Place and Route Techniques for FPGA Architecture Advancement

Akshay Sharma

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

Doctor of Philosophy

University of Washington

2005

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

Akshay Sharma

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 Supervisory Committee:

_____

Scott Hauck

Reading Committee:

_____

Scott Hauck

_____

Carl Ebeling

_____

Larry McMurchie

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, 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

Place and Route Techniques for FPGA Architecture Advancement

Akshay Sharma

Chair of the Supervisory Committee:
Associate Professor Scott Hauck
Electrical Engineering

Efficient placement and routing algorithms play an important role in FPGA architecture research. Together, the place-and-route algorithms are responsible for producing a physical implementation of an application circuit on the FPGA hardware. The quality of the place-and-route algorithms has a direct bearing on the usefulness of the target FPGA architecture. The benefits of including powerful new features on an FPGA might be lost due to the inability of the place-and-route algorithms to fully exploit these features. Thus, the advancement of FPGA architectures relies heavily on the development of efficient place-and-route algorithms.

The subject of this dissertation is the development of place and route techniques that could play an important role in FPGA architecture advancement. The work presented in this dissertation is divided into two topics:

Architecture-Adaptive FPGA Placement - The first topic deals with the development of a universal placement algorithm (Independence) that adapts to the target FPGA architecture. We have successfully demonstrated Independence's adaptability to three different architectures. Our results also show that Independence is able to adapt to a class of routing-poor FPGA architectures.

Pipelined Routing - The second topic focuses on the development of a routing algorithm (PipeRoute) that can be used to route application circuits on high-speed, pipelined FPGA architectures. In our experiments, PipeRoute was able to successfully route netlists on a coarse-grained pipelined architecture. The algorithm incurred a 20% overhead when compared to a realistic lower bound. We also used PipeRoute in an exploratory flow to find an architecture that was up to 19% better than a hand-architected pipelined architecture.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Number Page

LIST OF TABLES

Table Number Page

ACKNOWLEDGEMENTS

A number of people and organizations played important roles during my tenure in graduate school. The National Science Foundation (NSF) and Altera Inc provided financial support while I was conducting the research presented in this dissertation. The NSF's and Altera's generous support played a crucial role in making this dissertation a reality.

I would like to thank the RaPiD group (notably Chris Fisher), Katherine Compton, and Shawn Phillips for providing the infrastructure for my work on the development of pipelined FPGA routing algorithms. My research on architecture-adaptive FPGA placement relied on software tools provided by Vaughn Betz and Andre DeHon. In addition to providing software tools, Andre DeHon patiently answered a number of technical questions on getting the tools up and running.

Graduate school can be an intellectually lonely experience. Thanks are due to Ken Eguro, Chandra Mulpuri, and Mark Holland for being very effective sounding boards. I appreciate every moment spent standing at a white board and discussing research ideas with them, and am grateful for the time they took out of their busy schedules to help me out. I would also like to thank Larry McMurchie for sharing his valuable insight and experience with me. Larry's comments and observations went a long way in improving the quality of my research work.

The role played by my research advisors cannot be overemphasized. Together, Scott Hauck and Carl Ebeling were the most effective advisors any graduate student could possibly hope for. They single-handedly transformed me from a wide-eyed, novice graduate student (who knew very little) to a seasoned researcher who can confidently propose and defend research ideas. Scott's and Carl's constant belief in me, coupled with their patience and understanding, made my experience in graduate school truly fulfilling.

Finally, I don't think enough can ever be said in appreciation of my family's support. My stint in graduate school was largely a selfish endeavor and yet, Mary Ann and my parents always stood by me and tried to make every day worthwhile. Their love and unwavering support kept me stable and focused, and without them I would have probably pressed the eject button on my doctoral studies a long time ago.

DEDICATION

To the millions of underprivileged women, children and men the world over. I hope this work benefits you some day.

Chapter 1: Introduction

A Field Programmable Gate Array (FPGA) is a pre-fabricated silicon device that can be reconfigured to implement different applications. The reconfigurability of an FPGA is derived from reprogrammable Static Random Access Memory (SRAM) cells (Figure 1-1). By programming the SRAM cells, the functionality of FPGA logic units can be tailored to implement a particular computation. Interconnections between logic units are established by programming SRAM cells to connect prefabricated routing wires together. Thus, any given application can be mapped to an FPGA by programming the functionality and connectivity of logic units based on the specific characteristics of the application.

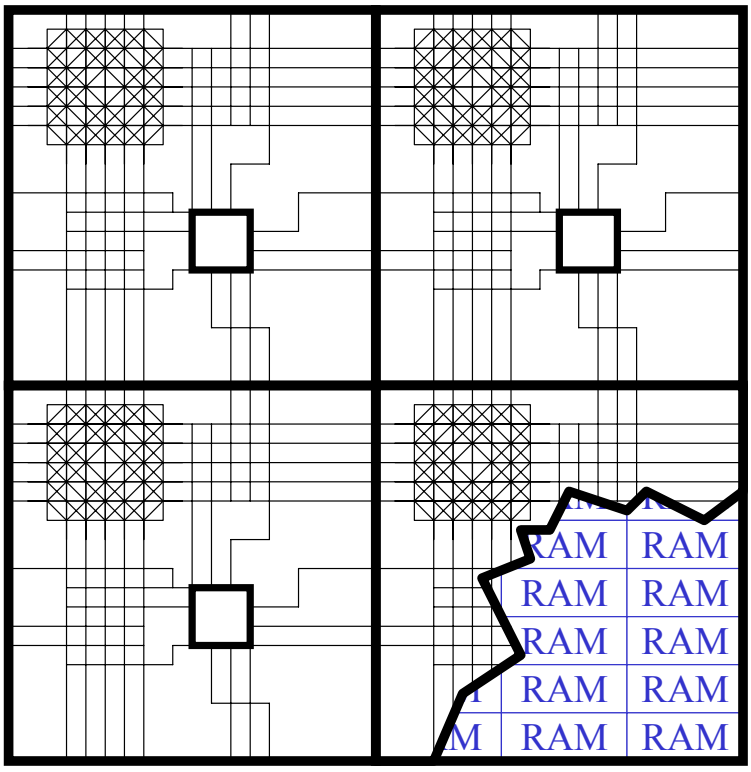

Figure 1-1: A conceptual illustration of an SRAM programmable FPGA [21]. Logic blocks are shown as bold black boxes, and routing wires are shown as black intersecting horizontal and vertical lines.

The reconfigurability of FPGAs is fundamentally different from that of traditional general-purpose microprocessors. An application is implemented on a microprocessor by compiling the application to a stream of hardware instructions that are sequentially decoded and executed by fixed, general-purpose logic resources. Unlike FPGAs, the functionality of a microprocessor's logic resources cannot be modified on a per-application basis. Instead, each application is compiled to a unique stream of instructions that are executed on the microprocessor. Since it is possible to express almost any application as a sequence of instructions, microprocessors are arguably the most flexible computational devices today. However, microprocessors often incur a performance penalty due to the very flexibility that is their claim to fame. To support flexibility, the fixed logic resources in a microprocessor are deliberately designed to efficiently execute certain basic computations. Consequently, applications that would benefit from customized, tailor-made logic resources often take a performance hit when executed on general-purpose microprocessors.

While microprocessors are attractive for their flexibility, an Application Specific Integrated Circuit (ASIC) is a computational device that is customized to a specific application. Since the exact nature of the application is known beforehand, the hardware resources in an ASIC are designed to provide the highest performance implementation of the application. The price paid by ASICs because of their superlative performance characteristics is flexibility. Once an ASIC has been fabricated, it is generally not possible to modify the ASIC to implement any other application other than the one it was intended for. Further, since the Non-Recurring Engineering (NRE) costs involved in designing and fabricating an ASIC are comparatively high, it is generally infeasible to design and fabricate ASICs in low volumes.

Since their introduction in the mid eighties, FPGAs have evolved from a simple, low-capacity gate array technology to devices [2,3,59,60] that provide a mix of coarse-grained datapath units, microprocessor cores, on-chip A/D conversion, and gate counts in the millions. Today, FPGAs are firmly ensconced in the space of computational devices that was originally dominated by microprocessors and ASICs. Much like microprocessors, FPGA-based systems can be reprogrammed on a per-application basis. At the same time, FPGAs offer significant performance benefits over microprocessor implementations for a number of applications. Although these performance benefits are still generally an order of magnitude less than equivalent ASIC implementations, the low NRE costs, fast time-to-market, and flexibility of FPGAs make them an attractive choice for low-to-medium volume applications.

Realizing that FPGA performance levels lag ASICs, FPGA architectures have been intensely researched over the past two decades. A major aspect of FPGA architecture research is the development of Computer Aided Design (CAD) tools for mapping applications to FPGAs. It is well established that the quality of an FPGA-based implementation is largely determined by the effectiveness of the accompanying suite of CAD tools. The benefits of an otherwise well designed, feature rich FPGA architecture might be diminished if the CAD tools are not able to take advantage of the features that the FPGA provides. Thus, CAD algorithm research is crucial to the architectural advancement that is necessary to narrow the performance gaps between FPGAs and other computational devices like ASICs.

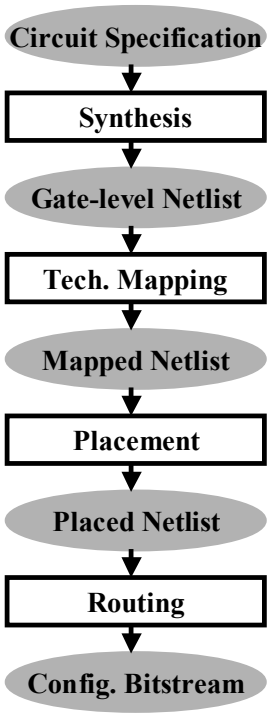

Figure 1-2: A typical FPGA CAD tool flow.

A typical FPGA CAD tool-flow is shown in Figure 1-2. Initially, a circuit specification of the application is produced either by means of schematic capture, or a high-level description in a Hardware Description Language (HDL). The appropriate circuit specification serves as the input to a Synthesis tool. The Synthesis tool synthesizes the circuit specification into a circuit (we will use the terms circuit and netlist interchangeably from this point on) that consists of basic logic gates and their interconnections (hereafter called 'nets'). In the Technology Mapping phase, the gate-level netlist is

transformed into a functionally equivalent netlist that is expressed in terms of the logic units that are provided by the FPGA device. The mapped netlist serves as an input to the Placement tool[1], which determines the actual physical location of each netlist logic block in the FPGA layout. After the physical location of each logic block has been determined, the Routing tool[2] determines the FPGA routing resources that are needed to route the nets that connect the placed logic blocks. At the end of a successful routing phase, a stream of configuration bits is produced. The configuration bitstream is used to program SRAM cells in the FPGA fabric so that the target application can be implemented.

The subject of this dissertation is the development of placement and routing (together termed 'place-and-route') algorithms for the advancement of FPGA architectures. The research presented in this work is divided into two topics. The first deals with the development of a universal placement algorithm that adapts to the interconnect structure of the target FPGA device. The second topic focuses on the development of a routing algorithm for pipelined FPGAs.

[1] The phrases "placement tool", "placement algorithm" and "placer" will be used interchangeably from this point on.
[2] The phrases "routing tool", "routing algorithm" and "router" will be used interchangeably from this point on.

Chapter 2: FPGA Architectures

In this chapter, we describe different FPGA architectural paradigms. The goal is to provide the reader with a high-level background of both commercial and academic FPGA technologies. We use individual case studies to describe features of selected architectures that have clearly different logic and/or interconnect structures.

2.1 Island-Style FPGAs

Currently, island-style architectures (Figure 2-1) represent the dominant FPGA architectural style. State-of-the-art devices [3,59] offered by commercial vendors like Xilinx and Altera have island-style architectures.

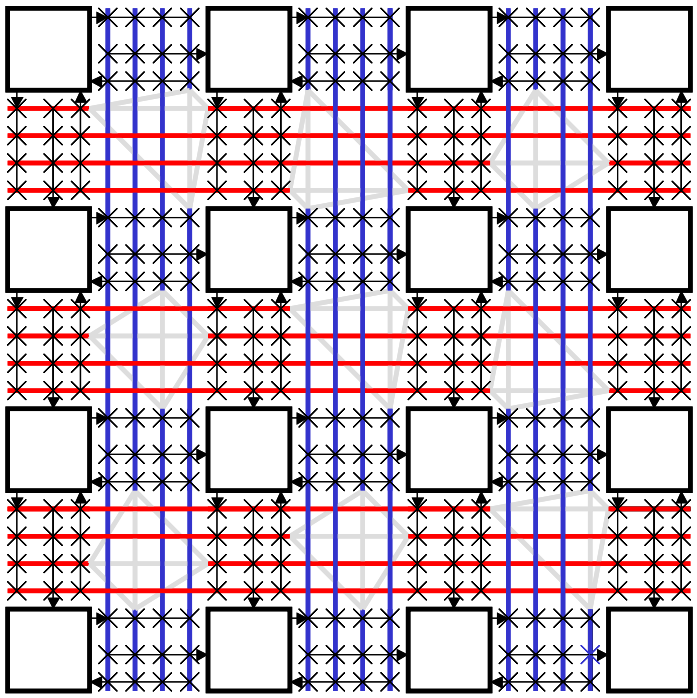

Figure 2-1: An illustration of an island-style FPGA. The white boxes represent logic blocks. The horizontal (red) and vertical (blue) intersecting lines represent routing wires. The logic blocks connect to surrounding wires using programmable connection-points (shown as crosses), and individual wires connect to each other by means of programmable routing switches (shown as gray lines).

The logic structure of an island-style FPGA consists of configurable logic blocks (CLBs). In the simplest case, a CLB may be composed from a single programmable four-input lookup table (4-LUT) coupled with a D flip-flop (D-FF). The 4-LUT is used to implement any four-input combinational function, and the D-FF provides the option of registering the output of the 4-LUT. The CLBs in commercial devices generally consist of multiple 4-LUT / D-FF pairs that are clustered together. Intra-cluster communication between the 4-LUT / D-FF pairs is accomplished using a localized high-speed interconnect scheme.

Island-style FPGAs are characterized by a routing-rich interconnect structure in which each CLB is an 'island' in a sea of routing resources. CLBs are interconnected using prefabricated segmented routing wires that run in the horizontal and vertical direction. A connection between a routing wire and a CLB terminal is established using a programmable connection point. A routing wire can be connected to another routing wire using a programmable switch. Routing switches can be either buffered or unbuffered, while connection points are generally buffered. In Figure 2-1, routing wires are represented as intersecting red and blue lines, connection points are represented as crosses, and switches are shown in gray.

Altera's Stratix II [3] architecture is a commercial example of an island-style FPGA (Figure 2-2). The logic structure consists of LABs (which is Altera's term for CLBs), memory blocks, and digital signal processing (DSP) blocks. LABs are used to implement general-purpose logic, and are symmetrically dispersed in rows and columns throughout the device fabric. The DSP blocks are custom designed to implement full-precision multipliers of different granularities, and are grouped into columns. Input- and output-only elements (IOEs) represent the external interface of the device. IOEs are located along the periphery of the device.

Each Stratix II LAB consists of eight Adaptive Logic Modules (ALMs). An ALM consists of two adaptive LUTs (ALUTs) that have eight inputs altogether. The construction of an ALM allows the implementation of two separate four-input Boolean functions. Further, an ALM can also be used to implement any six-input Boolean function, and some seven-input functions. In addition to lookup tables, an ALM provides two programmable registers, two dedicated full-adders, a carry chain, and a register-chain. The full-adders and carry chain can be used to implement arithmetic operations, and the register-chain is used to build shift registers. The outputs of an ALM drive all types of interconnect provided by the Stratix II device. Figure 2-3 illustrates a LAB's interconnect interface.

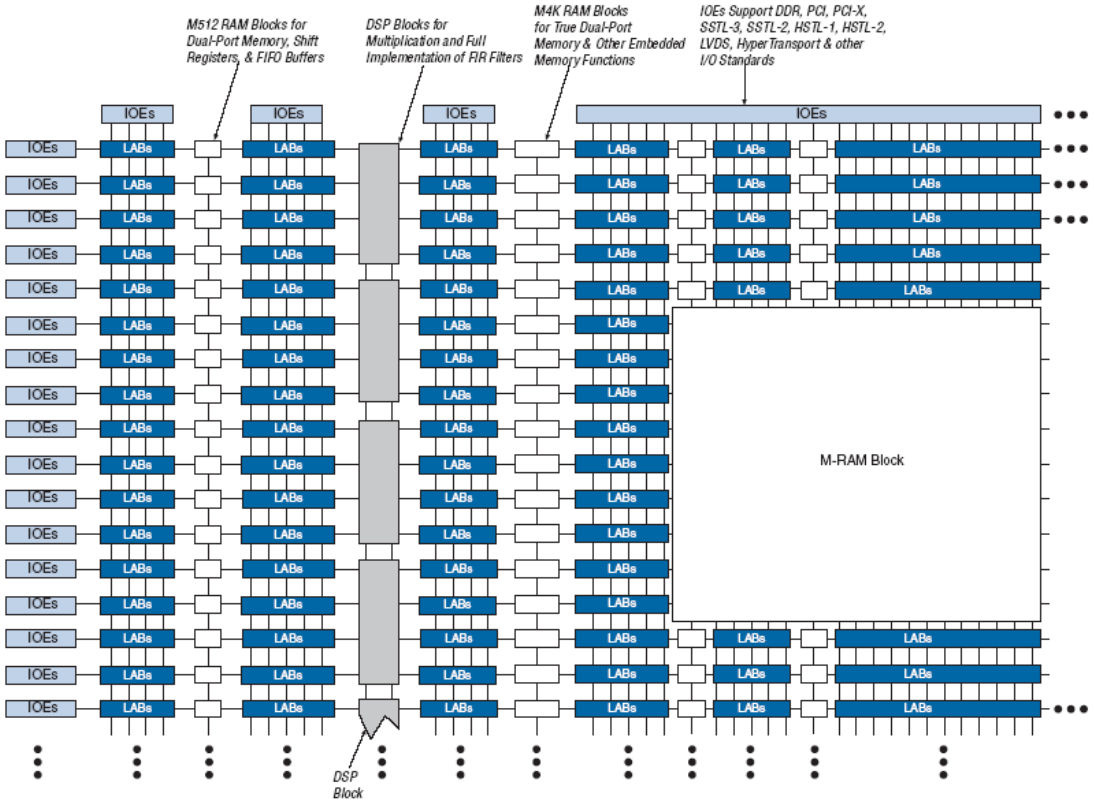

Figure 2-2: [3] A functional overview of Altera's Stratix-II device. The LABs represent clustered lookup table-based logic blocks. The optimized DSP blocks are provided to enhance the performance of signal processing applications. The IOEs are used to connect the fabric to external devices.

Interconnections between LABs, RAM blocks, DSP blocks and the IOEs are established using the MultiTrack interconnect structure. The interconnect structure consists of wire segments of different lengths and speeds. The interconnect wire-segments span fixed distances, and run in the horizontal (row interconnects) and vertical (column interconnects) directions. The row interconnects (Figure 2-4) can be used to route signals between LABs, DSP blocks, and memory blocks in the same row. Row interconnect resources are of the following types:

- Direct connections between LABs and adjacent blocks
- R4 resources that span four blocks to the left or right.
- R24 resources that provide high-speed access across the width of the device.

Each LAB owns its set of R4 interconnects. A LAB has approximately equal numbers of driven-left and driven-right R4 interconnects. An R4 interconnect that is driven to the left can be driven by either the primary LAB (Figure 2-4) or the adjacent LAB to the left.

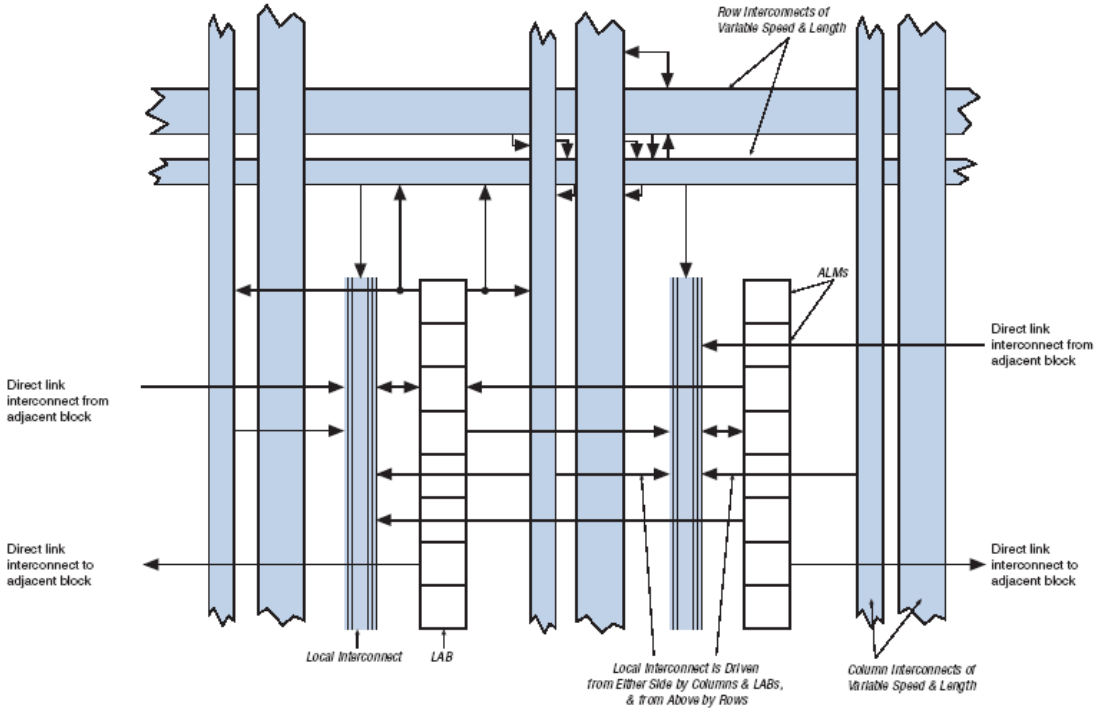

Figure 2-3: [3] The structure of a Stratix II LAB. Each LAB consists of eight ALMs. The local interconnect structure is used to provide intra-LAB communication, and local connectivity amongst adjacent LABs. The local interconnect structure can also be driven by the row and column routing resources in the general interconnect structure.

