Chapter 7 dealt with the difficulties of pipelined routing. As discussed in [32], this problem occurs on FPGA architectures that contain registers with very limited input and output connectivity. Assigning flip-flops in a netlist to register locations on these types of architectures during placement can also force these signals to use specific routing wires. This can seriously affect the routability of circuits that contain a large number of registers. Thus, registers must be found during the routing process on these architectures. Chapter 7 analyzed the only two known algorithms that address the pipelined routing problem and discussed why pipelined routers cannot use the existing timing-driven formulation suggested by PathFinder. Largely, the issue is that PathFinder forwards net criticality information from one routing iteration to the next. This subtly relies on the fact that the criticality of a net cannot drastically change between routing iterations. However, this assumption is not true for pipelined routing since the locations of registers in the system are not fixed by the placement. Much like the problems encountered in Chapter 5, forwarding criticality information from one routing iteration to the next can cause a pipelined router to favor degenerate solutions.

To solve this problem, Chapter 7 introduced assumed criticality searching. This technique performs multiple simultaneous waves of exploration that each assume that a net has a slightly different criticality. This approach removes the need for any a priori knowledge and discovers better possible routes under the prevailing conditions by allowing the system to more accurately balance delay and congestion. When combined with QuickRoute to form the Armada algorithm, compared to the congestion-only original QuickRoute technique, this approach improved critical path delay by approximately 0.6x without affecting the number of required routing tracks.

However, while this result is significant, particularly because it provides greater insight into a very new and relatively poorly explored CAD problem, the results found in Chapter 8 seem to indicate that timing-driven pipelined routing may not be necessary on future FPGAs. As mentioned earlier, the pipelined routing problem is caused by registers in an architecture that have limited input and output connectivity. However, the results in Chapter 8 suggest that these types of registers may not provide a compelling benefit for island-style FPGAs. Since commercial FPGAs generally follow a basic island-style structure, this may limit the applicability of pipelined routing algorithms.

That said, while commercial architectures may not require pipelined routing, they may benefit from more extensive use of assumed criticality searching. Modern architectures generally contain a wide range of interconnect resources, ranging from unit-length wires to wires that span the entire length of the device. These extremely diverse routing resources make it possible for the delay of a net to significantly change from one routing iteration to the next, even in the conventional routing problem, by simply moving a signal to a different type of wire resource. This variability could cause timing-driven routers to generate poor solutions on existing architectures. Since the assumed criticality technique evaluates the criticality of a net for each individual path largely independently, it can likely handle heterogeneous wires much more gracefully.

As alluded to earlier, Chapter 8 investigated different ways of increasing the amount of register resources in island-style FPGAs. This was shown to be a compelling question because while increasing the amount of pipelining and C-slowing performed on an application nearly linearly improved critical path delay, it also drastically increased the number of registers in the netlist. These circuits could have between 3-20x more registers than LUTs. Since conventional FPGAs only contain one register per LUT, these circuits require a huge number of additional BLEs. Although multiple previous research efforts have looked into solving this problem, the systems that they have suggested have largely been very specialized devices with limited commercial feasibility. Thus, Chapter 8 attempted to address the issue with a different basic philosophy: how can existing FPGAs be modified to benefit heavily registered applications while not disturbing the characteristics of the device for lightly registered applications?

The first part of Chapter 8 evaluated the benefits of adding registers with limited connectivity to the switchboxes in the interconnect. Although this possibility has been suggested in prior research as an efficient way of incorporating additional registers into an FPGA, using delay estimates from a 0.65nm device, it was shown that this could only reduce critical path delay by 0.914x over an architecture that restricted registers to the CLBs. Thus, due to the CAD implications this introduces for placement and routing, it is unlikely that it is worth incorporating registers into the interconnect.

A better alternative was explored in the second part of Chapter 8. This section looked at the possibilities of adding inexpensive register resources to the CLBs. Specifically, Chapter 8 investigated the impact of allowing unused LUTs to be turned into one or more shift registers and adding independent flip-flops. The potential usability of these resources was captured in a few equations. These equations were applied to roughly estimate the potential mapping efficiency of different architectures. This testing found that because allowing a 4-LUT to be used as a 1 to 16-bit shift register does not require adding any additional inputs or outputs to the system, this is likely the kind of modification that will provide the largest benefit to heavily registered applications, with the lowest impact to lightly registered netlists. Allowing all the LUTs in an FPGA to be used as SR-16s could reduce the number of required BLEs by up to 0.508x compared to architectures without any shift register capabilities. Allowing all the LUTs in an FPGA to be used as SR-16s could improve the mapping efficiency over Xilinx-style devices that allow half of the LUTs to be used as SR-16s by up to 0.783x.