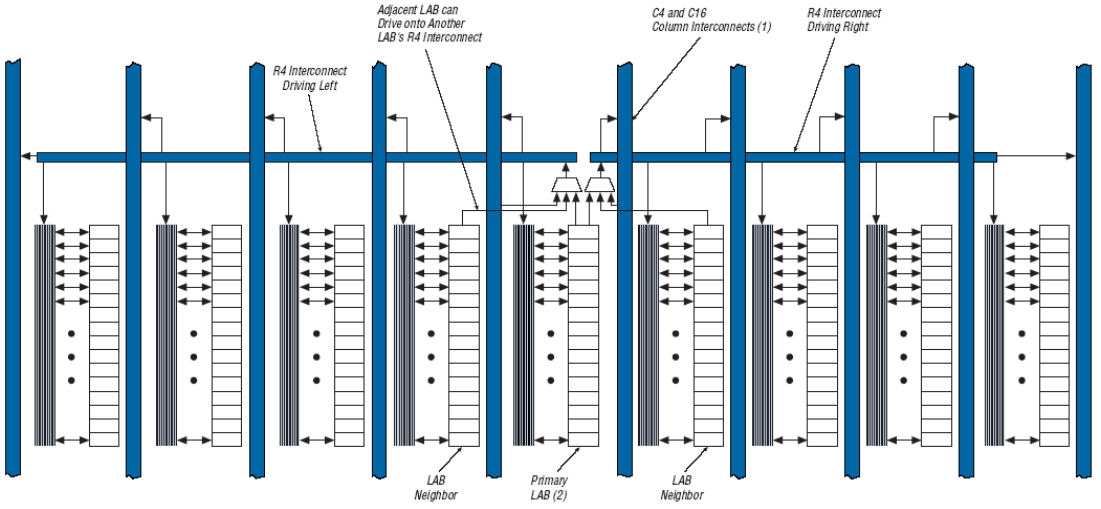

Figure 2-4: [3] R4 interconnect structure. (1) C4 and C16 column interconnects can drive R4 interconnects. (2) This pattern is repeated for every LAB in the row. (3) All 16 possible outputs of a LAB are shown.

Similarly, a driven-right R4 interconnect may be driven by the primary LAB or the LAB immediately to its right. Multiple R4 resources can be connected to each other to establish longer connections within the same row. R4 interconnects can also drive C4 and C16 column interconnects, and the R24 high-speed row resources.

The column interconnect structure is similar to the row interconnect structure. Column interconnects include:

- Carry chain interconnects within a LAB, and from LAB to LAB in the same column.
- Register chain interconnects.
- C4 resources that span four blocks in the up and down directions
- C16 resources for high-speed vertical routing across the height of the device.

The carry chain and register chain interconnects are separate from the local interconnect (Figure 2-3) in a LAB. Each LAB has its own set of driven-up and driven-down C4 interconnects. The C4 interconnects can also be driven by the LABs that are immediately adjacent to the primary LAB. Multiple C4 resources can be connected to each other to form longer connections within a column, and C4 interconnects can also drive row interconnects to establish column-to-column interconnections. C16 interconnects are high-speed vertical resources that span sixteen LABs. A C16 interconnect can drive row and column interconnects at every fourth LAB. A LAB's local interconnect structure cannot be directly driven by a C16 interconnect; only C4 and R4 interconnects can drive a LAB's local interconnect structure. Figure 2-5 shows the C4 interconnect structure in the Stratix II device.

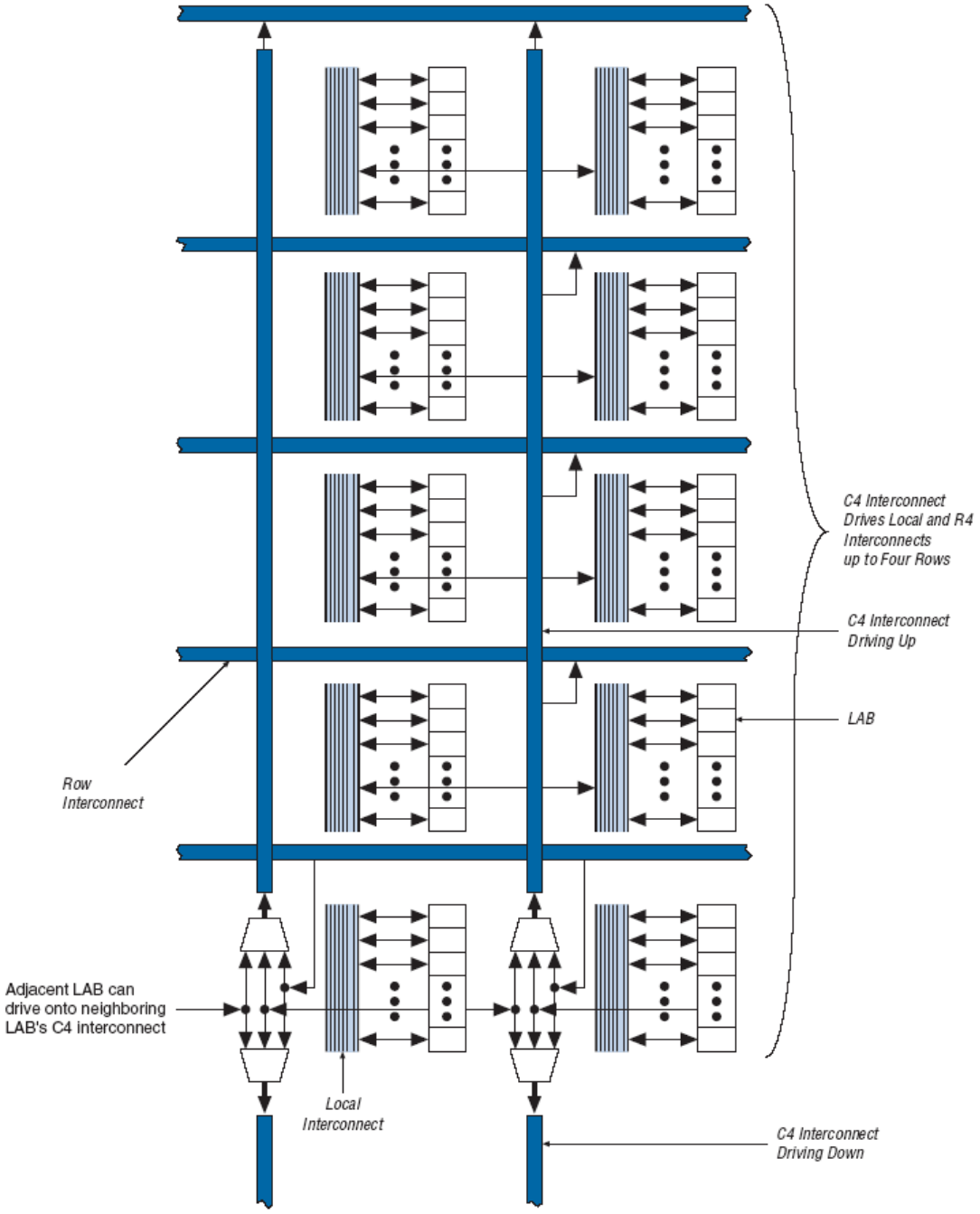

Figure 2-5: [3] C4 interconnect structure. Each C4 interconnect can drive either up or down four LAB rows.

2.2 Non Island-Style FPGAs

Island-style architectures are probably the most popular FPGA architectural style today. However, island-style FPGAs have certain shortcomings. Some FPGA architecture researchers believe that the interconnect structure of an FPGA is best utilized when the logic utilization of the device is relatively low [8,16]. This high-interconnect / low-logic utilization approach is in direct contrast to the high logic utilization approach that has been adopted for island style FPGAs. Other research-groups believe that the fine-to-medium grained structure of island-style FPGAs is not suited to streaming, compute-intensive applications [15,19]. All in all, while island-style FPGAs continue to be developed for commercial applications, FPGA architecture research is also conducted in parallel in the hope of improving performance. This section briefly surveys examples of non island-style FPGA architectures.

2.2.1 Hierarchical FPGAs

The interconnect structure of an island-style FPGA is generally designed to maximize logic utilization. Hierarchical FPGAs belong to the class of routing-poor FPGA architectures that are designed to increase interconnect utilization at the expense of logic utilization. The philosophy behind routing-poor architectures is increased silicon utilization through efficient use of the interconnect structure (which may account for ~ 80 – 90 % of the total area in island-style FPGAs).

A well-known academic hierarchical FPGA is the Hierarchical Synchronous Reconfigurable Array (HSRA) [16]. HSRA has a strictly hierarchical, tree-based interconnect structure (Figure 2-6). As a result, HSRA's logic and interconnect structures are not as closely coupled as the logic and interconnect structures of island-style FPGAs. Recall that each LAB in Altera's Stratix II device 'owns' R4 and C4 interconnects. In HSRA, the only wire-segments that directly connect to the logic units are at the leaves of the interconnect tree. All other wire-segments are decoupled from the logic structure.

An HSRA logic unit consists of a single 4-LUT / D-FF pair. The input-pin connectivity is based on a *c-choose-k* strategy [16], and the output pins are fully connected. The richness of HSRA's interconnect structure is defined by its *base channel width* and interconnect *growth rate*. The base channel width '*c*' is the number of tracks at the leaves of the interconnect tree (in Figure 2-6, c=3). The growth rate '*p*' is the rate at which the interconnect grows towards the root (in Figure 2-6, p=0.5). The growth rate is realized using the following types of switch-blocks:

- *Non-compressing* (2:1) switch blocks - The number of root-going tracks is equal to the sum of the number of root-going tracks of the two children.

- *Compressing* (1:1) switch blocks – The number of root-going tracks is equal to the number of root-going tracks of either child.

A repeating combination of non-compressing and compressing switch blocks can be used to realize any value of p less than one. For example, a repeating pattern of (2:1 → 1:1) switch blocks realizes p=0.5, while the pattern (2:1 → 2:1 → 1:1) realizes p=0.67. An HSRA that has only 2:1 switch blocks provides maximum interconnection bandwidth (i.e. a value of p=1).

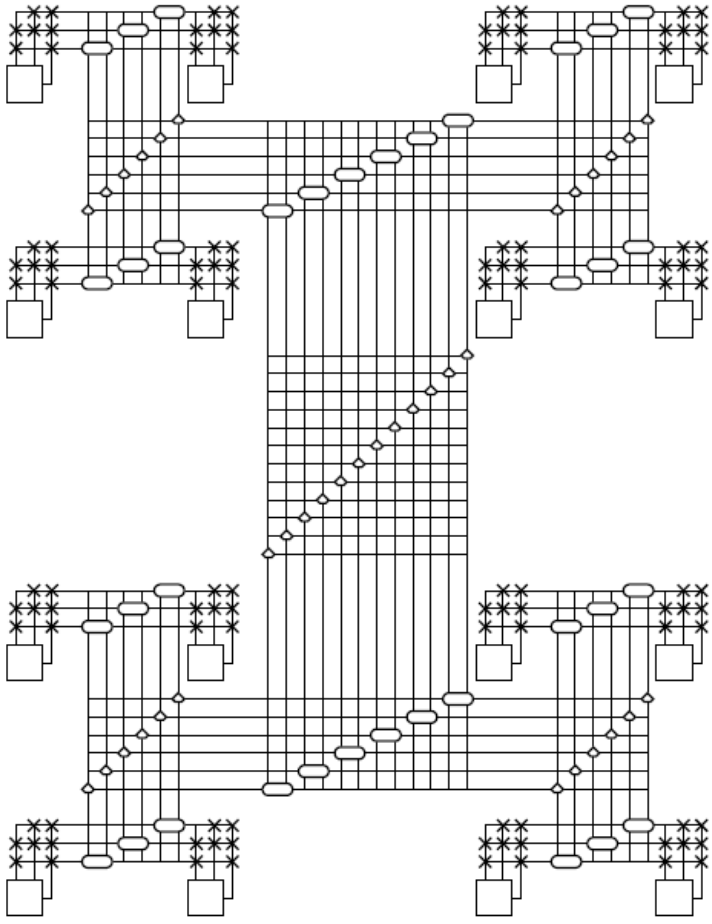

Figure 2-6: [16] An illustration of HSRA's interconnect structure. The leaves of the interconnect tree represent logic blocks, the crosses represent connection points, the hexagon-shaped boxes represent non-compressing switches, and the diamond-shaped boxes represent compressing switches. The base channel width of this architecture is three (c=3), and the interconnect growth rate is 0.5 (p=0.5).

2.2.2 RaPiD

The RaPiD [15] architecture is targeted to high-throughput, compute-intensive applications like those found in DSP. RaPiD is a domain-specific architecture that is designed to provide high-performance implementations of applications that fall within the DSP domain. The logic and interconnect structures are customized to the domain, and RaPiD does not support general-purpose combinational logic like commercial island-style devices.

RaPiD (Figure 2-7) has a coarse-grained, 1-dimensional structure (Figure 2-7). The logic structure contains 16-bit registers, ALUs, multipliers and small SRAM blocks. RaPiD's interconnect structure consists entirely of segmented 16-bit buses. There are two types of buses; *short* buses provide local communication between logic units, while *long* buses can be used to establish longer connections using bidirectional switches called *bus-connectors* (shown as the small square boxes in Figure 2-7).

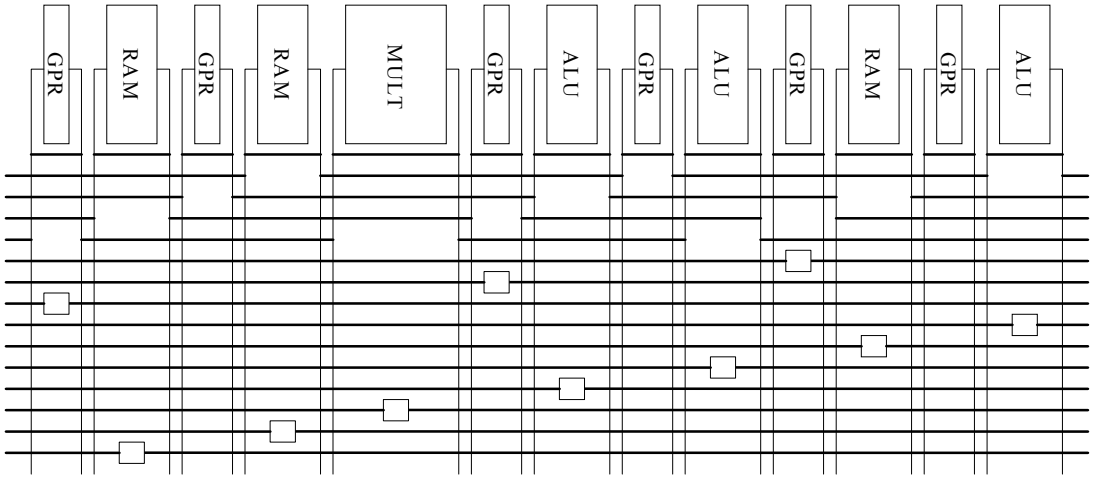

Figure 2-7: An example of a RaPiD architecture cell. Several RaPiD cells can be tiled together to create a representative architecture.

Since DSP applications are generally pipelined, RaPiD's logic and interconnect structures include an abundance of registers. Register banks are provided at the outputs of logic units and in the interconnect bus-connectors. Each register bank can be used to pick up between 0 – 3 pipelining registers.

2.2.3 FPGA Fabrics for Sytems-on-a-Chip (SoC)

Programmable logic cores are an important component of current SoC devices. Generally, programmable logic cores are implemented using vendor-supplied IP blocks that closely resemble the fabric of the vendor's island-style FPGA devices. However, the tools used to integrate the IP blocks

with the rest of the SoC toolflow (which usually resembles an ASIC toolfow) are still relatively immature. A solution presented in [22] uses the concept of *soft* programmable logic cores. The architecture of the soft core is represented in terms of a Hardware Description Language (HDL) like Verilog. This allows a customizable programmable core to be synthesized using an ASIC toolflow, thus eliminating the problems associated with integrating IP-specific tools with the SoC toolflow.

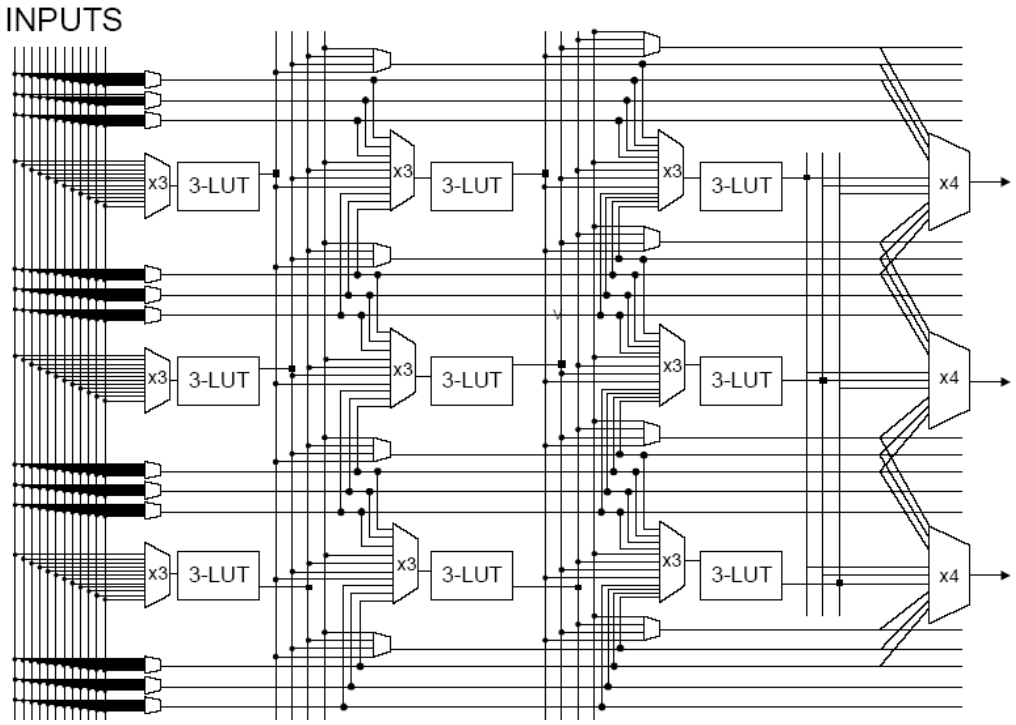

Figure 2-8: [22] Gradual, directional architecture. The interconnect structure is directional, and gradually increases from left to right. The primary inputs of the architecture are on the left, and the primary outputs are on the right.

An example of a soft programmable core is shown in Figure 2-8. The logic structure is purely combinational, and is similar to the logic structure of LUT-based island-style FPGAs. The interconnect structure is clearly different from an island-style interconnect structure. Conventional synthesis tools are unable to handle combinational loops, and thus the interconnect structure in Figure 2-8 is directional. Since programmable logic cores are intended for relatively simple computations, the highly flexible switchblock-based interconnect structure of an island-style FPGA is replaced with a relatively simple structure that consists entirely of programmable routing multiplexors. Finally, the richness of the

interconnect structure increases from left to right to allow internal 3-LUTs to drive the primary outputs of the fabric.

Our main goal in this chapter was to provide a snapshot of current commercial and academic FPGA architectures. In the next chapter, we separately describe the FPGA placement and routing problems, and discuss popular algorithms used to solve these problems.

Chapter 3: FPGA Placement and Routing

The most important architectural feature of an FPGA is arguably the interconnect structure. Since any FPGA has a finite number of discrete routing resources, a large share of architectural research effort is devoted to determining the composition of an FPGA's interconnect structure. During architecture development, the effectiveness of an FPGA's interconnect structure is evaluated using placement and routing tools (collectively termed place-and-route tool). The place-and-route tool is responsible for producing a physical implementation of an application netlist on the FPGA's prefabricated hardware. Specifically, the placer determines the actual physical location of each netlist logic block in the FPGA layout, and the router assigns the signals that connect the placed logic blocks to routing resources in the FPGA's interconnect structure. Due to the finite nature of an FPGA's interconnect structure, the success of the router is heavily reliant on the quality of the solutions produced by the placer. Not surprisingly, the primary objective of the placer is to produce a placement that can indeed be routed by the router.

In this chapter, we separately discuss the FPGA placement and routing problems. Further, we also describe the most popular algorithm that is used to solve each problem. The algorithms that we describe in this chapter form the basis of our work on architecture adaptive FPGA placement and pipelined routing.

3.1 FPGA Routing

The FPGA routing problem is to assign nets to routing resources such that no routing resource is shared by more than one net. Pathfinder [33] is the current, state-of-the-art FPGA routing algorithm. Pathfinder operates on a directed graph abstraction ($G(V,E)$) of the routing resources in an FPGA. The set of vertices V in the graph represents the IO terminals of logic units and the routing wires in the interconnect structure. An edge between two vertices represents a potential connection between the two vertices. Given this graph abstraction, the routing problem for a given net is to find a directed tree embedded in G that connects the source terminal of the net to each of its sink terminals. Since the number of routing resources in an FPGA is limited, the goal of finding unique, non-intersecting trees (hereafter called "routes") for all the nets in a netlist is a difficult problem.

Pathfinder uses an iterative, negotiation-based approach to successfully route all the nets in a netlist. During the first routing iteration, nets are freely routed without paying attention to resource sharing.

Individual nets are routed using Dijkstra's shortest path algorithm [14]. At the end of the first iteration, resources are congested because multiple nets have used them. During subsequent iterations, the cost of using a resource is increased based on the number of nets that share the resource, and the history of congestion on that resource. Thus, nets are made to negotiate for routing resources. If a resource is highly congested, nets that can use lower congestion alternatives are forced to do so. On the other hand, if the alternatives are more congested than the resource, then a net may still use that resource. The cost of using a routing resource '*n*' during a routing iteration is given by Equation 3.1.

Equation 3.1: $c_n = (b_n + h_n) * p_n$

b_n is the base cost of using the resource n, h_n is related to the history of congestion during previous iterations, and p_n is proportional to the number of nets sharing the resource in the current iteration. The p_n term represents the cost of using a shared resource n, and the h_n term represents the cost of using a resource that has been shared during earlier routing iterations. The latter term is based on the intuition that a historically congested node should appear expensive, even if it is currently lightly shared.

An important measure of the quality of the routing produced by an FPGA routing algorithm is critical path delay. The critical path delay of a routed netlist is the maximum delay of any combinational path in the netlist. The maximum frequency at which a netlist can be clocked has an inverse relationship with critical path delay. Thus, larger critical path delays slow down the operation of netlist. Delay information is incorporated into Pathfinder by redefining the cost of using a resource n (Equation 3.2).

Equation 3.2: $Cn = A_{ij} * d_n + (1 - A_{ij}) * c_n$

The c_n term is from Equation 3.1, d_n is the delay incurred in using the resource, and A_{ij} is the criticality given by Equation 3.3.