Although the necessary schematics and layouts needed to determine the implications of adding I/O pins to each CLB were not available, this testing did show that more extensive CLB modifications could further improve mapping efficiency. That said, it also showed that netlists with a lower amount of pipelining and C-slowing preferred architectures with extra independent flip-flops, while netlists with a higher amount of pipelining and C-slowing preferred systems with shift registers that were split into smaller independent banks. Largely, this is because although shift registers provide more register locations, they can only be implemented in BLEs with unoccupied LUTs.

Taken as a whole, this dissertation provides a glimpse into the future of FPGAs. As FPGAs are expected to implement more complex applications at a higher clock rate, pipelining and C-slowing will become a necessary part of the application development process. This has been shown to have serious implications for packing, placement and routing tools, along with the efficiency of the underlying architecture. The large number of registers in heavily pipelined and C-slowed circuits changes many of the basic characteristics of the netlists and creates different kinds of CAD problems compared to purely combinational or lightly registered applications. Failing to recognize these intrinsic shifts can easily increase the critical path delay and area of an application by a factor of two. That said, based upon the analysis and experiments in this dissertation, many of these issues can be dealt with by making relatively minor changes to existing CAD tools and FPGA architectures.

Looking into the future, the need for registered applications must be more fully acknowledged by CAD tool developers. Along these lines, tools must be developed that can assist programmers in determining the bottlenecks in their applications. Currently, pipelining and C-slowing must be applied manually. After placement and routing, application developers must carefully inspect their circuits to determine which signals fail to meet timing specifications. At that point, they must edit their HDL code to insert registers, hopefully avoiding making mistakes that change the functionality of their circuit. This process is unnecessarily difficult and haphazard. Visualization tools could help developers better understand problematic areas of their circuits, and automatic pipelining and C-slowing could prevent unnecessary errors.

Simply highlighting the critical and near critical paths in the circuit and indicating which lines in the HDL code generated these portions of the netlist would provide extremely useful feedback. Since HDL code is sent through logic synthesis and technology mapping routines largely hidden from the user, it can often be difficult to determine the relationship between structures in a mapped FPGA implementation and the source code.

Furthermore, once the developer has decided to pipeline or C-slow a section of their circuit, registers could be added automatically. While it can be time consuming to add registers to HDL code manually, it is relatively straightforward for a CAD tool to pipeline or C-slow specific sections of a circuit at the LUT level. The HDL code can then be automatically updated to reflect these changes. Although extensive testing would be necessary to determine the real-world usability of such a tool, this might make developing high-speed circuits considerably faster and easier.

In addition, while registered applications clearly change the problems presented to the CAD tools and the FPGAs themselves, the netlists and architectures explored here were relatively simplistic. More detailed testing must be done using larger benchmarks mapped to more sophisticated FPGAs. For example, the largest circuits in this testing only require about 1/20 the logic provided by a medium to large Xilinx device. Since larger circuits are naturally more complex, they also present different problems to the CAD tools.

For that matter, the flagship FPGAs of both Xilinx and Altera contain much more sophisticated logic resources. While they include specialized resources such as fast carry-chains and dedicated multipliers, they have also migrated from 4-LUTs to 5 and 6-LUTs. These type of resources change the way that netlists are mapped to FPGAs and affect the realities of the physical layouts. For example, while fast carry chains provide low delay, direct connections between CLBs, these connections currently do not have access to registers. Thus, using these resources has repercussions on the pipelining or C-slowing capabilities of the circuit. Furthermore, implementing logic using larger LUTs changes the ratio of logic to registers in the device. While this certainly has an effect on how circuits are mapped to the system, this also changes the area implications for using LUTs as shift registers or adding I/O pins.

Future fabrication technologies also present some interesting issues for FPGA architectures. For example, 3-D semiconductor structures might make it much easier to provide fast interconnect wires between different logic blocks. While this can make FPGAs simply run faster, as discussed earlier, this also has ramifications on the effectiveness of retiming. For that matter, this can also dramatically increase the amount of available transistor area. This may make enhancements such as including additional registers in each CLB significantly less expensive. Looking even further into the future, silicon nano-wire and carbon nano-tube devices have fundamentally different fabrication implications. Although some research has been done into reliability and testing issues of FPGA-like structures built from these technologies, some of the possible manufacturing techniques also have interesting ways of building extremely small state-holding components. This can have an interesting effect on the cost of introducing more registers into the system.

Bibliography

[1] V. Betz and J. Rose. “FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density”, ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 1999: 37-46.

[2] V. Betz, J. Rose, and A. Marquardt, Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, 1999.

[3] V. Betz and J. Rose, “VPR: A New Packing, Placement and Routing Tool for FPGA Research”, International Conference on Field-Programmable Logic and Applications, 1997: 213-22.

[4] W. Chow and J. Rose. “EVE: A CAD Tool for Manual Placement and Pipelining Assistance of FPGA Circuits”, International Symposium on Field-Programmable Gate Arrays, 2002: 85-94.

[5] J. Cong and S. Lim, “Physical Planning with Retiming”, International Conference on Computer-Aided Design, 2000: 2-7.

[6] J. Cong and C. Wu, “FPGA Synthesis with Retiming and Pipelining for Clock Period Minimization of Sequential Circuits”, Design Automation Conference, 1997: 644 - 649.

[7] J. Cong and X. Yuan, “Multilevel Global Placement with Retiming”, Design Automation Conference, 2003: 208-13.

[8] T. Cormen, C. Leiserson, and R. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.

[9] C. Ebeling, D. Cronquist and P. Franklin. “RaPiD - Reconfigurable Pipelined Datapath”, International Workshop on Field-Programmable Logic and Applications, 1996: 126-35.

[10] K. Eguro and S. Hauck. “Armada: Timing-Driven Pipeline-Aware Routing for FPGAs”, ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2006: 169-78.

[11] K. Eguro and S. Hauck. “Enhancing Timing-Driven FPGA Placement for Pipelined Netlists”, Design Automation Conference, 2008: 34-7.

[12] K. Eguro and S. Hauck. “Simultaneous Retiming and Placement for Pipelined Netlists”, IEEE Symposium on Field-Programmable Custom Computing Machines, 2008.

[13] K. Eguro, S. Hauck, A. Sharma, "Architecture-Adaptive Range Limit Windowing for Simulated Annealing FPGA Placement", Design Automation Conference, 2005: 439-44.

[14] H. Eisenmann and F. Johannes, “Generic Global Placement and Floorplanning”, Design Automation Conference, 1998: 269-74.

[15] H. Gao, Y. Yang, X. Ma, and G. Dong, “Analysis of the Effect of LUT Size on FPGA Area and Delay Using Theoretical Derivations”, IEEE Symposium on Quality of Electronic Design, 2005: 370-4.

[16] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, “PipeRench: A Reconfigurable Architecture and Compiler”, IEEE Computer, 2000: 70 - 6.

[17] S. Kirkpatrick, C. Gelatt and M. Vecchi, “Optimization by Simulated Annealing”, Science, vol. 220, no. 4598, May 13, 1983: 671-80.

[18] J. Kleinhans, G. Sigl, F. Johannes and K. Antreich, “GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization”, IEEE Transactions on Computer-Aided Design, vol. 10, no. 3, March 1991: 356-65.

[19] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs”, ACM/SIGDA Symposium on Field Programmable Gate Arrays, 2006: 21-30.

[20] J. Lam and J. M. Delosme, "Performance of a New Annealing Schedule", Design Automation Conference, 1988: 306-11.

[21] C. Leiserson, F. Rose, and J. Saxe, "Optimizing synchronous circuitry by retiming", Caltech Conference on VLSI, 1983: 87-116.

[22] C. Leiserson and J. Saxe, “Retiming Synchronous Circuitry”, Algorithmica, Vol. 6, 1991: 5-35.

[23] S. Li and C. Ebeling. “QuickRoute: A Fast Routing Algorithm for Pipelined Architectures”, IEEE International Conference on Field-Programmable Technology, 2004: 73-80.

[24] J. Lillis, C. K. Cheng and T. T. Y. Lin, “Algorithms for Optimal Introduction of Redundant Logic for Timing and Area Optimization”, International Symposium on Circuits and Systems, 1996: 452-5.