Equation 3.3: $A_{ij} = D_{ij} / D_{max}$

D_{ij} is the maximum delay of any combinational path that goes through the source and sink terminals of the net being routed, and D_{max} is the critical path delay of the netlist. Equation 3.2 is formulated as a sum of two cost terms. The first term in the equation represents the delay cost of using resource n, while

the second term represents the congestion cost. When a net is routed, the value of A_{ij} determines whether the delay or the congestion cost of a resource dominates. If a net is near critical (i.e. its A_{ij} is close to 1), then congestion is largely ignored and the cost of using a resource is primarily determined by the delay term. If the criticality of a net is low, the congestion term in Equation 3.2 dominates, and the route found for the net avoids congestion while potentially incurring delay.

Pathfinder has proved to be one of the most powerful FPGA routing algorithms to date. Pathfinder's negotiation-based framework that trades off delay for congestion is an extremely effective technique for routing signals on FPGAs. More importantly, Pathfinder is a truly architecture-adaptive routing algorithm. The algorithm operates on a directed graph abstraction of an FPGA's routing structure, and can thus be used to route netlists on any FPGA that can be represented as a directed routing graph.

3.2 FPGA Placement

The FPGA placement problem is to determine the physical assignment of the logic blocks in a netlist to locations in the FPGA layout. The primary goal of any FPGA placement approach is to produce a placement that can be successfully routed using the limited routing resources provided by the FPGA. Versatile Place and Route [5,6] (VPR) is the current, public-domain state-of-the-art FPGA placement tool. VPR consistently produces high-quality placements, and at the time of this writing, the best reported placements for the Toronto20 [7] benchmark netlists are those produced by VPR.

VPR uses a simulated annealing algorithm [26] that attempts to minimize an objective cost function. The algorithm operates by taking a random initial placement of the logic blocks in a netlist, and repeatedly moving the location of a randomly selected logic block. The move is accepted if it improves the overall cost of the placement. In order to avoid getting trapped in local minima, non-improving moves are also sometimes accepted. The temperature of the annealing algorithm governs the probability of accepting a "bad" move at that point. The temperature is initially high, causing a large number of bad moves to be accepted, and is gradually decreased until no bad moves are accepted. A large number of moves are attempted at each temperature. VPR provides a cooling schedule that is used to determine the number of moves attempted at each temperature, the maximum separation between logic blocks that can be moved at a given temperature, and the rate of temperature decay.

VPR's objective cost function is a function of the total wirelength of the current placement. The wirelength is an estimate of the routing resources needed to completely route all nets in the netlist. Reductions in wirelength mean fewer routing wires and switches are required to route nets. This is an

important consideration because the number of routing resources in an FPGA is limited. Fewer routing wires and switches typically also translate to reductions in the delay incurred in routing nets between logic blocks. The total wirelength of a placement is estimated using a semi-perimeter metric, and is given by Equation 3.4. N is the total number of nets in the netlist, $bb_x(i)$ is the horizontal span of net i, $bb_y(i)$ is its vertical span, and $q(i)$ is a correction factor. Figure 3-1 illustrates the calculation of the horizontal and vertical spans of a hypothetical net that has ten terminals.

Equation 3.4: $$WireCost = \sum_{i=1}^{N} q(i) * (bb_x(i) + bb_y(i))$$

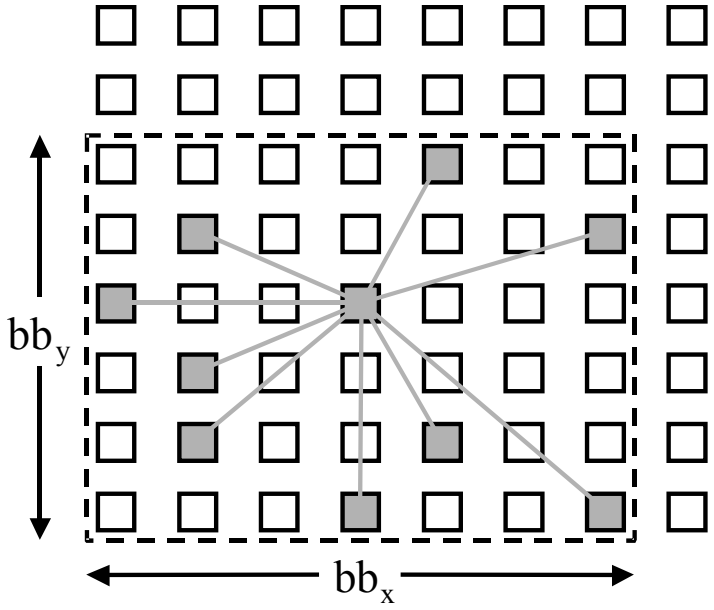

Figure 3-1: The horizontal and vertical spans of a hypothetical 10-terminal net [6]. The semi-perimeter of the net is $bb_x + bb_y$.

The cost function in Equation 3.4 does not explicitly consider timing information. Wirelength is a weak estimate of the delay of a net, especially when the net is routed on FPGAs that have a mix of segmented routing resources. In [30], VPR's placement algorithm is enhanced to include both wirelength and timing information. The enhanced algorithm (called TVPlace) starts out with a preprocessing step that creates a delay lookup table for the FPGA. This lookup table holds the delay of a minimum-delay route for every source-sink terminal pair in the FPGA's interconnect structure. During the placement process,

the lookup table is used to quickly estimate the delay of a net given a placement of its terminals. The timing cost of a placement is calculated using the cost functions in Equation 3.5 and Equation 3.6.

Equation 3.5: $$TimingCost(i, j) = Delay(i, j) * Criticality(i, j)^{CriticalityExponent}$$

Equation 3.6: $$TimingCost = \sum_{\forall i, j \subset circuit} TimingCost(i, j)$$

In Equation 3.5, *TimingCost(i,j)* represents the timing cost of a net that connects a source-sink pair *(i,j)*, *Delay(i,j)* is the delay of the net, and *Criticality(i,j)* is the criticality of the net. During the placement process, the delay of a net is obtained from the lookup table, while the criticality of a net is calculated using a static timing analysis. The *CriticalityExponent* is a parameter that can be tuned to control the relative importance of the criticality of a net. The formulation of the timing cost in Equation 3.5 encourages the placement algorithm to seek solutions that reduce *Delay(i,j)* for critical nets.

TVPlace's cost function is determined by both wirelength and timing cost, and is given by Equation 3.7.

Equation 3.7:

$$\Delta C = \lambda * (\Delta TimingCost \ / \ prevTimingCost) + (1 - \lambda) * (\Delta WireCost \ / \ prevWireCost)$$

Equation 3.7 calculates the change in cost of a placement using an auto-normalizing cost function that depends on changes in *WireCost* and *TimingCost*. The parameter λ is used to vary the relative importance of changes in *TimingCost* and *WireCost* during the placement process. The two normalization variables *prevWireCost* and *prevTimingCost* are updated at the beginning of a temperature iteration as per Equation 3.4 and Equation 3.6. The main benefit of using normalization variables is that changes in the cost of the placement do not depend on the actual magnitude of *TimingCost* and *WireCost*. This makes the cost function adaptive, since the size of a netlist or the target architecture does not skew cost calculations. Further, since *prevTimingCost* and *prevWireCost* are recalculated every temperature iteration, inaccuracies due to mismatched rates of change of the two cost components are minimized.

Chapter 4: Architecture Adaptive FPGA Placement – Motivation and Related Work

Currently, the modus operandi used in the development of placement algorithms is to use architecture-specific metrics to heuristically estimate the routability of a placement. For example, the routability of a placement on island-style FPGAs is estimated using the ever-popular semi-perimeter metric (Equation 3.4), while the routability of a placement on tree-based architectures [16] is estimated using cutsize-based metrics. Architecture-specific routability estimates limit the adaptability of a placement algorithm. To the best of our knowledge, there is no single placement approach that can adapt effectively to the interconnect structure of every FPGA in the architecture spectrum. This often proves to be an impediment in the early stages of FPGA architecture development, when the targeted placement algorithm is not well defined due to a lack of architectural information. We feel that research in FPGA architectures would stand to benefit from a universal placement algorithm that can quickly be retargeted to relatively diverse FPGA architectures.

4.1 VPR Targets Island-Style FPGAs

In Section 3.2 we described VPR, the state-of-the-art academic FPGA placement tool. Due to a strong prevalence of routing rich island-style FPGA architectures, VPR's placement algorithm is primarily targeted to island-style FPGAs (Figure 4-1). The semi-perimeter based cost function relies on certain defining features of island-style FPGAs:

Two-dimensional Geometric Layout – An island-style FPGA is laid out as a regular two-dimensional grid of logic units surrounded by a sea of routing wires and switches. As a result, VPR's cost function is based on the assumption that the routability of a net is proportional to the Manhattan distance (measured by semi-perimeter) between its terminals. A net with terminals that are far apart needs more routing resources than a net with terminals close to each other.

Uniform Connectivity – Island-style architectures provide uniform connectivity. The number and type of routing resources available for a net with a given semi-perimeter are independent of the actual placement of the terminals of the net. Thus, VPR determines the cost of a net based purely on its semi-perimeter, and not the actual location of the terminals of the net.

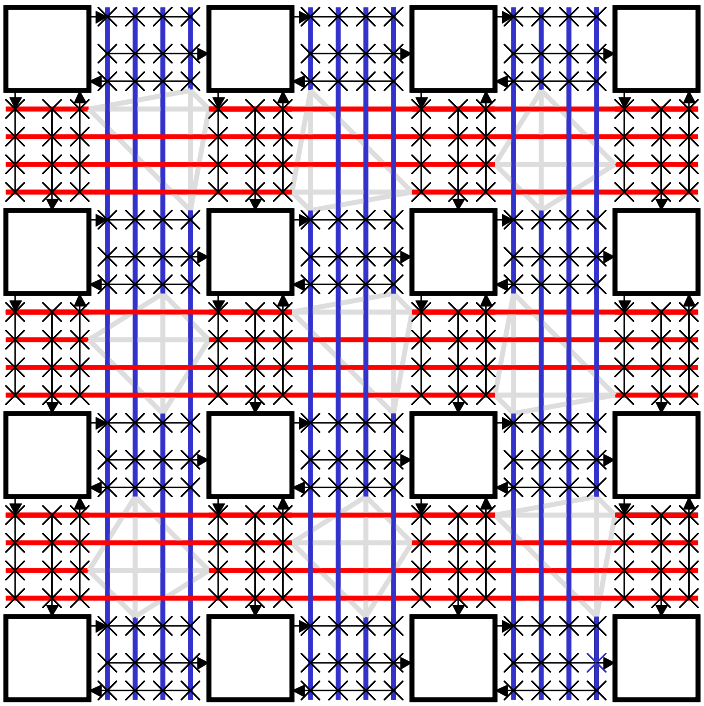

Figure 4-1: An illustration of an island-style FPGA. The white boxes represent logic blocks. The horizontal (red) and vertical (blue) intersecting lines represent routing wires. The logic blocks connect to surrounding wires using programmable connection-points (shown as crosses), and individual wires connect to each other by means of programmable routing switches (shown as gray lines).

VPR's dependence on island style FPGA architectures limits its adaptability to architectures that do not provide features of island-style FPGAs. For instance, the interconnect structure of an FPGA architecture may not conform to the Manhattan distance estimate of routability. One example is the hierarchical interconnect structure found in tree-based FPGA architectures [16] (Figure 4-2 (a)). In tree-based FPGAs, there is no way of estimating the number of routing resources between two logic units based on layout positions. In fact, for an architecture like HSRA [16], the number of routing resources required to connect a logic unit in one half of the interconnect tree to a logic unit in the other half does not depend on the actual locations of the logic units. A strictly semi-perimeter based cost function does not accurately capture the routability characteristics of tree-based FPGAs.

Another class of non-island style FPGA architectures provide heterogeneous interconnect structures. Triptych [8] (Figure 4-2 (b)) is an example of an FPGA architecture that provides only segmented vertical tracks. There are no segmented horizontal tracks; horizontal routes are built using directional, nearest-neighbor connections. A second example of an FPGA architecture that has non-uniform routing

resources can be found in [22] (Figure 4-2 (c)). The horizontal channels in this architecture gradually increase in width from left to right. For a given semi-perimeter, the amount of routing available to a net at the far right edge of this architecture exceeds the amount available at the far left edge. For both Triptych and the architecture presented in [22], the types and number of routing resources available to route a net clearly depends on the placement of the net's terminals. VPR's semi-perimeter based cost function is oblivious of the heterogeneity of such architectures.

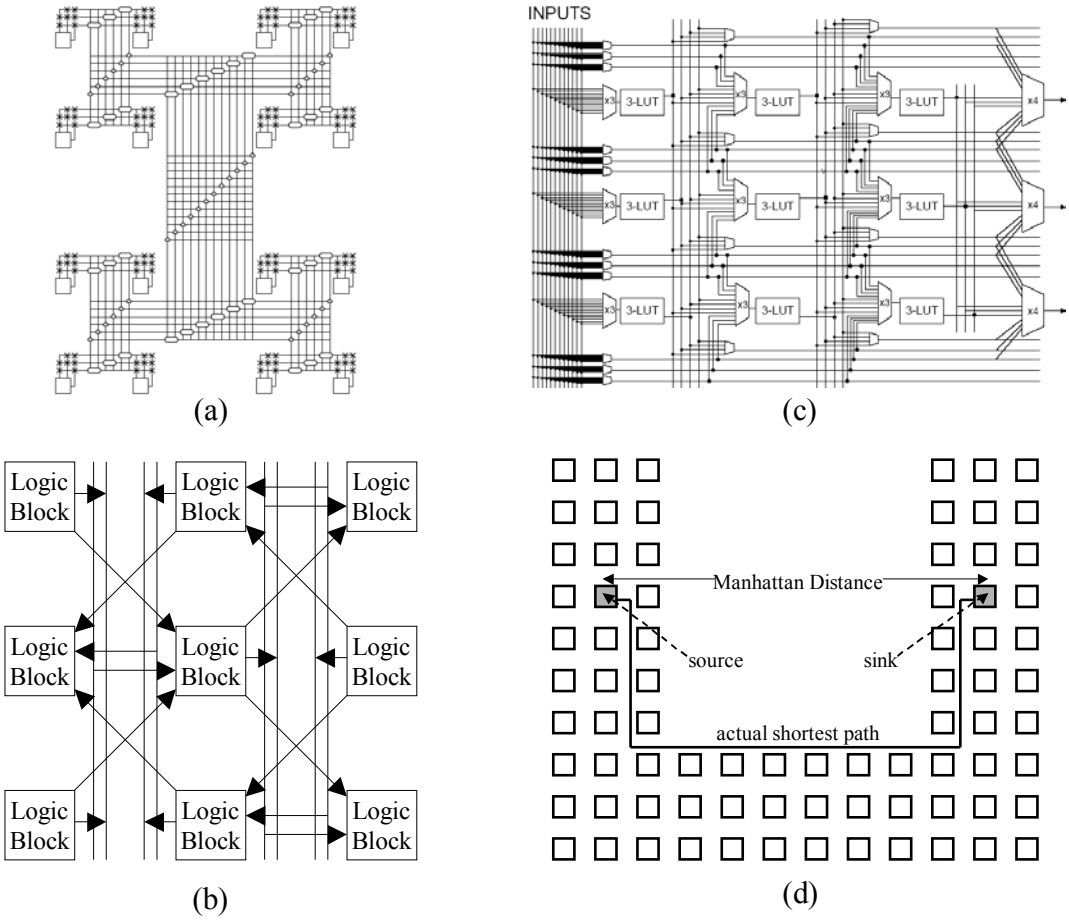

Figure 4-2: Non island-style FPGA architectures. (a) HSRA [16], (b) Triptych [8], (c) directional architecture from [22], and (d) a U-shaped FPGA core [58].

Finally, efforts to incorporate FPGA-like logic in System-on-Chip designs have motivated non-rectangular FPGA fabrics. In [22], the authors investigate a directional FPGA fabric that resembles the shape of a trapezoid. The FPGA fabrics proposed in [50] are built by abutting smaller, rectangular

fabrics of different aspect ratios (Figure 4-2 (d)). In both cases, the semi-perimeter metric is an inaccurate estimate of the resources available to route signals.

The primary feature that distinguishes the non-island style FPGAs discussed so far is the nature of the interconnect structure. The composition, flexibility, and heterogeneity of the routing resources directly influence the placement process. For every FPGA that has a unique interconnect structure, a placement cost function is formulated in terms of architecture specific parameters that accurately capture the cost of a placement. The architectural examples cited in this section clearly show that a semi-perimeter placement cost function does not adapt well to non-island style FPGAs. A cost function's adaptability lies in its ability to guide a placement algorithm to a high-quality solution across a range of architecturally diverse FPGAs.

The first major topic of this dissertation is the development of Independence, an architecture adaptive FPGA placement algorithm. The core of the Independence algorithm is tightly integrated placement and routing. The algorithm's adaptability is a direct result of using the Pathfinder algorithm to calculate the cost of a placement. Specifically, we use Pathfinder in the inner loop of a simulated annealing placement algorithm to maintain a fully routed solution at all times. Thus, instead of using architecture-specific routability estimates, we use the routing produced by an architecture adaptive router to guide the algorithm to a routable placement.

4.2 Previous Work in Integrated Placement and Routing

Since Independence is rooted in a simultaneous[3] place-and-route approach, we briefly survey existing research in integrated FPGA placement and routing in this section. Research in integrated place and route for FPGAs can be broadly categorized into three categories.

4.2.1 Partitioning-based Techniques

Partitioning-based FPGA placement is used to obtain a global routing of the netlist as a direct result of the partitioning process. Note that partitioning-based FPGA placement algorithms are not truly simultaneous place-and-route algorithms, since no explicit routing step is attempted during placement. However, since partitioning-based placement naturally produces the global routing of a netlist, we briefly survey partitioning-based techniques in the hope of identifying an approach that might aid in the development of Independence. Further, partitioning-based placement is a well-known divide-and-conquer approach to solving placement problems.

[3] We use the words 'integrated' and 'simultaneous' interchangeably from this point on.

Iterative k-way partitioning techniques are particularly well suited to tree-based FPGA architectures, and have been used to place and globally route netlists on HSRA [16] and k-HFPGA [55]. During recursive k-way partitioning, logic blocks are recursively clustered together into k smaller subtrees while reducing cutsize and/or area. At the end of the partitioning phase, the leaves of the netlist's partitioning tree are mapped to logic unit clusters in the tree-based architecture. Since there is a unique global route between any two logic unit clusters in a tree-based architecture, the global routing for the entire netlist is easily found from the placement.

Partitioning-based techniques have also been considered for simultaneously placing and routing netlists on island-style FPGA architectures. In [51], a recursive bipartitioning technique is used to place and globally route netlists on an island-style FPGA architecture. At the end of a bi-partitioning stage, if a net crosses the cutline, a pseudo-block is generated on the cutline to preserve a connection. Each psuedo-block corresponds to a track, and a sequence of pseudo-blocks between the terminals of a net corresponds to a global route for that net. When the bipartitioning is complete, each partition consists of a single switch-block with pseudo-blocks allocated at the partition edges. The global routing for the netlist is directly implied by the placed netlist.

A similar approach to integrated place-and-route for island-style FPGAs is presented in [1]. The FPGA is divided into m x n rectangular regions, and a partitioning heuristic is used to assign the logic blocks in a netlist to the regions. The assignment is improved using simulated annealing. A greedy congestion reduction heuristic is then used to select a Rectilinear Steiner Arborescence for each net such that cutsize is reduced. Finally, the nets that cross each edge of a region are assigned to switch-blocks located on the edge. This process is recursively carried out until each region consists of a single logic block.

Partitioning-based placement techniques can be used to simultaneously place and globally route netlists on FPGA architectures. However, since FPGAs have a finite number of discrete routing resources, heuristic estimates of the global routing requirements of a netlist during the placement process might not be the most accurate measure of the actual routing requirements of the netlist. A tighter coupling between partitioning-based placement and the interconnect structure of the FPGA might be obtained by finding detailed routes for signals during partitioning. However, the actual placement of a netlist is only known at the end of the partitioning-phase, and hence a complete detailed routing is not possible during the partitioning process.

4.2.2 Cluster-growth Placement

Cluster-growth placement is a technique that has been used to simultaneously place and route netlists on different FPGA architectures. In cluster-growth placement, signals are considered one at a time in a sequential manner. The terminals of the signal under consideration are placed based on a cost function derived from heuristic force-directed estimates [4], or global routing estimates [10]. Once a signal's terminals have been placed, it is not possible to change their placement to accommodate the demands of later signals.

Combining cluster-growth placement with detailed routing may seem like a good choice for architecture-adaptive placement. However, the quality of the placements produced by a cluster-growth approach is sensitive to the order in which signals are considered. Since determining an optimal ordering of the signals is a difficult task, cluster-growth placement is usually an iterative process. The signal ordering at the beginning of each pass is either random, or determined heuristically from netlist or architectural features.

4.2.3 Simulated Annealing Placement

Simulated Annealing [26] is one of the most powerful placement techniques for both FPGA [6] and ASIC [38] technologies. As mentioned in Section 3.2, VPR's placement algorithm uses simulated annealing to minimize wirelength and timing estimates. The quality of the resultant placements has proven to be consistently superior.

Simulated Annealing based simultaneous place-and-route techniques are presented in [35]. Fast global and detailed routing heuristics are used in the simulated annealing inner loop to estimate the routability and critical path delay of a placement. Separate techniques for row-based and island-style FPGAs are presented. A brief description of the techniques follows:

Row-based FPGAs (PRACT) [35]: The PRACT algorithm is targeted to row based FPGAs. The cost of a placement is a weighted, linear function of the number of globally unrouted nets, the number of nets that lack a complete detailed routing, and the critical path delay of the placement. For every move that is attempted during the annealing process, the nets that connect the moved logic blocks are ripped up and added to a queue of unrouted nets. After a move is made, fast heuristics attempt to find global and detailed routes for the ripped up nets. The global route for a net is found using geometric information specific to row-based FPGAs. The detailed route for a net in a channel is found using a greedy heuristic that tries to reduce segment wastage and the number of segments used. Critical path delays are updated using incremental static timing analysis. PRACT yielded up to a 29% improvement in delay and 33%

improvement in channel widths when compared to a place-and-route flow used at Texas Instruments (circa 1995).

Island style FPGAs (PROXI) [35]: The PROXI algorithm uses a cost function that is a linear, weighted function of the number of unrouted nets, and the critical path delay of the placement. No global routing is attempted. The interconnect structure of the FPGA is represented as a routing graph similar to the directed graph used by Pathfinder. For each placement move, the nets connecting the moved logic blocks are ripped up and added to a global queue of unrouted nets. Nets are rerouted using a maze routing algorithm augmented with a cost-to-target predictor. To keep runtime under control, the depth of the maze search is modulated as the annealing placement proceeds. The segmented nature of the routing resources is addressed by means of an explicit weighting scheme that encourages high fanout nets to use long segments, and low fanout nets to used shorter segments. This weighting scheme relies on the bounding box of the net being routed. Critical path delays are incrementally updated in a manner similar to PRACT. The placements produced by PROXI exhibited 8 – 15% delay improvement compared to Xilinx's XACT5.0 place-and-route flow.