[25] A. Marquardt, V. Betz and J. Rose, “Timing-Driven Placement for FPGAs”, ACM/SIGDA Symposium on Field Programmable Gate Arrays, 2000: 203-13.

[26] A. Marquardt, V. Betz, and J. Rose, “Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density”, ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 1999: 37-46.

[27] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, “A Reconfigurable Arithmetic Array for Multimedia Applications”, ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 1999: 135-43.

[28] L. McMurchie and C. Ebeling. “PathFinder: A negotiation-based performance-driven router for FPGAs”, ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 1995: 473-82.

[29] P. Pan, “Continuous Retiming: Algorithms and Applications”, International Conference on Computer Design, 1997: 116 - 21.

[30] K. Rajagopal, T. Shaked, Y. Parasuram, T. Cao, A Chowdhary and B, Halpin, “Timing Driven Force Directed Placement with Physical Net Constraints”, International Symposium on Physical Design, 2003: 60-6.

[31] R. Seidl, K. Eckl, and F. Johannes, “Performance-directed Retiming for FPGAs using Post-placement Delay Information”, Design, Automation and Test in Europe Conference, 2003: 770-5.

[32] A. Sharma, Development of a Place and Route Tool for the RaPiD Architecture, M.S. Thesis, University of Washington, Dept. of EE, 2001.

[33] A. Sharma, C. Ebeling and S. Hauck. "PipeRoute: A Pipelining-Aware Router for FPGAs", ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2003: 68-77.

[34] A. Sharma, C. Ebeling, S. Hauck, "PipeRoute: A Pipelining-Aware Router for FPGAs", University of Washington, Dept. of EE Technical Report UWEETR-2002-0018, 2002.

[35] D. Singh and S. Brown, “Incremental Placement for Layout-Driven Optimizations on FPGAs”, International Conference on Computer-Aided Design, 2002: 752-9.

[36] D. Singh and S. Brown, “Incremental Retiming for FPGA Physical Synthesis”, Design Automation Conference, 2005: 433-8.

[37] D. Singh and S. Brown, “Integrated Retiming and Placement for Field Programmable Gate Arrays”, ACM/SIGDA Symposium on Field Programmable Gate Arrays, 2002: 67-76.

[38] D. Singh and S. Brown, “The Case for Registered Routing Switches in Field Programmable Gate Arrays”, ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2001: 161-9.

[39] J. Swartz, V. Betz and J. Rose, “A Fast Routability-Driven Router for FPGAs”, ACM/SIGDA International Symposium on FPGAs. 1998, 140-9.

[40] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon. "HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array", ACM/SIGDA Symposium on Field Programmable Gate Arrays, 1999: 125-34.

[41] B. Von Herzen, “Signal Processing at 250MHz using High-Performance FPGA’s”, ACM International Symposium on FPGAs. 1997, 62-8.

[42] N. Weaver, J. Hauser, and J. Wawrzynek, “The SFRA: A Corner-Turn FPGA Architecture”, ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2004: 3-12.

[43] N. Weaver, Y. Markovskiy, T. Patel, and J. Wawrzynek, “Post-Placement C-slow Retiming for the Xilinx Virtex FPGA”, ACM/SIGDA Symposium on Field-Programmable Gate Arrays, 2003: 185-94.

[44] Wilton, Steven J. E. “Architecture and Algorithms For Field-Programmable Gate Arrays with Embedded Memory,” Ph.D. Thesis, University of Toronto, 1997.

[45] Xilinx Inc., “Virtex-II Platform FPGAs Complete Data Sheet” Version 3.5, 2007 downloaded from http://www.xilinx.com

[46] Xilinx Inc., XC4000XL 3.3V Field Programmable Gate Array Product Specification Version 2.1, 1998 downloaded from http://www.xilinx.com