The quality of the placement solutions produced by PRACT and PROXI was noticeably superior to commercial, state-of-the-art CAD flows at that time (circa 1995). The results were a strong validation of a simulated annealing based FPGA placement algorithm that is tightly coupled with routing heuristics. However, both algorithms have potential shortcomings from adaptability as well as CAD perspectives:

- The cost functions developed for the algorithms do not explicitly consider total wirelength or congestion. The only metric used to estimate the routability of a placement is the total number of unrouted nets. It can easily be seen that the total wirelength and congestion of a placement may change without affecting the number of unrouted nets. A cost function that is insensitive to such changes may allow wirelength and/or congestion to increase undesirably.
- The routing heuristics used by PRACT are tied to row-based FPGAs, and may be difficult to adapt to FPGA architectures that have different interconnect structures. At the same time, PROXI uses bounding box estimates to dynamically weight nodes of the routing graph when routing nets. This dynamic weighting approach is targeted at island-style architectures that have segmented routing resources.
- PROXI's routing algorithm does not allow sharing of routing nodes by multiple signals. Disallowing sharing prevents PROXI from leveraging the negotiation-based congestion resolution heuristics from the Pathfinder algorithm.

The approaches and techniques surveyed in this section are either targeted to certain architectural styles, or use relatively weak estimates of routability during the placement process. No clear cost formulation or technique emerges that can be used to produce high quality placements across a range of architecturally unique FPGAs. Research in FPGA architectures would stand to benefit from a placement algorithm that can quickly be retargeted to relatively diverse FPGA architectures, while producing high quality results at the same time.

Chapter 5: Independence – Architecture Adaptive Routability Driven Placement for FPGAs

In Chapter 4, we demonstrated that VPR's cost formulation might not adapt to non island-style FPGAs. We then postulated that an integrated place-and-route approach that tightly couples placement with an architecture-adaptive router (Pathfinder) is probably a more appropriate architecture-adaptive placement approach. In this chapter we describe Independence, an architecture-adaptive routability-driven FPGA placement algorithm. Pseudo code for the algorithm appears in Figure 5-1. The remainder of this chapter is a consolidated explanation of the pseudo code shown in Figure 5-1.

5.1 Placement Heuristic and Cost Formulation

Since simulated annealing has clearly produced some of the best placement results reported for FPGAs [7], we chose to use simulated annealing as Independence's placement heuristic. Independence's cooling schedule is largely an adoption of VPR's cooling schedule. This is because VPR's cooling schedule is adaptive, and incorporates some of the most powerful features from earlier research in cooling schedules. For similar reasons, we chose an auto-normalizing formulation for Independence's cost function. The main benefit of using normalization variables is that changes in cost of a placement do not depend on the actual magnitude of the cost variables. This makes the cost function adaptive, since the size of a netlist or the target architecture does not skew cost calculations. Independence's cost function is described in Equation 5.1

Equation 5.1: $\Delta C = \Delta WireCost / prevWireCost + \lambda * \Delta CongestionCost / CongestionNorm$

WireCost – The wire cost of a placement (Equation 5.2) is calculated by summing the number of routing resources used by each signal in the placed netlist. Routing resource usage is measured by simply traversing the route-tree of each signal and increasing *WireCost*. In Equation 5.2, N is the number of signals in the netlist, and $NumRoutingResources_i$ is the number of routing resources in the route tree of signal i. The normalization variable *prevWireCost* in Equation 5.1 is equated to the *WireCost* of a placement before a placement move is attempted.

```
Independence(Netlist, G(V,E)){
   // Create an initial random placement.
   createRandomPlacement(Netlist, G(V,E));

   N = set of all nets in Netlist;

   // Freely route all nets in N; similar to Pathfinder's first routing iteration. R contains the complete, current routing of the
   // nets in N at any time during the placement.
   R = routeNets(N, G(V,E));

   // Calculate the cost of the placement.
   C = calculateCost(R, G(V,E));

   // Calculate the starting temperature of the anneal.
   T = StartTemperature(Netlist, G(V,E), R);

   while(terminatingCondition() == false){
      while(innerLoopCondition() == false){
         // Randomly generate the two locations involved in the move.
         (x0,x1) = selectMove(G(V,E));

         // Get the nets connected to the logic blocks mapped to x0 and/or x1.
         Nx = getNets(x0, x1);

         // Cache the routes of the nets connected to the logic blocks mapped to x0 and/or x1.
         CacheR = getRoutes(Nx);

         // Rip up the nets connected to the logic blocks mapped to x0 and/or x1.
         R = R - cacheR;

         // Swap the logic blocks mapped to x0 and/or x1. Update the source/sink terminals of the nets in Nx to
         // reflect the new placement.
         swapBlocks(x0, x1);

         // Reroute the nets connected to the logic blocks that are now mapped to x0 and/or x1.
         R = R + routeNets(Nx, G(V,E));

         // Calculate the change in cost due to the move
         newC = calculateCost(R, G(V,E));
         ΔC = newC - C;

         if(acceptMove(ΔC, T) == true){
            // Accept the move.
            C = newC;
         }
         else{
            // Restore the original placement and routing
            swapBlocks(x0, x1);
            R = R - getRoutes(Nx) + cacheR;
         }
      }

      // Update temperature T.
      T = updateTemp();

      // Update history costs.
      updateHistoryCosts(R, G(V,E));

      // Refresh routing.
      R=Φ;
      R=routeNets(N, G(V,R));
   }
}
```

Figure 5-1: Pseudo code for the Independence algorithm. Brief comments accompany the code, and are shown in gray.

Equation 5.2: $$WireCost = \sum_{i=1}^{N} NumRoutingResources_i$$

CongestionCost – The congestion cost (Equation 5.3) represents the extent to which the routing resources are congested in a given placement, and is calculated by summing the number of signals that overuse each congested resource. The congestion cost of a placement is calculated by traversing the routing graph and increasing *CongestionCost* when a shared resource is encountered. In Equation 5.3, $Occupancy_i$ is the number of signals that are currently using routing resource i, $Capacity_i$ is the capacity of routing resource i, and R is the total number of vertices in the routing graph of the target architecture. It could be argued that *CongestionCost* renders *WireCost* redundant, since the objective of an FPGA placement algorithm is to produce a routable netlist. However, a cost function that is unaware of changes in wire cost will not recognize moves that might improve future congestion due to reductions in routing resource usage.

Equation 5.3: $$CongestionCost = \sum_{i=1}^{R} max(Occupancy_i - Capacity_i, 0)$$

CongestionNorm: This is the auto-normalization term for the *CongestionCost* of a placement. Note that the congestion cost of the placement cannot be used as a normalizing factor, since *CongestionCost* might be zero towards the end of the annealing process. In our current implementation of Independence, we equate *CongestionNorm* to *prevWireCost*.

λ – This weighting parameter (Equation 5.1) controls the relative importance of changes in wire and congestion costs. Since *CongestionNorm* is continuously recalculated as the placement algorithm progresses, the collective λ / *CongestionNorm* term also changes dynamically. As a result, the relative importance of changes in congestion cost varies with time. This behavior is similar to that of the cost function presented in [50], in which the weighting parameter of an individual cost term (overlap penalty) was dynamically varied during the annealing process. This dynamic parameter tuning approach proved very effective in eliminating overlap penalty while minimizing increases in wirelength.

5.2 Integrating Pathfinder

FPGA routing is a computationally intensive process. Admittedly, it is infeasible to reroute all the signals in a netlist after each placement move. Our solution is to start out with an initially complete routing, and then incrementally reroute signals during placement. Specifically, only the signals that connect to the logic blocks involved in a move are ripped up and rerouted. This is based on the intuition that for any given move, major changes in congestion and delay will be primarily due to the rerouting of signals that connect moved logic blocks.

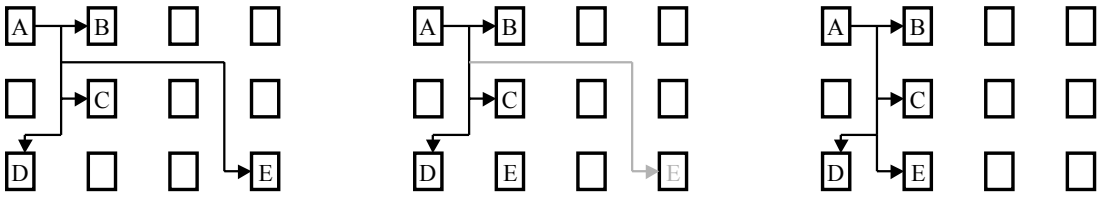

Figure 5-2: Logic block E is moved to the location immediately to the right of D. Its input branch (shown in gray) is ripped up, and a new route is found from the partial route-tree that connects logic blocks A, B, C and D.

Signals that sink at the moved logic blocks are handled differently from signals that originate at the moved logic blocks. When a logic block is moved during the placement process, only the branch that connects an input nets to the logic block is ripped up and rerouted. This approach is similar to Pathfinder's signal router, which uses the entire partially routed net as a starting point for the search for a new sink terminal. Ripping up and rerouting only branches is based on the assumption that the relocation of a single terminal of a multi-terminal net will not drastically alter the net's route. The runtime benefits of only routing branches are compelling, especially because FPGA logic units generally have a relatively large number of inputs. Figure 5-2 illustrates the process of ripping up and rerouting the input branches of moved logic blocks.

While input nets are partially ripped up and rerouted, the output nets of moved logic blocks are completely rerouted. Merely routing the output of a moved logic block to the nearest point in the partial route-tree could produce a poor route. Figure 5-3 illustrates the benefits of completely rerouting the output net of a moved logic block.

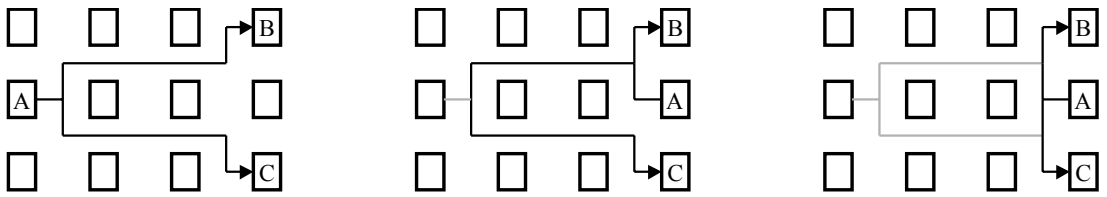

Figure 5-3: Logic block A is moved to the location between B and C. If we reroute from A to the partial route-tree, the resultant route requires far more routing than is necessary. Ripping up and rerouting the entire net produces a better routing.

Since we only attempt an incremental rip-up and reroute after every move, the routes found for signals during the early parts of an annealing iteration may not accurately reflect the congestion profile of the placement at the end of an iteration. Hence, we periodically refresh the netlist's routing by ripping up and rerouting all signals. This ripup-and-reroute step is equivalent to performing the final iteration of a Pathfinder run. Currently, the netlist is ripped up and rerouted at the end of every temperature iteration.

In light of the fact that the placement of a netlist is constantly changing during simulated annealing, it is necessary to examine whether Pathfinder's cost function is directly applicable to finding routes during incremental rip-up and reroute. When routing a signal, Pathfinder uses the number of signals currently sharing a routing node (*presentSharing*), and the history of congestion on the node (*historyCost*) to calculate the cost of the routing node. Since the netlist is completely routed at any given point in the placement process, the current sharing of routing nodes can easily be calculated, and thus we directly adopt Pathfinder's *presentSharing* cost term.

Pathfinder's history cost term is motivated by the intuition that routing nodes that have been historically congested during the routing process probably represent a congested area of the placed netlist. Thus, if a routing node is shared at the end of a routing iteration, its history cost is increased by a fixed amount to make the node more expensive during subsequent iterations. Note that the process of updating history costs during a Pathfinder run makes history cost an increasing function. An increasing history cost formulation is inappropriate for Independence. An increasing history cost would reflect the congestion on a routing node during the entire placement process. However, since placements are in constant flux during the placement process, the congestion on a routing node during the early stages of the annealing process (when placements are very different) might not be relevant to the routing process towards the end.

Independence uses a decaying function to calculate history costs during incremental rip-up and reroute. Specifically, we use a mathematical formulation that decreases the relevance of history information from earlier parts of the placement process. Currently, we update history costs once every temperature iteration based on the assumption that the number of signals ripped up and rerouted during a temperature iteration is roughly equivalent to the number of signals routed during a single or small number of Pathfinder iterations. The history cost of a routing node during a temperature iteration '*i+1*' is presented in Equation 5.4.

Equation 5.4:

if (shared)

$$historyCost_{i+1} = \alpha * historyCost_i + \beta$$

else

$$historyCost_{i+1} = \alpha * historyCost_i$$

In Equation 5.4, i is a positive integer, and α and β are empirical parameters. Currently, $\alpha = 0.9$ and $\beta = 0.5$. Thus, the history cost of a shared routing node during a new iteration is determined by 90% of the history cost during earlier iterations plus a small constant. As an example, the history cost of a node that is shared during the first five iterations progressively goes from 0 to 0.5, to 0.95, to 1.36, and to 1.72. In cases where a routing node is not shared during a temperature iteration, its history cost is allowed to decay as per Equation 5.4.

As a final note, we would like to point out that congestion plays two roles in the Independence algorithm. First, the total congestion cost of a placement plays a direct role in contributing to the overall cost of a placement (Equation 5.1). We make the task of eliminating congestion an explicit goal of the placement process. At the same time, we also use Pathfinder's congestion resolution mechanism during incremental rip-up and reroute, and at the end of every temperature iteration to eliminate sharing. Thus, Independence uses a two-pronged approach to eliminate congestion during the placement process.

In the next chapter, we demonstrate Independence's adaptability to three architectures that have clearly different interconnect structures. Further, we also present an empirical study of the effect of the congestion weighting parameter λ on the quality of the solutions produced by the Independence algorithm.

Chapter 6: Validating Independence

The objective of our validation strategy is to demonstrate Independence's adaptability to different interconnect styles. Our experiments target three interconnect structures; island-style, tree-based (HSRA), and a one-dimensional architecture that has limited inter-track switching capabilities (RaPiD). The main reasons for selecting these as target architectures are:

- Each of the three architectures have clearly different interconnect structures. Targeting Independence to architectures with different interconnect structures will assess its adaptability.
- The existence of place-and-route tools for all three architectures. This allows us to directly compare the quality of the placements produced by Independence with those produced by architecture specific placement techniques.

6.1 Island-Style Architectures

6.1.1 Experiment 1

Our first experiment compares the placements produced by Independence with VPR when targeted to a clustered, island-style architecture. Each logic block cluster in this architecture has eighteen inputs, eight outputs, and eight 4-LUT/FF pairs per cluster. The interconnect structure consists of staggered length four track segments and disjoint switchboxes. The input pin connectivity of a logic block cluster is 0.4*W (where W is the channel width) and output pin connectivity is 0.125*W. The island-style architecture described here is similar to the optimal architecture reported in [31].

Table 6-1 lists minimum track counts obtained on routing placements produced by VPR and Independence. Column 1 lists the netlists used in this experiment, column 2 lists the total number of logic blocks plus IO blocks in the netlist, column 3 lists the total number of nets in the netlist, column 4 lists the size of the minimum square array required to just fit the netlist, column 5 lists the minimum track counts required to route the placements produced by VPR, and column 6 reports the minimum track counts needed to route[4] placements produced by Independence. The final row in Table 6-1 lists the sum of the minimum track counts (which is our quality metric for all experiments presented in this chapter) required by VPR and Independence across the benchmark set.

[4]The placements produced by VPR and Independence are both routed using VPR's implementation of the Pathfinder algorithm.

Table 6-1: A comparison of the placements produced by VPR and Independence.

Netlist	Nblocks	Nets	Size	VPR	Ind
s1423	51	165	6x6	17	18
term1	77	144	6x6	17	17
vda	122	337	9x9	33	33
dalu	154	312	8x8	25	26
x1	181	352	10x10	22	23
apex4	193	869	13x13	60	61
i9	195	214	7x7	19	19
misex3	207	834	14x14	45	48
ex5p	210	767	12x12	60	60
alu4	215	792	14x14	39	41
x3	290	334	8x8	26	25
rot	299	407	8x8	27	29
tseng	307	780	12x12	34	36
pair	380	512	9x9	36	36
dsip	598	762	14x14	31	31
SUM				**491**	**503**

The track-counts listed in Table 6-1 show that the quality of the placements produced by Independence is within 2.5% of those produced by VPR. We consider this a satisfactory result, since it demonstrates that Independence can target island-style FPGAs and produce placements that are close in quality to an extensively tuned, state-of-the-art placement tool.

6.1.2 Experiment 2

Our second experiment studies Independence's adaptability to routing-poor island-style architectures. The philosophy behind routing-poor architectures [8,16] is increased silicon utilization through efficient use of the interconnect structure (which often accounts for ~90% of the total area in current FPGA families). Routing-poor architectures attempt to increase interconnect utilization at the expense of logic utilization. This is in direct contrast to VPR's exploratory style that fixes logic utilization, and then increases interconnect richness until a netlist's placement is successfully routed. Figure 6-1 (top left) shows a placement produced by VPR for the netlist *alu2* on a target architecture[5] that has four times as many logic blocks as a minimum size square array required to fit the netlist. VPR's router needs five tracks to route this placement. Our first observation is the tightly packed nature of the placement in Figure 6-1 (top left), and our second observation is that the placement produced by VPR does not

[5]Each logic block has a single LUT/FF pair, and the interconnect structure contains only length-one wire segments. This is the VPR "challenge" architecture [3].

change with the actual number of tracks in the target architecture. As a result, VPR is unable to produce routable placements for *alu2* on target architectures that have less than five tracks. VPR's limited adaptability to routing-poor architectures is a direct consequence of VPR's semi-perimeter based cost formulation that has no knowledge of the actual number of routing resources in the target device.

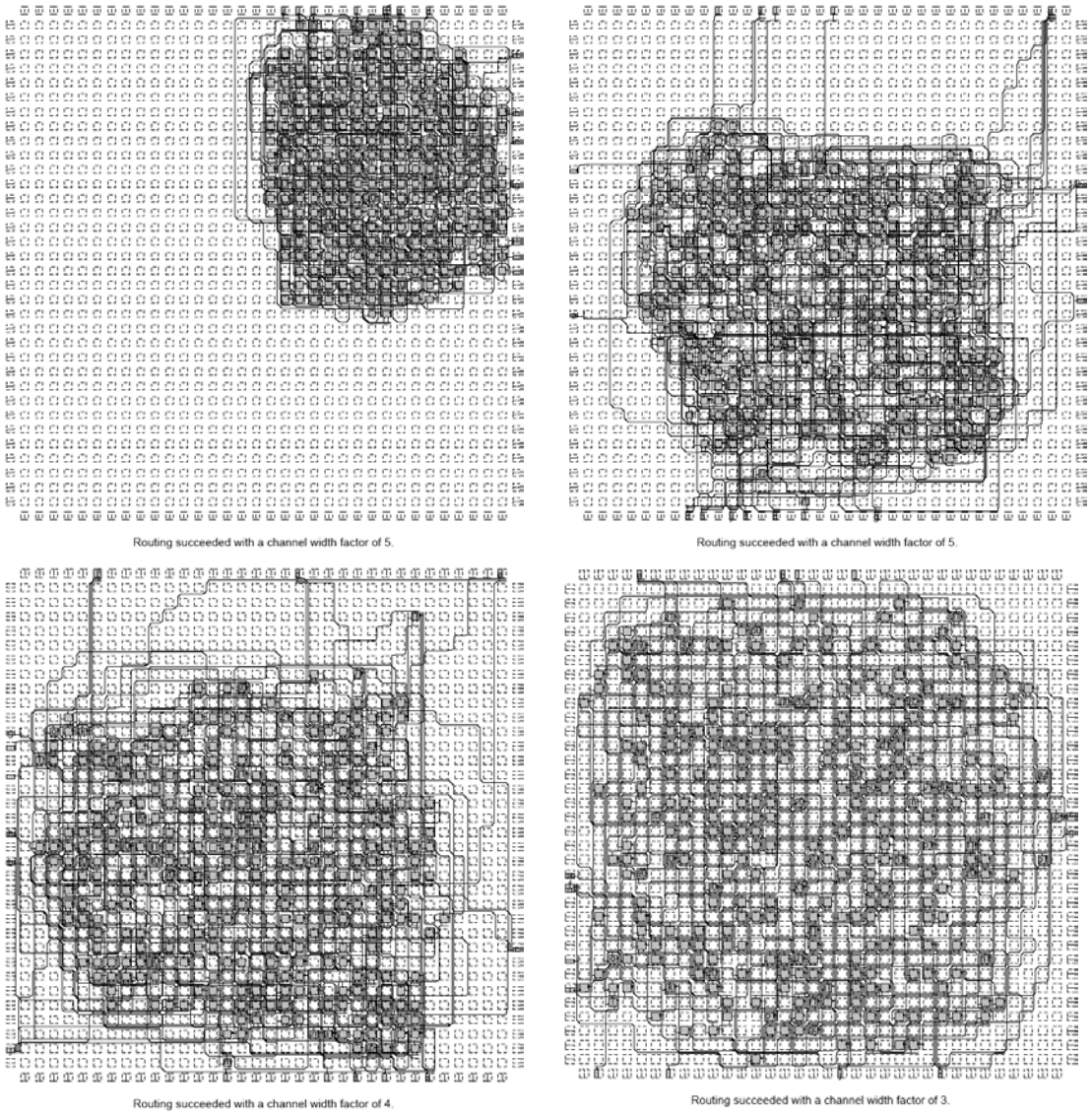

Figure 6-1: A placement produced by VPR for *alu2* on a 34x34 array (top left). VPR needed 5 tracks to route this placement. Placements produced by Independence for *alu2* on a 34x34 array that has 5 (top right), 4 (bottom left) and 3 (bottom right) tracks respectively.

Unlike VPR, Independence's integrated approach that tightly couples the placement algorithm with an architecture adaptive router is in fact able to produce routable placements on routing-poor island-style architectures. Figure 6-1 shows successfully routed placements produced by Independence on 34x34 arrays that have five (Figure 6-1 top right), four (Figure 6-1 bottom left) and three tracks (Figure 6-1 bottom right) respectively.

Table 6-2: Quantifying the extent to which Independence adapts to routing-poor island-style architectures.

Netlist	N_{blocks}	W_{VPR}	$1.0*W_{VPR}$	$0.9*W_{VPR}$	$0.8*W_{VPR}$	$0.7*W_{VPR}$	$0.6*W_{VPR}$	$0.5*W_{VPR}$
s1423	51	17	17	16	14	12	11	9
vda	122	33	33	30	27	24	20	17
rot	299	30	30	27	24	21	18	15
alu4	215	37	37	34	30	26	23	19
misex3	207	43	43	39	35	31	26	22
ex5p	210	52	52	47	42	37	32	26
tseng	307	33	33	30	27	24	20	17
apex4	193	52	52	47	42	37	32	26
diffeq	292	31	31	28	25	22	19	16
dsip	598	34	34	31	28	24	21	17

Table 6-2 shows the extent to which Independence is able to adapt to routing-poor island-style FPGAs. The parameters of the target array are identical to those used in *Experiment 1* (Section 6.1.1). The only exception is the logic capacity, which is four times (the width and height of the target array are each 2X the minimum required to fit the netlist) that of a minimum size square array. Column 1 lists the netlists used in the experiment, column 2 lists the number of logic blocks plus IO blocks in the netlist, and column 3 lists the minimum track counts needed by VPR to route each netlist. Let the minimum track count needed by VPR to route a netlist be W_{VPR}. Columns 4 through 9 list the number of tracks in a target architecture that has $1.0*W_{VPR}$, $0.9*W_{VPR}$, $0.8*W_{VPR}$, $0.7*W_{VPR}$, $0.6*W_{VPR}$, and $0.5*W_{VPR}$ tracks respectively. In Columns 4 – 9, a lightly shaded table entry (black text) means that Independence produces a routable placement on that architecture, while a dark shaded entry (white text) means that Independence is unable to produce a routable placement. So, for example, the lightly shaded table entry 37 for the netlist *ex5p* means Independence produces a routable placement for *ex5p* on a 37-track (0.7*52) architecture. Similarly, the dark shaded entry 32 for *ex5p* means that Independence fails to produce a routable placement for *ex5p* on a 32-track (0.6*52) architecture. The results in Table 6-2 show that Independence produces up to 40% better placements than VPR on routing-poor island-style interconnect structures. Note that VPR does not possess the ability to adjust to routing-poor architectures, and thus cannot use the extra space to reduce track count.

Finally, since the height and width of a target array in *Experiment 2* is approximately twice the minimum required, the bandwidth[6] of any target array in *Experiment 1* is approximately equal to the bandwidth of the corresponding target array in *Experiment 2* at $0.5*W_{VPR}$. Coincidentally, $0.5*W_{VPR}$ is also the point at which Independence is not able to produce any further reductions in track count. Thus, although the target arrays are of different sizes, both VPR and Independence produce placements that require comparable bandwidths.

6.2 Hierarchical Architectures – Experiment 3

Our third experiment targets an architecture (HSRA) that has a hierarchical, tree-based interconnect structure (Figure 6-2). The richness of HSRA's interconnect structure is defined by its *base channel width* and interconnect *growth rate*. The base channel width '*c*' is the number of tracks at the leaves of the interconnect tree (in Figure 6-2, *c*=3). The growth rate '*p*' is the rate at which the interconnect grows towards the root (in Figure 6-2, *p*=0.5). The growth rate is realized using the following types of switch-blocks:

- *Non-compressing* (2:1) switch blocks - The number of root-going tracks is equal to the sum of the number of root-going tracks of the two child switch blocks.
- *Compressing* (1:1) switch blocks – The number of root-going tracks is equal to the number of root-going tracks of either child switch block.

A repeating combination of non-compressing and compressing switch blocks can be used to realize any value of *p* less than one. A repeating pattern of 2:1 → 1:1 switch blocks realizes *p*=0.5, while the pattern 2:1 → 2:1 → 1:1 realizes *p*=0.67.

In HSRA, each logic block has a single LUT/FF pair. The input-pin connectivity is based on a *c-choose-k* strategy [16], and the output pins are fully connected. The base channel width of the target architecture is eight, and the interconnect growth-rate is 0.5. The base channel width and interconnect growth rate are both selected so that the placements produced by HSRA's CAD tool are noticeably depopulated.

[6] The bandwidth is measured by the number of routing tracks that cut a horizontal or vertical partition of the target array.

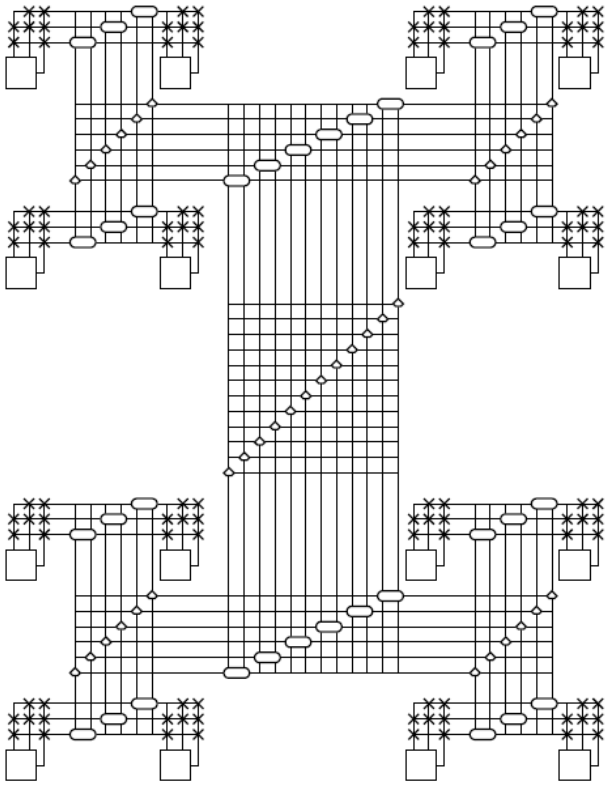

Figure 6-2: [16] An illustration of HSRA's interconnect structure. The leaves of the interconnect tree represent logic blocks, the crosses represent connection points, the hexagon-shaped boxes represent non-compressing switches, and the diamond-shaped boxes represent compressing switches. The base channel width of this architecture is three (*c*=3), and the interconnect growth rate is 0.5 (*p*=0.5).

Table 6-3 compares the minimum base channel widths required to route[7] placements produced by HSRA's placement tool and Independence. Column 1 lists the netlists used in this experiment, column 2 lists the number of LUTs in each netlist, column 3 lists the minimum base channel widths required to route placements produced by HSRA's placement tool, and column 4 lists the minimum base channel widths required to route placements produced by Independence. To ensure a fair comparison, Independence is targeted to architectures with the same horizontal span (*lsize* as defined in [16]) and interconnect levels as required by HSRA's placement tool (Figure 6-3). Overall, Independence is able to produce placements that require 21% fewer tracks compared to HSRA's placement tool.

[7]The placements produced by HSRA's CAD tool and Independence were both routed using HSRA's router (*arvc*).

Table 6-3: Independence compared to HSRA's placement tool.

Netlist	N_{LUTs}	HSRA	Ind
mm9b	120	10	9
cse	134	11	8
s1423	162	10	8
9sym	177	11	8
ttt2	198	10	8
keyb	209	12	9
clip	243	11	9
term1	246	11	10
apex6	258	10	10
vg2	277	11	9
frg1	282	12	10
sbc	332	11	8
styr	341	12	9
i9	347	11	9
C3540	382	11	8
sand	406	12	9
x3	441	11	10
planet	410	12	9
rd84	405	12	8
dalu	502	12	8
SUM		**223**	**176**

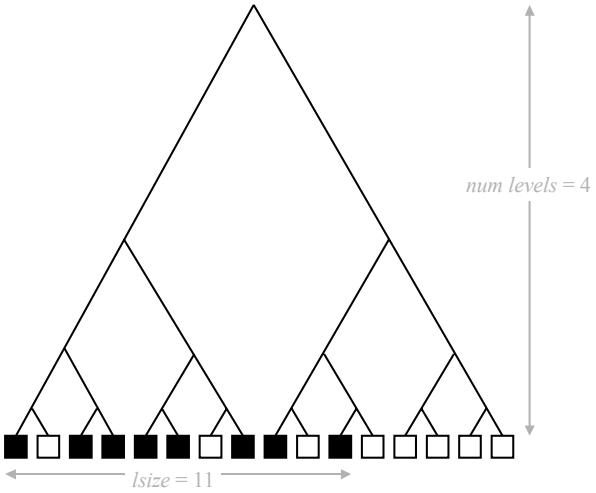

Figure 6-3: Although there are only eight logic blocks (black boxes) in the netlist, HSRA's placement tool spreads placements out to match the interconnect requirements of the netlist with the interconnect bandwidth provided by the architecture. In this example, the placement has an *lsize* = 11 and requires $\log_2 16$ = 4 interconnect levels. In medium-to-high stress cases, both lsize and num levels are inversely proportional to the base channel width of the device.

6.3 RaPiD – Experiment 4

Our fourth experiment targets the RaPiD architecture. RaPiD's interconnect structure consists of segmented 16-bit buses. There are two types of buses; *short* buses provide local communication between logic blocks, while *long* buses can be used to establish longer connections using bidirectional switches called *bus-connectors* (shown as the small square boxes in Figure 6-4). RaPiD's interconnect structure is relatively constrained because there is no inter-bus switching capability in the interconnect structure. A bus-connector can only be used to connect the two bus-segments incident to it. Thus, RaPiD's interconnect structure is an interesting candidate for a routability-driven placement algorithm.

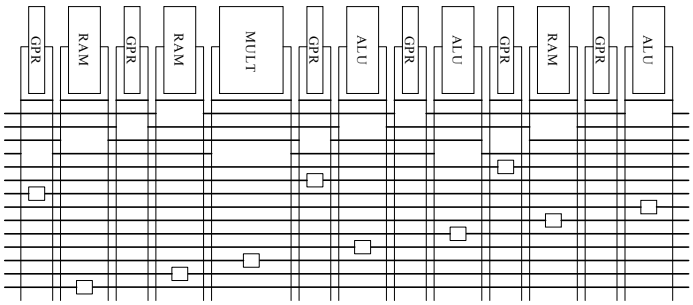

Figure 6-4: RaPiD's interconnected structure consists of segmented 16-bit buses. The small square boxes represent bidirectional switches called bus connectors.

Table 6-4 presents the results of *Experiment 4*. Column 1 lists the netlist, column 2 lists the number of RaPiD cells in the target array, column 3 lists the minimum track-count required by placements produced by the placer described in [40], and column 4 lists the minimum track-count required to route placements produced by Independence. Overall, the min track-counts required by RaPiD's placer and Independence were within 0.7%.

Table 6-4: A comparison of the track-counts required by a placement tool targeted to RaPiD and Independence.

Netlist	Ncells	RaPiD	Ind
matmult4	16	12	11
firtm	16	9	11
sort_rb	8	11	11
sort_g	8	11	11
firsymeven	16	8	9
cascade	16	10	10
sobel	18	15	13
fft16	12	11	12
imagerapid	14	12	11
fft64	24	29	28
log8	48	12	14
SUM		**140**	**141**

6.4 The Effect of Congestion Weighting Parameter λ

Independence's cost function depends on the wire cost and congestion cost of a placement (Equation 5.1, Equation 5.2, and Equation 5.3). The wire cost is the number of routing resources required by a fully routed placement, and the congestion cost is a measure of the total congestion. Independence's cost function is reproduced in Equation 6.1.

Equation 6.1: $\Delta C = \Delta WireCost / prevWireCost + \lambda * \Delta CongestionCost / CongestionNorm$

In this section, we empirically study the effect of the weighting parameter λ on the quality of the placements produced by Independence. Figure 6-5 shows the variation in placement quality vs. the weighting parameter λ. The x-axis represents different values of λ, and the y-axis represents minimum track counts normalized to the lowest track-count produced by Independence across the λ values.

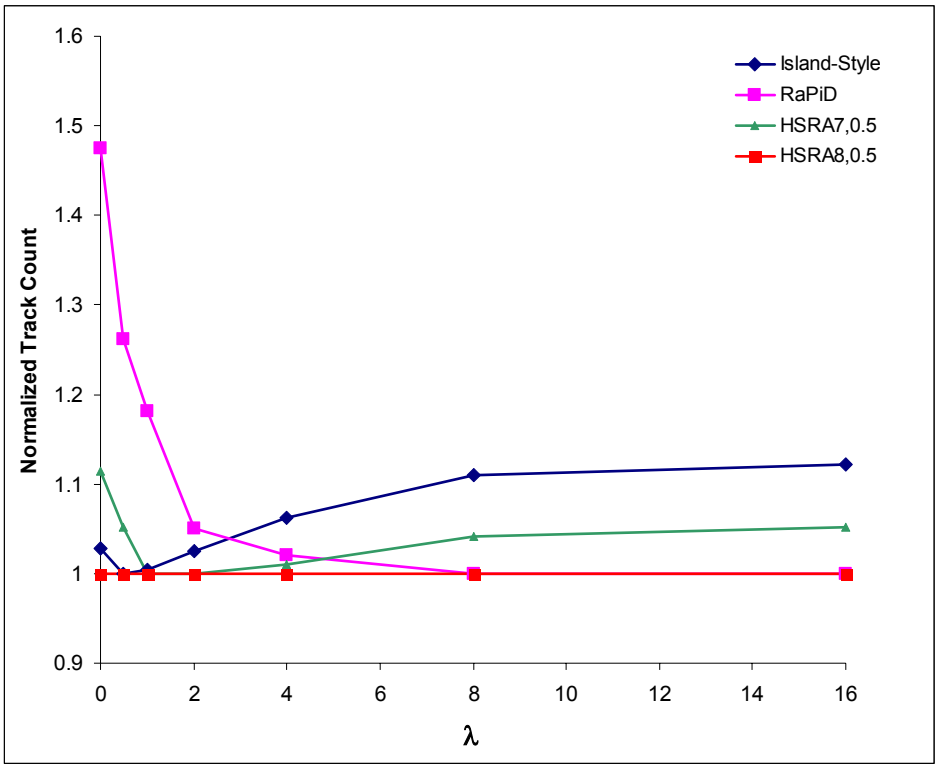

Figure 6-5: The effect of weighting parameter λ on the quality of the placements produced by Independence.

The setup and parameters that we used to obtain the four curves in Figure 6-5 are as follows:

- Island-Style – The Island-Style curve represents normalized track-counts obtained on using Independence to place netlists on a clustered, island-style architecture. The parameters of the target devices are identical to the devices used in *Experiment 1* and *Experiment 2* (Section 6.1). Each netlist is placed on a minimum size square array.
- RaPiD – The RaPiD curve represents normalized track counts obtained on using Independence to place netlists on the RaPiD architecture. Each netlist was placed on a RaPiD array that had the minimum number of cells required to just fit the netlist.
- HSRA8,0.5 – This curve represents the minimum track counts obtained when using the parameters in *Experiment 3* (Section 6.2). Specifically, Independence is targeted to an architecture that has a base channel width of eight, and an interconnect growth rate of 0.5. Further, Independence is constrained to produce placements that do not exceed the *lsize* or number of levels reported by HSRA's placement tool.

- HSRA7,0.5 – This curve represents the minimum track counts obtained on using Independence to place netlists on an architecture that has a base channel width of seven, and an interconnect growth rate of 0.5. Note that we target Independence to a complete binary tree that has twice the minimum number of logic units required by the netlist. For example, if there are 150 logic blocks in the netlist, then the size of the target array is 2*256 = 512 logic units. Further, we relax the *lsize* constraint to allow Independence to use the entire target device during placement.

In Figure 6-5, a non-zero value of λ produces placements of better quality than $\lambda = 0$ on three out of the four cases that we investigated. The island-style curve has minimum at $\lambda = 0.5$, the RaPiD curve asymptotically approaches a minimum at $\lambda = 8$, and the HSRA7,0.5 curve's minima occur at $\lambda = 1$ and $\lambda = 2$. The fourth case (HSRA8,0.5) is insensitive to λ. We suspect that the parameters that we used in the HSRA8,05 study may represent a aggressively high-stress case, which might be the reason for the HSRA8,0.5 curve's insensitivity to λ. On the other hand, the HSRA7,0.5 curve is sensitive to λ. The HSRA7,0.5 case is probably lower stress because of the relaxed *lsize* constraint and size of the target array.

The primary message of this study is that a non-zero value of λ will probably produce better placements than a cost function that depends on only wire cost. No strong conclusions can be made about a "magic" value for λ, or whether it is even possible to guess a ballpark value for λ given prior knowledge of the FPGA's interconnect structure. In view of these conclusions, a strong candidate for future work is the development of heuristics that auto-determine λ based on the characteristics of the netlist and the target interconnect structure.

6.5 Runtime

A comparison of placement runtimes on island-style interconnect structures is shown in Table 6-5. Column 1 lists benchmark netlists, column 2 lists the number of logic plus IO blocks in the netlist, column 3 lists the number of nets, column 4 lists the size of the target array, column 5 lists VPR's runtime, column 6 lists Independence's runtime, and column 7 lists the ratio between Independence's and VPR's runtime. All runtimes are in seconds. The most current version of our implementation requires between approximately three minutes (s1423) and seven hours (dsip). The implementation includes runtime enhancements based on the A* algorithm (Chapter 7). Note that dsip is one of the smaller netlists in the Toronto20 set. For larger netlists, Independence's runtime might be on the order

of days. Clearly, there is a compelling need to explore techniques that might reduce Independence's runtime.

Table 6-5: A comparison of placement runtimes on island-style structures.

Netlist	Nblocks	Nets	Size	VPR	Ind	Norm
s1423	51	165	6x6	0.3	192	640
term1	77	144	6x6	0.34	193	568
i9	195	214	7x7	0.71	555	782
dalu	154	312	8x8	0.95	1124	1183
vda	122	337	9x9	1	1187	1187
x3	290	334	8x8	1.25	1354	1083
rot	299	407	8x8	1.39	1925	1385
x1	181	352	10x10	1.29	2257	1750
pair	380	512	9x9	1.85	3365	1819
ex5p	210	767	12x12	2.6	5924	2278
apex4	193	869	13x13	2.82	7670	2720
tseng	307	780	12x12	2.75	8725	3173
misex3	207	834	14x14	3.08	10054	3264
alu4	215	792	14x14	3.1	10913	3520
dsip	598	762	14x14	4.95	24719	4994

6.6 Summary

The results of our experiments in Sections 6.1, 6.2 and 6.3 demonstrate Independence's adaptability to three different interconnect styles. The quality of the placements produced by Independence are within 2.5% of VPR, 0.7% of RaPiD's placement tool, and 21% better than HSRA's placement tool. Further, our experiment with routing-poor island-style structures shows that Independence is appropriately sensitive to the richness of interconnect structures. When considered together, the results presented in Sections 6.1, 6.2 and 6.3 are a clear validation of using an architecture-adaptive router to guide FPGA placement.

Finally, the empirical study presented in Section 6.4 shows that a congestion-aware placement cost function produces better results than a cost function that depends solely on wire cost. However, the exact value of the congestion weighting parameter λ differs across architectures.

Chapter 7: Accelerating Independence Using the A* Algorithm

The Independence algorithm integrates an adaptive, search-based router with a simulated annealing placement algorithm. Using a router in the simulated annealing inner loop is clearly a computationally expensive approach. In this chapter we discuss the A* algorithm, a technique that has been used to speed up Pathfinder with a negligible degradation in quality [33,49]. We also describe an adaptive technique that can be used to speed up Pathfinder (and consequently Independence) when routing netlists on FPGAs that have different interconnect structures.

The A* algorithm speeds up routing by pruning the search space of Dijkstra's algorithm. The search space is pruned by preferentially expanding the search wavefront in the direction of the target node. Thus, when the search is expanded around a given node, the routing algorithm expands the search through the neighbor node that is nearest the target node. This form of directed search is accomplished by augmenting the cost of a routing node with a heuristically calculated estimate of the cost to the target node.

Consider Equation 7.1, in which g_n is the cost of a shortest path from the source to node n, and h_n is a heuristically calculated estimate of the cost of a shortest path from n to the target node (hereafter, we refer to this estimate as a 'cost-to-target' estimate). The value f_n is the estimated cost of a shortest path from the source to the target that contains the node n. The A* algorithm uses f_n to determine the cost of expanding the search through node n. Note that Dijkstra's algorithm uses only g_n to calculate the cost of node n.

Equation 7.1: $f_n = g_n + h_n$

To guarantee optimality, the cost-to-target estimate h_n at a given node n must be less than or equal to the actual cost of the shortest path to the target. Overestimating the cost to the target node may provide even greater speedups, but then the search is not guaranteed to find an optimal path to the target.

7.1 The Heuristic Estimate

The heuristic cost-to-target estimator plays a crucial role in accelerating A* search. Two factors that influence the efficacy of the cost-to-target estimator are:

- *Accuracy*: The accuracy of the estimator directly influences the quality of the solutions produced by the A* algorithm. In the ideal case, a cost-to-target estimator will always return the exact cost of a shortest path from a node *n* to the target node. An estimator that returns the exact cost of the shortest path marches the A* search directly towards the target node, and no redundant nodes are expanded during this search. Thus, in the ideal case, we can find a lowest cost path to the target node in the shortest possible time.
- *Cost*: The cost of calculating the cost-to-target estimate may affect the speedups produced by the A* algorithm. A cost-to-target calculation is done for every node involved in the A* search, and thus cost-to-target estimates should be relatively simple to calculate. If the computational effort required to calculate the estimates is high, then the A* search may slow down. A cost-to-target estimator might deliberately mis-estimate costs to keep calculations simple, despite the fact that exact estimates can indeed be obtained.

In island-style FPGAs, the cost-to-target estimate of an interconnect wire w_n is the Pathfinder-based cost of a shortest path from w_n to the target sink terminal [6]. The estimate calculation requires a count of the number of wires on the shortest path. The number of wires is calculated using the coordinates of the logic units that are located at the extremities of w_n, the length of the wire w_n, and the coordinates of the target sink terminal. The length of wire w_n is measured in terms of the number of logic units spanned by w_n. In case the interconnect structure consists of wires that have different lengths, then the shortest path is assumed to consist entirely of wires of length w_n.

The calculation of cost-to-target estimates in island-style architectures relies on geometric information. As discussed in Section 4.1, non island-style interconnect structures might not conform to a two-dimensional geometric layout, and geometric information may not be useful in estimating the number of wires between a wire w_n and a target sink terminal. In the hierarchical tree-based interconnect structure shown in Figure 7-1, the number of wires between w_n and the target t is estimated using interconnect level numbers. The wire w_n's level number is 2, t's level number is 0, and the root switchbox's level number is 4. Thus, an estimate of the number of wires between w_n and t is equal to the sum of the number of wires from w_n to the root switchbox (4 – 2 = 2) and the number of wires from the root switchbox to t (4 – 0 = 4). This estimate is equal to six wires in all.