Appendix A

Conventional MCNC Netlists

Combinational Circuits	Input Pins	Output Pins	LUTs	FFs	Required BLEs	Logical Depth	Pipeline Amt	C-slow Amt
e64	65	65	273	0	273	4	0	1
ex5p	8	63	1064	0	1064	7	0	1
apex4	9	19	1261	0	1261	6	0	1
misex3	14	14	1397	0	1397	7	0	1
alu4	14	8	1522	0	1522	7	0	1
des	256	245	1591	0	1591	6	0	1
seq	41	35	1750	0	1750	7	0	1
apex2	38	3	1878	0	1878	8	0	1
spla	16	46	3690	0	3690	8	0	1
pdc	16	40	4575	0	4575	9	0	1
ex1010	10	10	4598	0	4598	8	0	1
Sequential Circuits	**Input Pins**	**Output Pins**	**LUTs**	**FFs**	**Required BLEs**	**Logical Depth**	**Pipeline Amt**	**C-slow Amt**
s1423	18	5	220	74	220	14	0	1
tseng	52	122	1046	385	1046	8	0	1
dsip	229	197	1362	224	1362	3	0	1
diffeq	64	39	1494	377	1494	10	0	1
bigkey	229	197	1699	224	1699	3	0	1
s298	4	6	1930	8	1930	15	0	1
frisc	20	116	3539	886	3539	8	0	1
elliptic	131	114	3602	1122	3602	8	0	1
s38584.1	38	304	6156	1260	6156	9	0	1
s38417	29	106	5974	1463	5974	11	0	1
clma	62	82	8364	33	8364	16	0	1

Depth = 1 MCNC Netlists

Combinational Circuits	Input Pins	Output Pins	LUTs	FFs	Required BLEs	Logical Depth	Pipeline Amt	C-slow Amt
e64	65	64	273	409	409	1	3	1
ex5p	8	63	1064	1472	1472	1	6	1
apex4	9	18	1261	1348	1348	1	5	1
misex3	14	14	1397	1714	1714	1	6	1
alu4	14	8	1522	1867	1867	1	6	1
des	256	245	1591	2838	2838	1	5	1
seq	41	35	1750	2235	2235	1	6	1
apex2	38	3	1878	2413	2413	1	7	1
spla	16	46	3690	4596	4596	1	7	1
pdc	16	40	4575	5767	5767	1	8	1
ex1010	10	10	4598	5796	5796	1	7	1
Sequential Circuits	**Input Pins**	**Output Pins**	**LUTs**	**FFs**	**Required BLEs**	**Logical Depth**	**Pipeline Amt**	**C-slow Amt**
s1423	17	4	220	1486	1486	1	13	14
tseng	51	122	1046	4202	4202	1	0	8
dsip	228	189	1362	1544	1544	1	2	2
diffeq	63	39	1494	6304	6304	1	0	10
bigkey	228	190	1699	2114	2114	1	2	3
s298	3	6	1930	4555	4555	1	3	15
frisc	19	116	3539	13600	13600	1	7	8
elliptic	130	114	3602	12877	12877	1	0	8
s38584.1	31	189	6156	17928	17928	1	8	9
s38417	28	52	5974	23589	23589	1	4	11
clma	61	66	8364	18158	18158	1	4	16

Depth = 0.33 MCNC Netlists

Combinational Circuits	Input Pins	Output Pins	LUTs	FFs	Required BLEs	Logical Depth	Pipeline Amt	C-slow Amt	Post - Retiming C-slow
e64	65	64	273	1614	1614	0.33	4	1	3
ex5p	8	63	1064	4629	4629	0.33	7	1	3
apex4	9	18	1261	4125	4125	0.33	6	1	3
misex3	14	14	1397	5226	5226	0.33	7	1	3
alu4	14	8	1522	5667	5667	0.33	7	1	3
des	256	245	1591	10017	10017	0.33	6	1	3
seq	41	35	1750	6933	6933	0.33	7	1	3
apex2	38	3	1878	7362	7362	0.33	8	1	3
spla	16	46	3690	13974	13974	0.33	8	1	3
pdc	16	40	4575	17469	17469	0.33	9	1	3
ex1010	10	10	4598	17448	17448	0.33	8	1	3
Sequential Circuits	**Input Pins**	**Output Pins**	**LUTs**	**FFs**	**Required BLEs**	**Logical Depth**	**Pipeline Amt**	**C-slow Amt**	**Post - Retiming C-slow**
s1423	17	4	220	4521	4521	0.33	14	14	3
tseng	51	122	1046	12858	12858	0.33	1	8	3
dsip	228	189	1362	5913	5913	0.33	3	2	3
diffeq	63	39	1494	19107	19107	0.33	1	10	3
bigkey	228	190	1699	7596	7596	0.33	3	3	3
s298	3	6	1930	13683	13683	0.33	4	15	3
frisc	19	116	3539	41502	41502	0.33	8	8	3
elliptic	130	114	3602	39411	39411	0.33	1	8	3
s38584.1	31	189	6156	54021	54021	0.33	9	9	3
s38417	28	52	5974	70908	70908	0.33	5	11	3
clma	61	66	8364	54855	54855	0.33	5	16	3

RaPiD Benchmarks

Netlist	# of Required RaPiD Cells	Min # of Registers	Max Latency of Any Sink
firtm	16	20	16
fft16	12	40	3
sobel	18	49	5
matmult4	16	129	31
cascade	16	226	21
firsymeven	16	377	31
imagerapid	14	149	11
sort_g	11	159	32
sort_rb	11	159	31

CURRICULUM VITAE
Kenneth Eguro

Education

Ph.D., Electrical Engineering	**University of Washington – Seattle, WA**
1/2003 – 10/2008	*Supporting High-Performance Pipelined Computation in Commodity-Style FPGAs* Advisor – Prof. Scott Hauck
M.S., Electrical Engineering	**University of Washington – Seattle, WA**
6/2001 – 12/2002	*RaPiD AES: Developing an Encryption-Specific FPGA Architecture* Advisor – Prof. Scott Hauck
Graduate Work	**University of Illinois – Champaign, IL**
9/2000 – 5/2001	Coursework on computer architecture & parallel programming
B.S., Computer Engineering	**Northwestern University – Evanston, IL**
9/1996 – 6/2000	Concentration in VLSI and Computer Aided Design Honors Thesis – *synFPGA: Application Specific FPGA Synthesis*

Research Experience

Researcher	**Microsoft Research – Redmond, WA**
11/2008 –	Work focusing on the application of hardware-based accelerators
Research Assistant	**EE Dept., University of Washington – Seattle, WA**
6/2001 – 10/2008	Member of Adaptive Computing Machines and Emulators Lab, investigating FPGA architectures and CAD algorithms
Intern	**Microsoft Research, Hardware Device Group – Redmond, WA**
8/2005 – 11/2005 6/2004 – 9/2004	Development of applications and a graphical programming language to explore a prototype reconfigurable computing platform
Undergraduate Researcher	**ECE Dept., Northwestern University – Evanston, IL**
1/1999 – 6/2000	Research in fast placement and routing algorithms with Prof. Majid Sarrafzadeh.
9/1998 – 9/1999	Research into applications of reconfigurable logic in high-performance computing with Prof. Scott Hauck

Teaching Experience

Instructor	**EE Dept., University of Washington – Seattle, WA**
9/2007 – 12/2007	Course Instructor for EE471 – Computer Design and Organization
9/2006 – 12/2006	Course Instructor for EE471 – Computer Design and Organization
3/2006 – 6/2006	Course Instructor for EE471 – Computer Design and Organization
1/2006 – 3/2006	Course Instructor for EE541 – Automatic Layout for Integrated Circuits
Teaching Assistant	**EE Dept., University of Washington – Seattle, WA**
9/2004 – 12/2004	Teaching assistant and guest lecturer for EE540 – VLSI Testing
9/2000 – 6/2001	ECE Dept., University of Illinois – Champaign, IL Conducted weekly lectures for ECE290 - Introduction to Computer Engineering
Research Mentor	**EE Dept., University of Washington – Seattle, WA**
6/2001 – 12/2002	Managed 12 undergraduate students to assist with research in FPGA applications
Tutor	**Athletics Dept., Northwestern University – Evanston, IL**
9/1998 – 6/1999	Tutoring C/C++ and digital design multiple times per week

Honors

Academic & Research

2003 – 2004 Academic Year	Finalist, Microsoft Research Fellowship
2002 – 2003 Academic Year	Nominated, UW EE Dept. Yang Research Award
1999 – 2000 Academic Year	Graduated first in class, Computer Engineering curriculum Winner, Northwestern ECE Dept. IEC Everitt Award

Teaching	**University of Washington**
2007 – 2008 Academic Year	Winner, College of Engineering Teaching Assistant Innovator Award
	Nominated, EE Dept. Outstanding Teaching Assistant Award
2006 – 2007 Academic Year	Winner, EE Dept. Outstanding Teaching Assistant Award
	Nominated, College of Engineering Teaching Assistant Innovator Award
2005 – 2006 Academic Year	Nominated, EE Department Outstanding Teaching Award
	University of Illinois
2000 – 2001 Academic Year	Nominated, ECE Dept. Harold L. Olesen Teaching Assistant Award

Publications

Book Chapter and Patent

- K. Eguro and S. Hauck, “Fast Compilation Techniques” In S. Hauck and A. Dehon (Eds.) *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, Morgan Kaufmann/Elsevier, 2008.
- Provisional US Patent #4178-Inv-0001P.1USPRO, *Enhancing Timing-Driven Placement*, filed 12/10/2007.