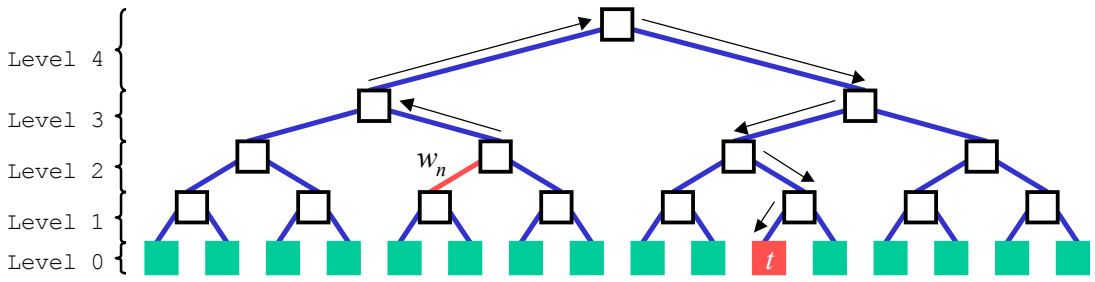

Figure 7-1: The number of wires between w_n and t is estimated using interconnect level numbers. In this case, there are 4 – 2 wires from w_n to the root switchbox plus 4 – 0 wires from the root switchbox to t.

The central calculation in computing cost-to-target estimates is determining the shortest path to the target in the absence of routing congestion. On island-style interconnect structures, geometric information is used to calculate the shortest path. Interconnect level information is used to calculate the shortest path in tree-based interconnect structures. In general, a cost-to-target estimator uses features that are specific to the interconnect structure. Our goal in this chapter is to develop a cost-to-target estimation technique that adapts to the target FPGA's interconnect structure. There are multiple reasons for the potential usefulness of an adaptive cost-to-target estimation scheme:

- *Usability Considerations*: Our vision of a production version of Independence is a stand-alone tool that will require minimal user intervention. An architecture-specific cost-to-target estimator may necessitate source code modifications and possible changes to Independence's interface on a per-architecture basis. We feel that users should not be expected to provide any architecture-specific enhancements to speed up Independence.
- *Cost of Computing Estimates*: Our approach to adaptive cost-to-target estimation requires minimal computation during the routing process. For each interconnect wire, we provide a pre-computed table of cost-to-target estimates for each sink terminal in the target device. Thus, every time a wire is expanded during A* search, the cost of obtaining a cost-to-target estimate is simply the cost of a table lookup. Note that the estimates produced by our technique are also guaranteed to be exact or underestimates.
- *Automatically Generated Architectures*: During domain-specific reconfigurable architecture generation [12,13], the nature of the reconfigurable device's interconnect structure may be significantly different across application domains. If an Independence and Pathfinder place-and-route flow is used to map applications to such architectures, then the cost-to-target estimator used by this flow must adapt to different interconnect structures. Expecting the user

to modify the flow to produce cost-to-target estimates goes against the underlying philosophy of automatic architecture generation.

7.2 The K-Means Algorithm

The ideal approach to an adaptive estimator is to simply use Dijkstra's algorithm to pre-compute and tabulate the cost of a shortest path from each wire to every sink terminal in the interconnect structure. This approach guarantees an exact estimate of the shortest path in the absence of routing congestion. However, while the computational complexity of this approach is manageable, the space requirements for routing-rich structures may explode. Assuming an island-style, 10-track, 100x100 FPGA that has only single-length segments, the memory required to store the cost-to-target lookup table would be measured in GigaBytes. Memory requirements of this size are probably impractical.

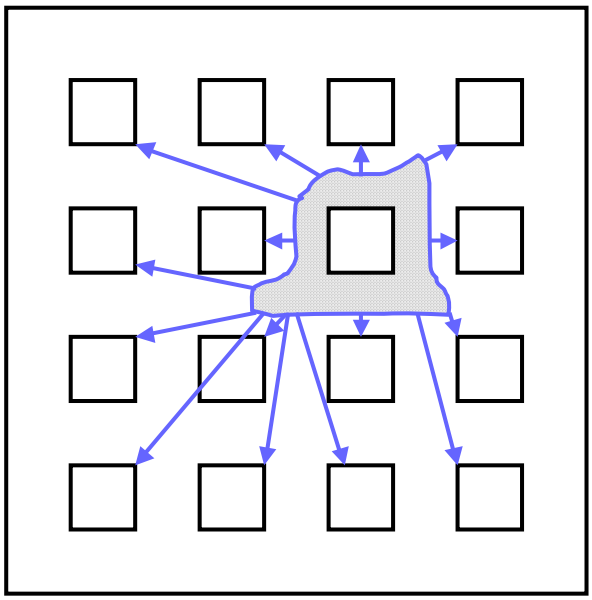

Figure 7-2: Calculating the cost-to-target estimates for a set of wires that share a table entry. The set of wires is collectively shown as the dotted region, and the small squares represent sink terminals in the interconnect structure. The cost-to-target estimate for a given sink terminal is the cost of a shortest path from the wire that is closest to the sink terminal.

Sharing a table entry among multiple wires that have similar cost-to-target estimates can reduce the memory requirement of the lookup table. For example, if a hundred wires share a table entry, the size of the table can be reduced by a hundred times. The cost-to-target estimate for a given sink terminal is the same for all wires that share the table entry, and can be calculated using a Dijkstra search that begins at the wire closest to the target (Figure 7-2). Specifically, the entire set of wires that share a table entry

constitutes a “super” source node for the Dijkstra search. In this manner, we ensure that the shortest-path estimate to a given sink terminal is the cost of the shortest path from the wire that is closest to the sink terminal.

The important question now is how to identify wires that should share a table entry. Clearly, we would like to identify clusters of wires that have similar cost-to-target estimates, so that we can collect them together in a set that points to a single entry in the cost-to-target lookup table. Clustering an island-style structure can be accomplished by associating each wire with its closest logic unit, and then clustering all wires associated with the same logic unit together. Since the logic and interconnect structures of an island-style FPGA are closely coupled, this approach may produce clusters of wires that have reasonably similar cost-to-target estimates.

On hierarchical structures, the accuracy of an associate-with-closest-logic-unit approach may not be quite as good. For example, consider the tree-like interconnect structure in Figure 7-3. The routing wire that is topmost in the interconnect hierarchy is equally close to all logic units, while the wires in the next level are equally close to half the logic units, and so on. Associating wires with individual logic units in a strictly hierarchical interconnect structure may result in large cost-to-target underestimates.

In Figure 7-3, assume that the wires shown in black are associated with the black logic unit, and that the cost-to-target estimates for the cluster wires have been calculated using the method illustrated in Figure 7-2. The wire that directly connects to the black logic unit will have a cost-to-target estimate of five for the logic units in the northeast, southeast and southwest quadrants of the architecture. Note that the actual cost is nine wires for the northeast quadrant, and ten for the southeast and southwest quadrants. Estimates that are a factor of two below exact might slow down the router considerably. However, every wire in the cluster shown in Figure 7-3 does not suffer from the same problem. The cluster wire that is topmost in the interconnect hierarchy (black vertical line down the middle of Figure 7-3) will have exact cost-to-target estimates for all logic units in the northeast, southeast and southwest quadrants, and underestimates for logic units in the northwest quadrant.

To summarize, one would expect the associate-with-closest-logic-unit approach to work well for island-style structures. However, due to the approach’s potential limitations on hierarchical structures, we feel that a more sophisticated technique might be necessary to produce reasonably accurate cost-to-target estimates across different interconnect styles.

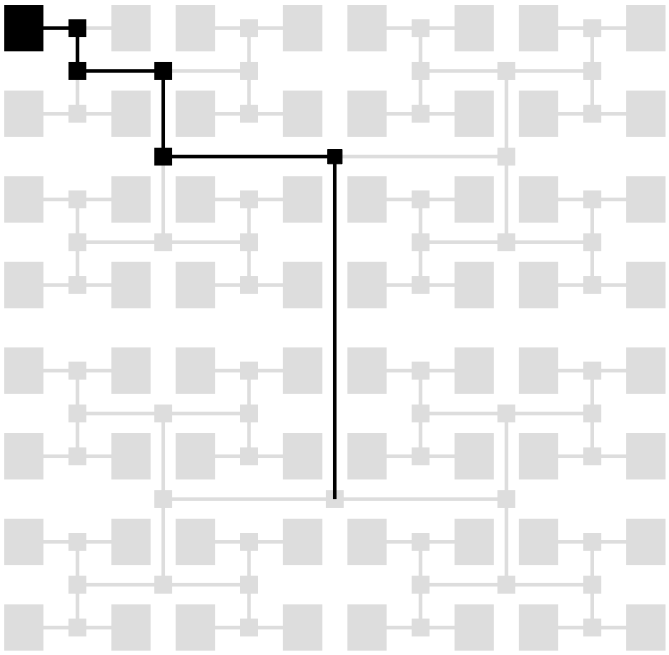

Figure 7-3: An example of a tree-based, hierarchical interconnect structure. Assume that the wires shown in black belong to the same cluster.

Our solution to the architecture adaptive clustering problem is to use the K-means algorithm [28]. K-means clustering is an iterative heuristic that is used to divide a dataset into K non-overlapping clusters based on a proximity metric. The proximity metric is used to calculate the similarity between data-points, and is designed based on the properties of the dataset. The user specifies the value of K and the nature of the proximity metric. In the beginning, the centroid[8] of each cluster is initialized to a randomly selected data-point. Next, the distance from each data point to the centroid of every cluster is calculated, and the data-point is assigned to the cluster with the closest centroid. At the end of a clustering iteration, when every data-point has been assigned to a cluster, the centroids of each cluster are recalculated and cluster assignments are removed. This process is carried out in an iterative fashion until a terminating condition is met. Pseudo code for the K-Means algorithm appears in Figure 7-4.

[8] The centroid of a cluster is the statistical mean of the coordinates of the data-points that belong to the cluster.

```
// D is the set of data-points in n-dimensional space that have to be divided into K clusters.
// The co-ordinates of a data-point d_i ∈ D are contained in the vector d_i.vec.
// d_i.vec is an n-dimensional vector.

K-Means {
    for i in 1…K {
        randomly select a data-point d_i from the set D.
        initialize the centroid of cluster clus_i to d_i.vec.
    }

    while (terminating condition not met) {
        for each d_i ∈ D {
            remove d_i's cluster assignment.
        }

        for each d_i ∈ D {
            for j in 1…K {
                diff_ij = vectorDifference (d_i.vec,clus_j.centroid)
            }
            assign d_i to the cluster clus_y such that diff_iy is minimum.
        }

        for j in 1…K {
            recalculate clus_j.centroid using the data-points currently assigned to
            clus_j.
        }
    }

}
```

Figure 7-4: Psuedocode for the K-Means clustering algorithm. When the algorithm completes execution, every data-point in the set D is assigned to a cluster.

There are several compelling reasons to use the K-means algorithm to adaptively cluster FPGA architectures. First, the algorithm is relatively easy to implement. Second, the performance of the K-means algorithm improves if the initial selection of K is roughly the same as the number of the natural clusters in the dataset. Third, the performance of the algorithm may be further improved if the initial cluster centers correspond to the centers of the natural clusters of the dataset. The regularity of FPGA architectures can be exploited to determine the number of clusters, and the center of each cluster. For example, a good starting point for cluster centers in an FPGA may be the logic units in the architecture, and an equal number of randomly selected routing wires in the interconnect structure.

7.3 Applying the K-Means Algorithm to Produce Interconnect Clusters

In this section we briefly describe our choices for the algorithmic parameters that characterize the K-Means algorithm.

Dataset (D): The dataset D simply consists of all the routing wires in the interconnect structure of the target device.

Number of Clusters (K): We experimentally determined that a value of K greater than or equal to the number of logic units in the target device is a reasonable choice. Section 7.4 describes the effect of K on the quality of clustering solutions.

Initial Seed Selection: The initial seeds consist of K/2 randomly selected logic-block output wires and K/2 randomly selected routing wires.

Coordinate Space and Proximity Metric: The most important consideration in applying the K-Means algorithm to solve the interconnect clustering problem is the proximity metric. Specifically, we need to determine a coordinate space that is representative of the A* cost-to-target estimate at each wire in the dataset. In our implementation, the coordinates of a routing wire represent the cost of the shortest path to a randomly chosen subset S of the sink terminals in the interconnect structure. The coordinates of each routing wire are pre-calculated using Dijkstra's algorithm and stored in a table.

If the number of sink terminals in S is n, then the coordinates of a routing wire $d_i \in$ D are represented by an n-dimensional vector $d_i.vec$. Each entry c_{ij} ($j \in 1 \ldots n$) in the vector $d_i.vec$ is the cost of a shortest path from the routing wire d_i to the sink terminal j. For a given wire d_i, the entire vector $d_i.vec$ is computed using a single pass of Dijkstra's algorithm. The vector $d_i.vec$ is used by the K-Means algorithm to calculate the "distance" between the wire d_i and the centroid of each cluster. The distance between d_i and a cluster centroid is defined as the vector difference between $d_i.vec$ and the cluster centroid.

Note that the size of S directly influences the memory requirements of our clustering implementation. In the extreme case when S contains every sink terminal in the target device, the memory requirements would match the prohibitively large requirements of a table that stores the cost of the shortest path from each routing wire to every sink terminal. This would clearly defeat the purpose of using a clustering

algorithm to reduce the memory requirements of an A* estimate table. It is thus necessary to sub-sample the number of sink terminals in the target device when setting up the set S.

Table 7-1: A comparison of the memory requirements of a clustering implementation that sub-samples the sink terminals with a table that stores estimates for every sink terminal in the target device. The sub-sample set S contains 6% of the sink terminals in the target device. Each entry in columns 3, 4 and 5 is in Giga Byte.

Size	ChanWidth	Full	SubSample	Estimates
10	10	0.0012	0.0001	0.0001
20	10	0.0151	0.0009	0.0007
30	10	0.0707	0.0043	0.0035
40	10	0.2152	0.0130	0.0106
50	10	0.5132	0.0310	0.0253
60	10	1.0474	0.0631	0.0518
70	10	1.9185	0.1155	0.0949
80	10	3.2449	0.1951	0.1607
90	10	5.1629	0.3103	0.2559
100	10	7.8268	0.4703	0.3882
110	10	11.4087	0.6854	0.5662
120	10	16.0986	0.9669	0.7994
130	10	22.1044	1.3275	1.0980
140	10	29.6517	1.7805	1.4735
150	10	38.9842	2.3406	1.9380
160	10	50.3636	3.0236	2.5045
170	10	64.0690	3.8462	3.1869
180	10	80.3979	4.8262	4.0001
190	10	99.6654	5.9825	4.9599
200	10	122.2044	7.3351	6.0828

Table 7-1 compares the memory requirements of a clustering-based implementation that sub-samples the sink terminals with a table that stores the cost of a shortest path from each routing wire to every sink terminal in the target device. The target architecture is assumed to be a square island-style array that has only single-length wire segments. In our calculations, we assume that the sizes of a floating point number, integer number, and a pointer are all four bytes. Column 1 lists the number of rows (and columns) in the target array, and column 2 lists the channel width of the target array. Let the total number of sink terminals in the target array be N_T. Column 3 lists the memory requirements of a table that stores the cost of a shortest path from each wire to every sink terminal in the target device (i.e. $|S| = N_T$), column 4 lists the size of a table that stores costs to only 6% of the sink terminals ($|S| = 0.06*N_T$),

and column 5 lists the size of a table that holds cost-to-target estimates for the clusters produced by a K-Means implementation where K = number of logic units in the target device. All memory requirements are reported in Gigabyte. It is clear from Table 7-1 that the memory requirements of a table that stores costs to every sink terminal in the target device quickly become impractical when compared to a sub-sampling based K-Means clustering approach.

Terminating Condition: The K-Means algorithm is terminated when less than 1% of the dataset changed clusters during the previous clustering iteration.

On completion of the clustering algorithm, the actual A* estimates for a cluster are calculated using Dijkstra's algorithm. To guarantee exact or under-estimates, the entire set of cluster members constitutes a "super" source node for the Dijkstra search. In this manner, we ensure that the shortest-path estimate to a given sink terminal is the cost of the shortest path from the cluster member that is closest to the sink terminal.

7.4 Results

We conduct four experiments to test the validity of using the K-Means algorithm to cluster the interconnect structure of an FPGA. The first experiment studies the effect of sub-sampling the sink terminals in the target device on the quality of clustering solutions. The second experiment studies the effect of the number of clusters (K) on quality, and the third experiment compares the quality of clustering-based A* estimates with heuristically calculated estimates. The fourth experiment quantifies the difference between the cost-to-target estimates produced by our clustering technique and actual shortest-path costs. To evaluate the adaptability of our techniques, we conduct the experiments on an island-style interconnect architecture and HSRA. Since the truest measure of the quality of an A* estimate is routing runtime, our quality metric is defined to be the runtime per routing iteration when routing a placement on the target device. The placements for our experiments on island-style architectures are obtained using VPR, and placements for experiments on HSRA are produced using Independence.

In *Experiment 1* and *Experiment 2*, we use a subset of the benchmark netlists in Table 6-1 (island-style) and Table 6-3 (HSRA). In *Experiment 3* and *Experiment 4*, we use an expanded set of benchmark netlists for both island-style structures and HSRA. The parameters of the target architecture used in island-style experiments are identical to the architecture used in Section 6.1.1, and the HSRA parameters are identical to those used in Section 6.2.

Before describing the results of our experiments, we briefly discuss the methods we used to calculate heuristic estimates on the island-style architecture and HSRA:

Island-Style: On the island-style architecture, the cost-to-target estimate of an interconnect wire w_n is the Pathfinder-based cost of a shortest path from w_n to the target sink terminal [6]. The estimate calculation requires a count of the number of wires on the shortest path. The number of wires is calculated using the coordinates of the logic units that are located at the extremities of w_n, the length of the wire w_n, and the coordinates of the target sink terminal. The length of wire w_n is measured in terms of the number of logic units spanned by w_n. In case the interconnect structure consists of wires that have different lengths, then the shortest path is assumed to consist entirely of wires of length w_n.

HSRA: On HSRA, the number of wires on a shortest path between w_n and the target t is estimated using interconnect level numbers. In Figure 7-1, the wire w_n's level number is 2, t's level number is 0, and the root switchbox's level number is 4. Thus, an estimate of the number of wires between w_n and t is equal to the sum of the number of wires from w_n to the root switchbox (4 – 2 = 2) and the number of wires from the root switchbox to t (4 – 0 = 4). This estimate is equal to six wires in all.

7.4.1 Experiment 1

This experiment studies the effect of sub-sampling the number of sink terminals in the target device. The value of K in this experiment is equal to the number of logic units in the target device. Figure 7-5 shows the variation in quality of clustering solutions. The x-axis represents the fraction of sink nodes that are used to represent the coordinates of each wire during K-Means clustering. The subset of sink nodes used in the experiment is randomly generated. The y-axis represents routing runtime measured in seconds per routing iteration. The blue curves show the variation in routing runtimes when using A* estimates produced by the clustering algorithm. The flat pink line shows the routing runtime when using architecture-specific heuristic A* estimates.

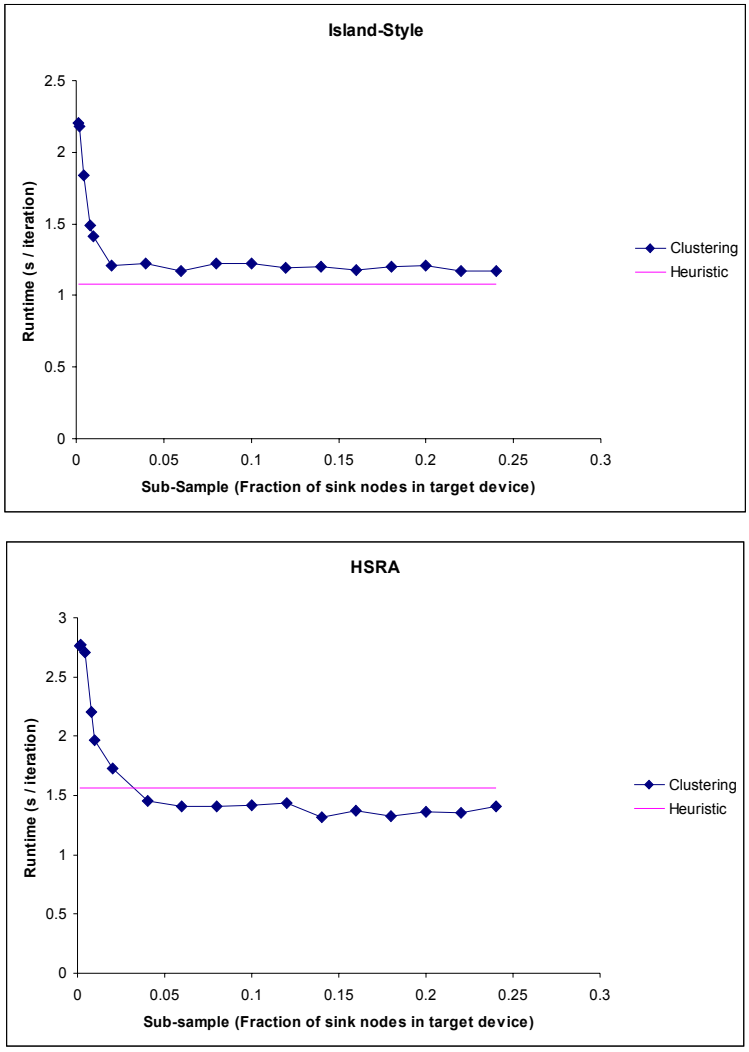

Figure 7-5: The effect of sub-sampling the number of sink nodes on routing runtime.

Figure 7-5 shows that using as little as 5% of the sink terminals during clustering may be sufficient to produce estimates that are comparable to heuristic estimates. This is not a surprising result. Due to the regularity of an FPGA's interconnect structure, a small subset of sink terminals may be sufficient in resolving the interconnect wires into reasonably formed clusters.

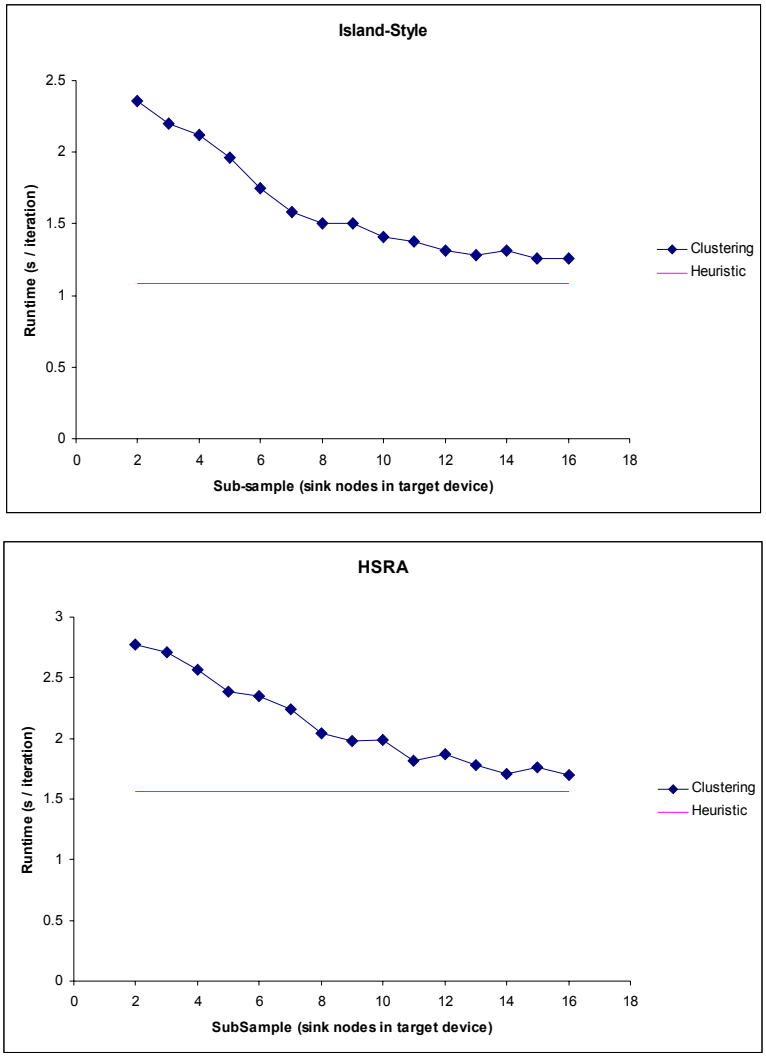

Figure 7-6: Using a small number of sink nodes may produce clustering solutions of acceptable quality.

Note that 5% of the sink terminals represents a variable number of sink terminals across the set of benchmarks. Depending on the size of the netlist, 5% of the sink terminals could be anywhere between two and fifty sink terminals. Figure 7-6 shows the results of a second study that evaluates the quality of clustering solutions when using a small, fixed number of sink terminals. Figure 7-6 shows that using a small number (say 16) of randomly selected sink nodes may be enough to produce clustering solutions that are within approximately 15% of heuristic estimates.

At first glance, the charts shown in Figure 7-6 might seem surprising. Using only sixteen sink terminals to resolve an interconnect structure into reasonably well-formed clusters might appear too good to be true. However, we think the reason for this behavior is the regularity of island-style architectures and HSRA. In Figure 7-7, if we were to use only one sink terminal in an island-style device, then we would get roughly circular clusters centered at the sink logic block. Increasing the number of sinks to two would cluster the routing wires at the intersection of the two circles together. This would result in "spotty" disconnected clusters. On increasing the number of sink terminals to three, it might be possible to separate routing wires into tighter clusters. This behavior has parallels with the triangulation method that is used to localize the position of a point in multi-dimensional space.

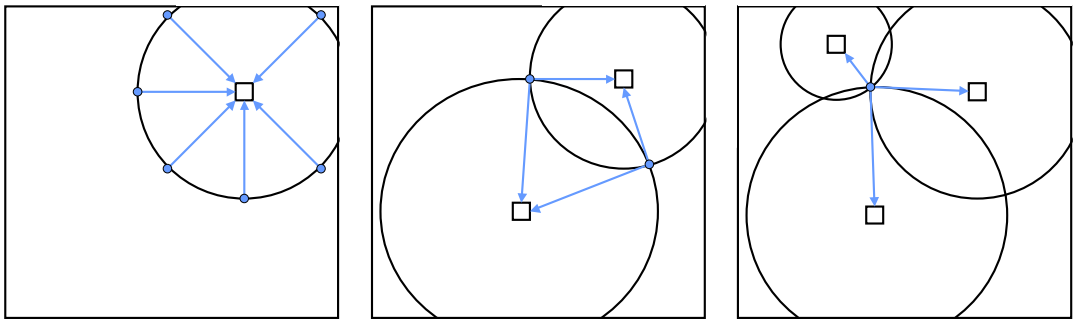

Figure 7-7: Cluster resolution on an island-style device as the number of sink terminals is increased from one to three.

In a similar fashion, merely increasing the number of sink terminals from one to two on HSRA improves the tightness of the clusters noticeably (Figure 7-8). In both top and bottom figures, the gray blobs encompass a single cluster. In the top figure, the number represents the coordinates of the corresponding interconnect wire. Recall that the coordinates of an interconnect wire correspond to the cost of a shortest path to the sink terminals in the set S. In the bottom figure, each pair of numbers represents the coordinates of the corresponding interconnect wire. Note the improvement in cluster tightness as the number of sink terminals is increased from one to two.

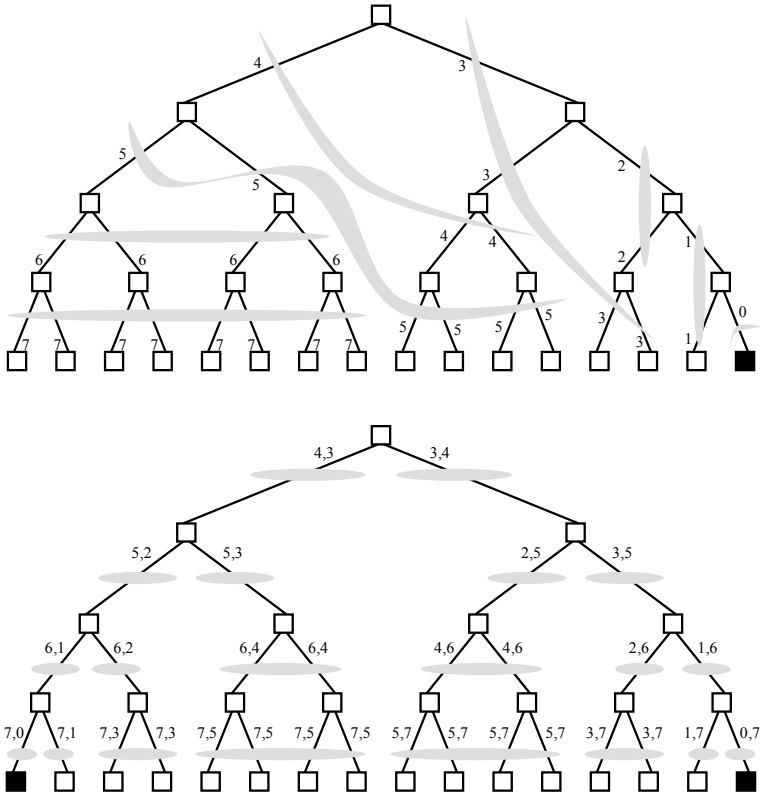

Figure 7-8: Cluster resolution on HSRA when using only one (top) and two (bottom) sink terminals in the sub-sample set S. The logic units shown in black represent the sink terminals in the sub-sample set S.

7.4.2 Experiment 2

Experiment 2 studies the effect of the number of clusters (K) on the quality of clustering solutions. We use a sub-sample value of 6% for island-style architectures, and 14% for HSRA. Figure 7-9 shows the effect of K on routing runtime. The x-axis shows the value of K as a fraction of the number of logic units in the target device, and the y-axis shows routing runtime in seconds per routing iteration. The charts in Figure 7-9 show that a value of K equal to or greater than the number of logic units in the target device produces clustering solutions of qualities similar (less than 10%) to heuristic estimates.

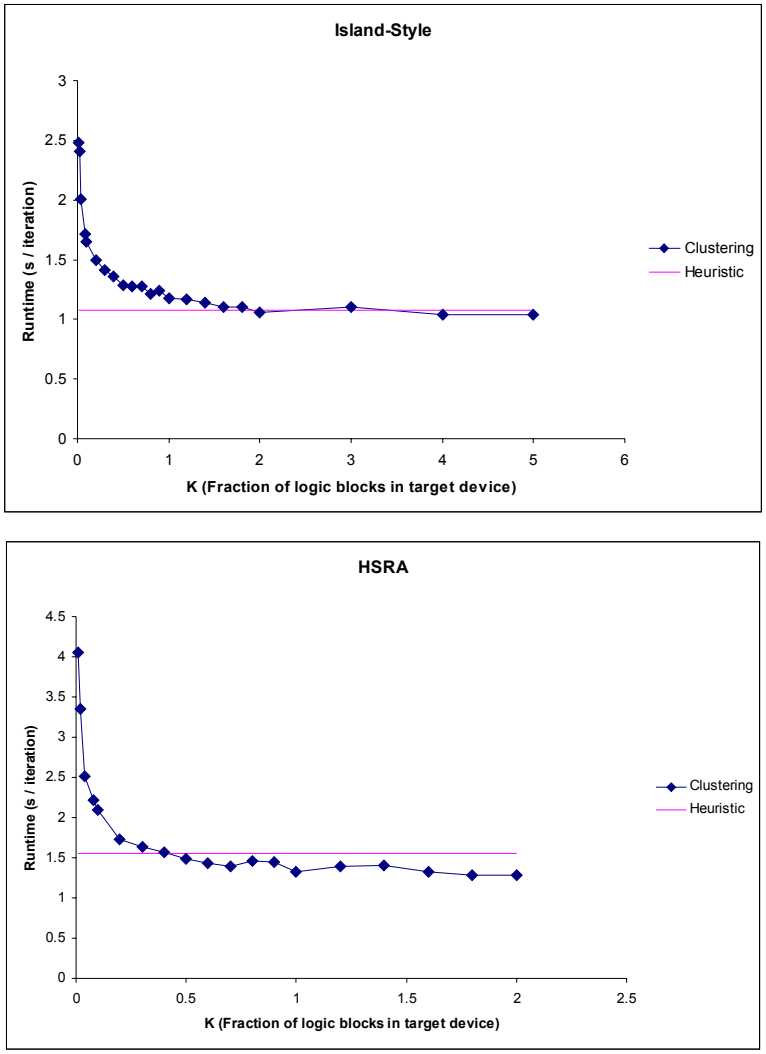

Figure 7-9: The effect of K on routing runtime.

7.4.3 Experiment 3

Our final experiment is a quantitative comparison of the quality of the A* estimates produced by our clustering technique vs. heuristically calculated estimates. We use a sink sub-sample value of 6%, and the value of K is equal to the number of logic units in the architecture. Table 7-2 shows the results we obtained on an island-style interconnect structure. Column 1 lists the netlist, column 2 lists the size of the smallest square array needed to just fit the netlist, column 3 lists the sum of the logic blocks and IO blocks in the netlist, and column 4 lists the number of nets in the netlist. Columns 5, 6, and 7 list routing runtimes for clustering-based A* estimates, heuristic estimates, and undirected (no A* estimates) search. Column 8 lists routing runtimes obtained clustering the interconnect wires directly with the

logic units in the target device (Section 7.2). In this "low-effort" clustering step, we run only the first iteration of K-Means clustering with K equal to the number of logic units in the target device. Further, the initial seeds are chosen to be logic unit outputs. Running K-Means with these parameters has the effect of associating each interconnect wire with its closest logic unit. The entries in columns 5, 6, 7 and 8 are normalized to the routing runtimes produced by heuristic estimates.

Table 7-2: A comparison of routing runtimes on an island-style architecture. For each netlist, the entries in columns 5 (clustering-based estimates), 7 (undirected search) and 8 (cluster with logic blocks) are normalized to the entry in column 6 (heuristic estimates).

Netlist	Size	N_{Blocks}	Nets	Clus	Heur	no A*	assocLB
term1	6x6	74	144	1.44	1.00	4.22	0.89
s1423	6x6	51	165	1.57	1.00	3.86	1.57
i9	7x7	195	214	1.30	1.00	3.40	1.30
dalu	8x8	154	312	0.93	1.00	4.04	0.93
vda	9x9	122	337	1.08	1.00	4.78	1.20
x1	10x10	181	352	0.94	1.00	4.66	1.13
rot	8x8	299	407	1.11	1.00	3.32	0.95
pair	9x9	380	512	0.94	1.00	4.83	0.89
apex1	11x11	192	566	0.96	1.00	6.03	0.97
dsip	14x14	598	762	1.06	1.00	8.21	1.13
ex5p	12x12	210	767	1.12	1.00	7.30	1.03
s298	16x16	253	767	1.37	1.00	10.38	1.58
tseng	12x12	307	780	1.07	1.00	6.30	1.05
alu4	14x14	215	792	1.14	1.00	7.48	1.09
misex3	14x14	207	834	1.08	1.00	9.80	1.16
apex4	13x13	193	869	1.02	1.00	5.04	1.10
diffeq	14x14	292	1033	1.13	1.00	5.29	1.19
bigkey	15x15	640	1040	1.18	1.00	8.95	1.38
seq	15x15	297	1055	1.10	1.00	7.22	1.19
des	15x15	701	1178	1.17	1.00	4.35	1.20
apex2	16x16	281	1249	1.09	1.00	8.19	1.08
frisc	22x22	582	2022	1.02	1.00	8.56	1.08
elliptic	22x22	699	2247	1.00	1.00	10.73	1.23
ex1010	25x25	621	3129	1.15	1.00	9.66	0.92
s38584.1	29x29	1148	4174	1.20	1.00	17.07	1.07
clma	33x33	1199	5269	1.02	1.00	15.25	1.03
				1.11	**1.00**	**6.59**	**1.12**

Across the set of benchmarks, our clustering-based technique is approximately 11% slower than the runtimes achieved by heuristically estimating A* costs. Both heuristic and clustering-based estimates are approximately 6.6X faster than an undirected search-based router. Finally, the geometric average of the estimates produced by K-Means clustering (column 5) is almost identical to the estimates produced by a "low-effort" clustering step (column 8). The near identical averages show that the associate-with-

closest-logic-unit approach presented in Section 7.2 works as well as a more sophisticated clustering approach on an island-style architecture.

Table 7-3 shows the results we obtained on HSRA. Column 1 lists the netlist, column 2 lists the number of logic units in the target device, column 3 lists the number of 4-LUTs in the netlist, and column 4 lists the number of nets in the netlist. Columns 5, 6, 7 and 8 list routing runtimes for clustering-based A* estimates, heuristic estimates, undirected (no A* estimates) search, and A* estimates based on clustering the interconnect wires directly with the logic units in the target device (Section 7.2). The entries in columns 5, 6, 7 and 8 are normalized to the routing runtimes produced by heuristic estimates. Across the set of benchmarks, our clustering-based technique is approximately 7% faster than the runtimes achieved by heuristically estimating A* costs. Both heuristic and clustering-based estimates are approximately ten times faster than an undirected search-based router. The estimates produced by a low-effort clustering step (column 8) are approximately 16% slower than the estimates produced by K-Means clustering. This is consistent with our intuition (Figure 7-3) that associating interconnect wires with logic units in a hierarchical structure will probably produce cost-to-target underestimates.

Table 7-3: A comparison of routing runtimes on HSRA. For each netlist, the entries in columns 5 (clustering-based estimates), 7 (undirected search), and 8 (cluster with logic blocks) are normalized to the entry in column 6 (heuristic estimates).

Netlist	Size	N_{LUTs}	Nets	Clus	Heur	no A*	assocLB
mm9b	256	120	133	1.16	1.00	3.87	1.48
cse	256	134	141	1.03	1.00	4.39	1.22
s1423	256	162	180	0.92	1.00	5.23	1.00
9sym	512	177	186	0.81	1.00	15.42	1.20
ttt2	256	198	222	1.06	1.00	13.58	1.25
keyb	256	209	216	1.16	1.00	4.25	1.16
clip	512	243	252	1.02	1.00	21.38	1.14
term1	512	246	280	0.83	1.00	19.56	1.11
apex6	1024	258	393	1.24	1.00	6.53	1.26
vg2	512	277	302	0.96	1.00	16.81	1.16
frg1	1024	282	310	0.81	1.00	26.73	0.85
sbc	1024	332	373	0.87	1.00	12.41	1.13
styr	1024	341	350	0.83	1.00	13.60	1.06
i9	512	347	435	1.01	1.00	12.12	1.32
C3540	1024	382	432	0.79	1.00	5.89	0.79
sand	1024	406	417	0.80	1.00	10.67	0.88
x3	1024	441	576	0.80	1.00	3.60	0.88
planet	2048	410	417	0.81	1.00	13.67	1.14
rd84	2048	405	413	1.09	1.00	21.04	1.08
dalu	2048	502	577	0.82	1.00	16.62	0.84
				0.93	**1.00**	**10.39**	**1.08**

7.4.4 Experiment 4

Our final experiment quantifies the difference between the cost-to-target estimates produced by our clustering technique and actual shortest-path costs. We use a sink sub-sample value of 6%, and the value of K is equal to the number of logic units in the architecture.

Table 7-4: The difference between cost-to-target estimates and actual shortest-path costs on the island-style architecture

Netlist	Size	Fraction Sinks Underestimated	Avg Cost-to-Target Underestimate
term1	6x6	0.39	0.26
s1423	6x6	0.41	0.26
i9	7x7	0.41	0.24
dalu	8x8	0.37	0.24
vda	9x9	0.31	0.23
x1	10x10	0.35	0.22
rot	8x8	0.43	0.24
pair	9x9	0.35	0.23
apex1	11x11	0.36	0.22
dsip	14x14	0.33	0.19
ex5p	12x12	0.32	0.21
s298	16x16	0.32	0.19
tseng	12x12	0.35	0.2
alu4	14x14	0.32	0.19
misex3	14x14	0.35	0.19
apex4	13x13	0.28	0.2
diffeq	14x14	0.32	0.19
bigkey	15x15	0.34	0.19
seq	15x15	0.34	0.19
des	15x15	0.33	0.19
apex2	16x16	0.3	0.18
frisc	22x22	0.33	0.16
elliptic	22x22	0.33	0.16
ex1010	25x25	0.3	0.15
s38584.1	29x29	0.3	0.14
clma	33x33	0.29	0.13
Geomean		**0.34**	**0.20**

Table 7-4 shows the results we obtained on the island-style architecture. Column 1 lists the benchmark netlists and column 2 lists the size of the target device. Column 3 lists the fraction of the total number of sinks in the target device that are underestimated. For example, the 0.39 entry for the netlist term1 means that – on average – the cost-to-target estimates at a routing wire in the target device are less than the cost of a shortest path from the wire to 39% of the sinks. Column 4 quantifies the actual difference in costs between the cost-to-target estimates and actual shortest paths. The 0.26 entry for term1 means

that the difference between the cost-to-target estimates at a routing wire and actual shortest path costs is 26% when averaged across all underestimated sinks. Overall, our clustering technique underestimates 34% of the sinks in a target device, and the difference between estimated costs and actual shortest path costs is 20%. Similarly, on HSRA (Table 7-5) our clustering technique underestimates 9% of the sinks in a target device, and the difference between estimated costs and actual shortest paths is 23%.

Table 7-5: The difference between cost-to-target estimates and actual shortest-path costs on HSRA.

Netlist	Size	Fraction Sinks Underestimated	Avg Cost-to-Target Underestimate
mm9b	256	0.29	0.2
cse	256	0.25	0.18
s1423	256	0.25	0.19
9sym	512	0.1	0.27
ttt2	256	0.23	0.18
keyb	256	0.24	0.18
clip	512	0.08	0.28
term1	512	0.14	0.21
apex6	1024	0.05	0.29
vg2	512	0.07	0.28
frg1	1024	0.06	0.24
sbc	1024	0.07	0.23
styr	1024	0.05	0.27
i9	512	0.13	0.18
C3540	1024	0.05	0.27
sand	1024	0.05	0.22
x3	1024	0.06	0.23
planet	2048	0.04	0.25
rd84	2048	0.03	0.27
dalu	2048	0.03	0.27
Geomean		**0.09**	**0.23**

7.5 Summary

The results of our experiments show that K-Means clustering produces A* estimates that are comparable to architecture-specific heuristic estimates. A sink sub-sample value of 6%, coupled with a value of K that is equal to the number of logic units in the target device, produces estimates that are 7% better than heuristically calculated estimates for HSRA, and within 11% of heuristic estimates for island-style interconnect structures. *Experiment 1* also shows that a small number of sink terminals might be sufficient to produce estimates that are comparable to heuristic estimates. Finally, the quality of the clustering solutions produced by a low-effort clustering step is surprisingly good when compared to a more sophisticated K-Means clustering approach. All in all, we feel that either the low-effort or

more sophisticated clustering approach presented in this section may be a compelling candidate for producing A* estimates of good quality across different interconnect styles.

Chapter 8: Pipelined FPGA Architecture

Current commercial offerings from FPGA vendors like Xilinx [59,60] and Altera [2,3] provide a wide range of capabilities. Recall from Chapter 2 that Altera's StratixII device has dedicated arithmetic circuitry, embedded DSP blocks that can be configured to implement high-speed multipliers and filters, and distributed RAM blocks. These custom-designed features can be coupled with the LUT-based logic structure and rich routing fabric to implement entire systems at a time. However, improvements in FPGA clock cycle times have consistently lagged behind advances in device functionality and capacities. Even the simplest circuits cannot be clocked at more than a few hundred megahertz.

In the world of microprocessors and ASICs, pipelining is widely used to reduce the clock-cycle time of a netlist. Pipelining is the process of inserting registers in a netlist so that the maximum combinational path delay between successive register stages is less than that of the original unpipelined netlist. If the computation implemented by a netlist can be divided into 'N' successive stages, then registers are inserted at stage boundaries. In this manner, the clock cycle time of a netlist may potentially be reduced by a factor of N. However, the latency (or total execution time) of the computation increases due to imbalanced path delays and register setup times.

Sequential Retiming [27] is a powerful heuristic that can be used to pipeline a circuit while preserving its functionality. The most basic retiming operation involves moving registers from the inputs of a gate to its output(s), or vice versa. Consider the example shown in Figure 8-1. Assuming a unit gate-delay model, the circuit on top has a critical path delay of three units. Moving the registers across the leftmost AND gate reduces the delay to two units. The delay is further reduced to one unit when the registers at the output of the rightmost AND gate are moved to its inputs.