Refereed Publications

- K. Eguro and S. Hauck “Simultaneous Retiming and Placement for Pipelined Netlists”, *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2008, 139-48.
- K. Eguro and S. Hauck, “Enhancing Timing-Driven FPGA Placement for Pipelined Netlists”, *Design Automation Conference*, 2008, 34-7.
- K. Eguro, “Supporting Heavily Pipelined Reconfigurable Computing on Commodity Devices”, *SIGDA Ph.D. Forum at Design Automation Conference*, 2006.
- K. Eguro and S. Hauck, "Armada: Timing-Driven Pipeline-Aware Routing for FPGAs", *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2006, 169-78.
- K. Eguro and S. Hauck, "Resource Allocation for Coarse Grain FPGA Development", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. Vol. 24, No. 10, Oct 2005, 1572-81.
- K. Eguro, S. Hauck and A. Sharma, "Architecture-Adaptive Range Limit Windowing for Simulated Annealing FPGA Placement", *Design Automation Conference*, 2005, 439-44.
- K. Eguro and S. Hauck, “Issues and Approaches to Coarse-Grain Reconfigurable Architecture Development”, *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003, 111-20.
- M. Wang, X. Yang, K. Eguro, and M. Sarrafzadeh, "Multi-Center Congestion Minimization during Placement", *ACM International Symposium on Physical Design*, 2000, 147-52.
- X. Yang, M. Wang, K. Eguro, and M. Sarrafzadeh, "A Snap-On Placement Tool", *ACM International Symposium on Physical Design, 2000*, 153-58.

Technical Reports

- S. Hauck, K. Compton, K. Eguro, M. Holland, S. Phillips, A. Sharma, "Totem: Domain-Specific Reconfigurable Logic", *University of Washington, Dept. of EE Technical Report*, 2006.
- K. Eguro and S. Hauck, "Issues of Wirelength Cost Models in Routing-Constrained FPGAs", *University of Washington, Dept. of EE Technical Report* UWEETR-2004-0006, 2004.
- K. Eguro and S. Hauck, “Decipher: Architecture Development of Reconfigurable Encryption Hardware”, *University of Washington, Dept. of EE, Technical Report*, 2002.
- K. Eguro and S. Hauck, “synFPGA: Application Specific FPGA Synthesis”, *Northwestern University, Dept. of ECE Technical Report*, 2000.

Invited Presentations

- “Incremental Timing Analysis for FPGA Placement”, Simon Fraser University, 8/8/2008.
- “Reconfigurable Computing: Architectural and Design Tool Challenges”, Microsoft Corporation, 5/8/2008.
- “Simultaneous Retiming and Placement”, Achronix Corporation, 4/16/2008.
- “Timing Concerns of Pipeline-Aware Placement and Routing”, Dept. of Energy Tech. Review, 12/11/2007.

- "Pipeline and Retiming-Aware Placement", Cascadia Workshop on FPGAs, 8/10/2007.
- "Pipelining Commodity Reconfigurable Devices", University of British Columbia, 9/22/2006.
- "Timing-Driven Pipeline-Aware Routing", Actel Corporation, 7/28/2006.

Research Interests

- The exploration of innovative, high-performance computing architectures
- Application of reconfigurable computing platforms
- Design automation and fast CAD algorithms
- Encryption and cryptanalysis

Professional Activities

Reviewer

4/2008 – present	IEEE Transactions on Computers
12/2005 – present	IEEE Transactions on Circuits and Systems I
12/2005 – present	EURASIP Journal on Embedded Systems

Professional Societies

3/2003 – present	Student Member of IEEE
9/1999 – 6/2000	Treasurer, Eta Kappa Nu Honor Society – Beta Tau Chapter