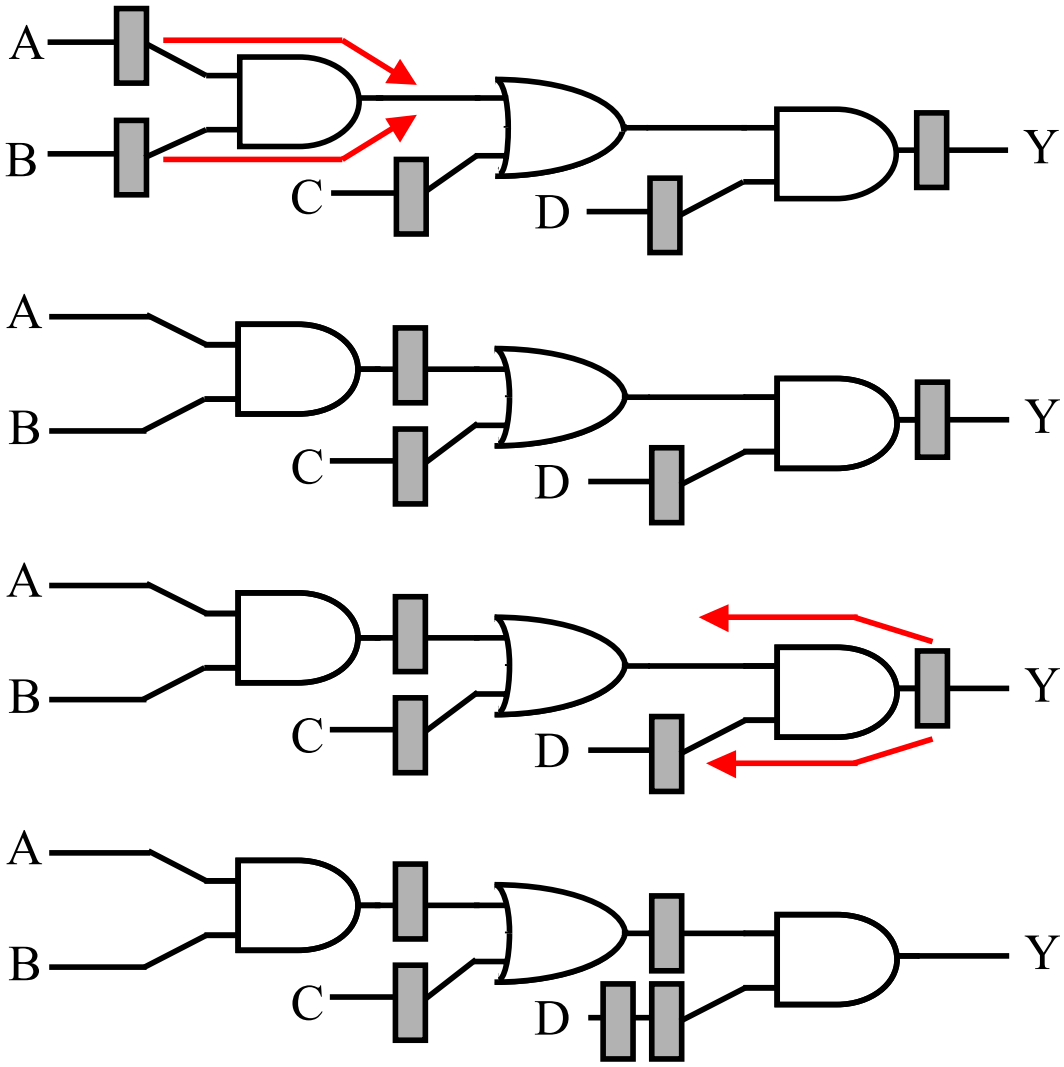

Figure 8-1: The circuit on top has a critical path delay of 3 units. A retiming operation (red arrows) moves registers from the inputs of the leftmost AND gate to its output. The critical path delay is reduced to 2 units. A second operation moves the register from the output of the rightmost AND gate to its inputs, further reducing the critical path to 1 unit. Note that the latency of the circuit remains unchanged.

An important limitation of classical retiming is that the number of registers around a cycle cannot be changed during the retiming process. While it is possible to move the registers around within the cycle, no new registers can be included. In [36], the authors proved that the minimum achievable clock period for a netlist that includes such cycles is at least the maximum average delay of a cycle. The average delay is defined as the total combinational path delay of the cycle divided by the number of registers on the cycle.

An extension of classical retiming called C-slow retiming is used to increase the number of registers around a cycle. In C-slow retiming, 'C' registers replace each register in the netlist, and the netlist is then retimed. Since there are C times as many registers available on a cycle, the minimum achievable clock period may be significantly reduced. However, C-slow retiming alters the functionality of a circuit, with a new output available only every C clock cycles. Consequently, C-slow retiming offers real benefits only when it is possible to multiplex the data from 'C' independent data sets at the rate of a new input datum every clock cycle.

8.1 Pipelined FPGA Architectures

Different research groups have tried to improve clock cycle times by proposing pipelined FPGA architectures. The main distinguishing feature of a pipelined FPGA is the possible location of registers in the architecture. To support pipelining and retiming, pipelined FPGAs provide pre-fabricated registers in both the logic and interconnect structures. Applications mapped to pipelined FPGAs are often retimed to take advantage of a relatively large number of registers in the logic and interconnect structures.

In this section, we briefly survey pipelined FPGA architectures and the heuristics used to allocate pipelining registers during the place-and-route phase. Our goal is to focus on the architectural and algorithmic support provided by FPGA architects in order to support pipelined and retimed application netlists.

8.1.1 Coarse-grained Architectures

RaPiD [15] is one example of a coarse-grained pipelined architecture. The RaPiD architecture is targeted to high-throughput, compute-intensive applications like those found in DSP. Since such applications are generally pipelined, the RaPiD datapath and interconnect structures include an abundance of registers. The 1-Dimensional (1-D) RaPiD datapath (Figure 8-2) consists of coarse-grained logic units such as ALUs, multipliers, small SRAM blocks, and general purpose registers (hereafter abbreviated GPRs). Each logic unit is 16 bits wide. To support pipelining in the logic structure, a register bank is provided at each output of a logic unit. The output register bank can be used to acquire between 0 – 3 registers.

The interconnect structure consists of 1-D routing tracks that are also 16 bits wide. There are two types of routing tracks: short tracks and long tracks. Short tracks are used to achieve local connectivity between logic units, whereas long tracks traverse longer distances along the datapath. In Figure 8-2, the

uppermost five tracks are short tracks, while the remaining tracks are long tracks. A separate routing multiplexer is used to select the track that drives each input of a logic unit. Each output of a logic unit can be configured to drive multiple tracks by means of a routing demultiplexer.

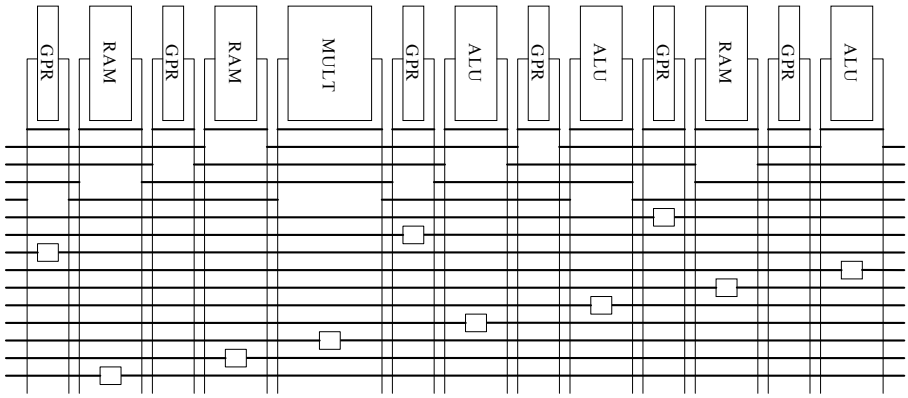

Figure 8-2: An example of a RaPiD architecture cell. Several RaPiD cells can be tiled together to create a representative architecture.

The long tracks in the RaPiD interconnect structure are segmented by means of bus connectors (shown as empty boxes in Figure 8-2 and abbreviated BCs). BCs serve two roles in the RaPiD interconnect structure. First, a BC serves as a buffered, bidirectional switch that facilitates the connection between two long-track segments. Second, a BC serves the role of an interconnect register site. RaPiD provides the option of picking up between 0 – 3 registers at each BC. The total number of BCs determines the number of registers that can be acquired in the interconnect structure.

In the next chapter, we describe heuristics that can be used to efficiently allocate pipelining registers when mapping applications to a RaPiD array.

8.1.2 Fixed-frequency FPGA Architectures

HSRA [52] and SFRA [56] are two examples of fixed-frequency architectures that guarantee the execution of an application at a fixed clock frequency. HSRA has a strictly hierarchical, tree-like routing structure (Figure 8-3), while SFRA (Figure 8-4) has a capacity depopulated island style routing structure. Applications mapped to HSRA and SFRA are aggressively C-slowed to reduce clock cycle times. An important consequence of C-slowing is that the register count in a netlist increases by a factor of C. Since an FPGA has limited resources, finding pipelining registers in the interconnect and/or logic structure without adversely affecting the routability and delay of a netlist is a difficult problem. Both

HSRA and SFRA circumvent this problem by providing deep retiming register-banks at the inputs of logic units, as well as registered switch-points.

Since fixed-frequency FPGAs provide register-rich logic and routing structures, there is no need to efficiently locate pipelining registers during placement and routing. Consequently, the place & route flows developed for SFRA and HSRA are unaware of pipelining registers. However, the area overhead incurred by these architectures due to their heavily pipelined structure is high. HSRA incurs approximately a 2X area overhead and a 5X latency overhead, and SFRA takes a 4X area hit. While these overheads might be justifiable for certain classes of applications (those that are amenable to C-slowing and/or heavy pipelining), they might be prohibitive for conventional, general-purpose FPGAs.

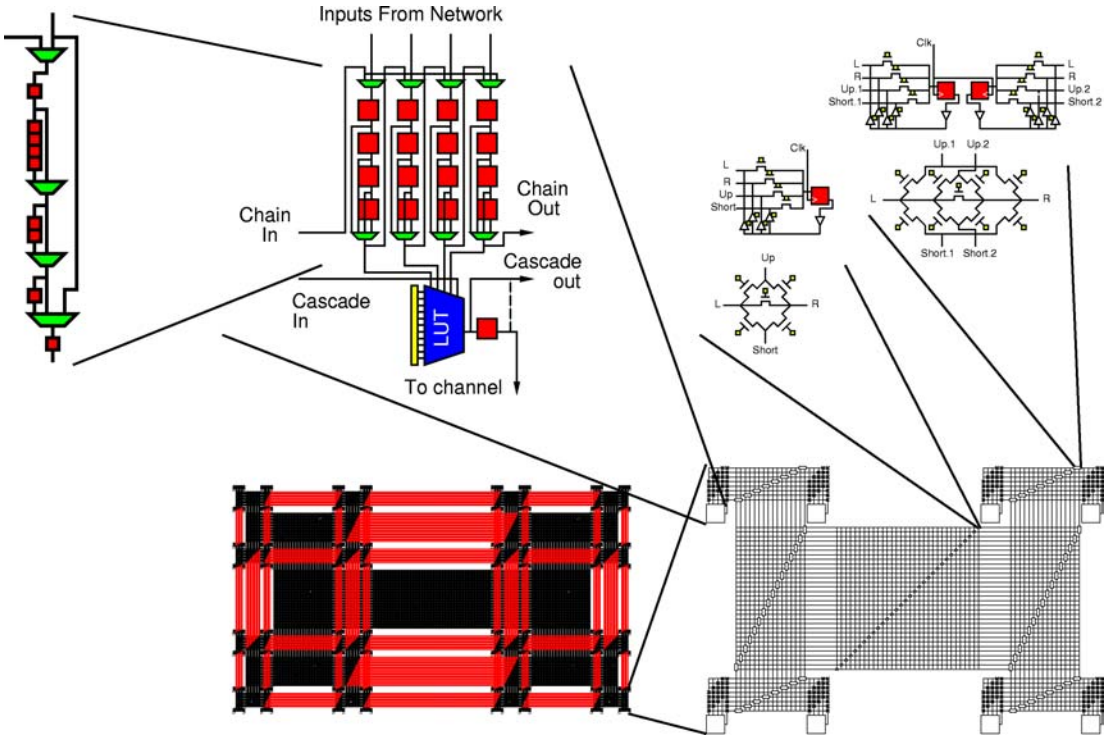

Figure 8-3: [52] The HSRA architecture. Each logic unit consists of a single 4-LUT. There is a retiming register chain provided at the inputs of the logic unit (top left), and a single register at the output of the logic unit. Registers are also provided in each switch in the interconnect structure (top right).

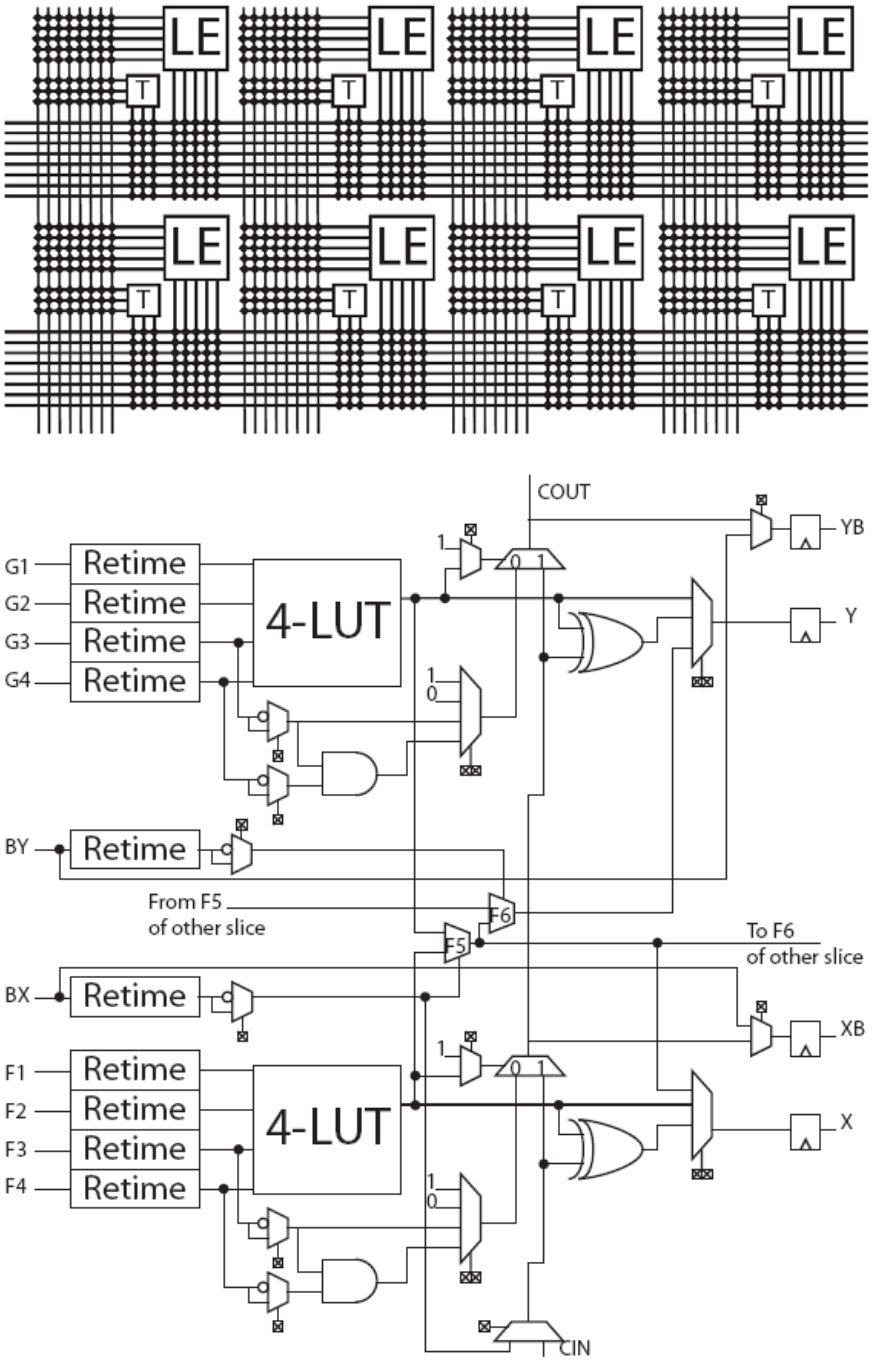

Figure 8-4: [56] The SFRA architecture. The interconnect structure (top) consists of capacity-depopulated corner turn switchboxes. Bidirectional pipelining registers are provided in the corner-turn switchboxes. Each logic unit (called an LE) consists of two slices. The structure of a slice (bottom) is similar to the slice of a Virtex [60] device. Note the retiming banks at the inputs of the slice.

8.1.3 General-purpose FPGA Architectures

A number of researchers have attempted to integrate pipelining with place and route flows for general-purpose FPGAs. An early effort (circa 1995) that made use of fine-grain pipelining to achieve a pre-determined target clock frequency is presented in [54]. In this work, the author achieved an operating frequency of 250 MHz by carefully hand mapping a DSP block to a Xilinx XC3000 series device. The hand-mapping technique was centered on the concept of an "event horizon". Given a target clock frequency, an event horizon defined the maximum distance a signal could travel in the interconnect structure in a single clock cycle. Von Herzen used a hand-tuned constructive placement approach to build circuits from the middle of the circuit out.

Since Von Herzen's early work, many researchers have attempted to integrate retiming with automatic place-and-route toolflows. The techniques presented in [39] use post place-and-route delay information to accurately retime netlists mapped to the Virtex family of FPGAs. To preserve the accuracy of delay information, the authors do not attempt to re-place-and-route the netlist after the completion of the retiming operation. Since the logic units (called 'slices') in Virtex devices have a single output register, the edges in the retiming graph are constrained to allow no more than a single register. This constraint might be overly restrictive, especially for applications that might benefit from a more aggressive retiming approach like C-slowing.

In [57], a post-placement C-slow retiming technique for the Virtex family of FPGAs is presented. Since C-slowing increases the register count of a netlist by a factor of C, a post-retiming heuristic is used to place registers in unused logic units. A search for unused logic units is begun at the center of the bounding box of a net. The search continues to spiral outward until an unused register location is found. When an unused register location is found, it is allocated. This process is repeated until all retiming registers have been placed. A significant shortcoming of this heuristic is its dependence on the pre-retiming placement of a netlist. If the placement of the netlist is dense, then the heuristic may not be able to find unused register locations within the net's bounding box. Instead, unused locations that are far removed from the net's terminals may be allocated. The resultant routes between the net's terminals and the newly allocated registers might become critical and thus destroy the benefits of C-slowing.

An alternative approach to locate post-placement retiming registers is presented in [47]. The newly created registers are initially placed into preferred logic units even if the logic units are occupied. A greedy iterative improvement technique then tries to resolve illegal overlaps by moving non-critical logic blocks to make space for registers. The cost of a placement is determined by the cumulative

illegality of the placement, overall timing cost, and wirelength cost. The timing cost is used to prevent moves that would increase critical path delay, while the wirelength cost is used to estimate the routability of a placement.

The retiming-aware techniques for general-purpose FPGAs presented so far use heuristics to place retiming registers in the logic units of the FPGA. An alternative to placing registers in the logic structure of a general-purpose FPGA is to allocate the registers in the routing structure. In [46], the authors propose a routing algorithm that attempts to move long (and hence critical) routes onto tracks that have registered routing switches. The algorithm exploits the planarity of the target architecture to permute the routes on a registered / unregistered track with those on a compatible unregistered / registered track. An architecturally constrained retiming algorithm is coupled with the routing step to identify tracks that are used by critical routes. All routes on a given critical track are then permuted with a compatible registered track, so that critical routes can go through registered routing switches. After the completion of retiming-aware routing, a final retiming step is performed to achieve a target clock period.

There are two important shortcomings of the retiming-aware routing algorithm presented in [46]. First, the process of permuting routes on to registered tracks may be overly restrictive, since all routes on a registered track must go through registered routing switches. While long routes may benefit from an assignment to a registered routing track, other less-critical routes on the track will use up registered switches unnecessarily. Second, the routing algorithm relies on planar FPGA architectures to enable track permutation. Consequently, the algorithm cannot be used to route netlists on non-planar FPGA architectures that have registered routing switches.

In summary, it is clear from our discussion on fixed-frequency architectures that the area overhead incurred in eliminating the problem of locating pipelining registers is high. At the same time, the heuristic techniques used to allocate pipelining registers in general purpose pipelined FPGAs are architecture-specific solutions that may not be applicable to a range of architecturally diverse pipelined FPGAs.

The next chapter describes the development of a pipelining-aware router (called PipeRoute) for the RaPiD architecture. When developing the PipeRoute algorithm, we tried to minimize reliance on RaPiD-specific features by using architecture independent abstractions and heuristics. Specifically, we represented RaPiD's interconnect structure as a routing graph, which allowed us to leverage

Pathfinder's congestion resolution mechanism. Furthermore, since PipeRoute's heuristics operate on a routing graph, the algorithm could potentially be used to route netlists on other pipelined FPGA architectures.

Chapter 9: PipeRoute – A Pipelining-Aware Router for FPGAs

The traditional FPGA routing problem is to determine an assignment of signals to limited routing resources while trying to achieve the best possible delay characteristics. In the case of pipelined netlists, the routing problem is different from the conventional FPGA routing problem. This is because a significant fraction of the signals in a netlist are deeply pipelined, and merely building a Minimum Spanning Tree (MST) for a pipelined signal is not enough. For example, consider the pipelined signal sig in Fig. 1 that has a source S and sinks K1, K2 and K3. The signal is pipelined in such a way that sink K1 must be delayed 3 clock cycles relative to S, sink K2 must be 4 clock cycles away, and sink K3 must be 5 clock cycles away. A route for sig is valid only if it contains enough pipelining resources to satisfy the clock cycle constraints at every sink. Due to the fact that there are a fixed number of sites in the interconnect where a signal can go through a register, it can be easily seen that a route found for sig by a conventional, pipelining-unaware FPGA router may not go through sufficient registers to satisfy the clock cycle constraint at every sink. Thus, the routing problem for pipelined signals is different from that for unpipelined signals.

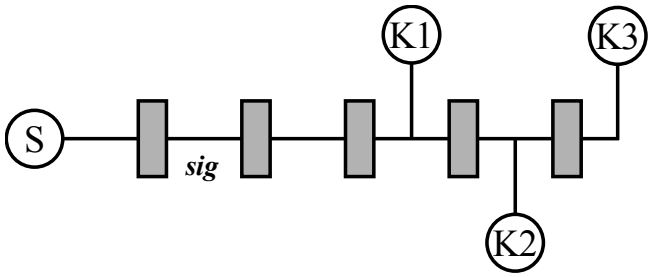

Figure 9-1: A multi-terminal pipelined signal. The register separation between S and the sinks K1, K2, K3 must be three, four and five respectively.

For a two-terminal pipelined signal, the routing problem is stated as:

Two-terminal N_D Problem: *Let G=(V,E) be an undirected graph, with the cost of each node v in the graph being $w_v >= 1$. The graph consists of two types of nodes: D-nodes and R-nodes. Let S, K∈V be two R-nodes. Find a path $P_G(S,K)$ that connects nodes S and K, and contains at least N ($N \geq 1$) distinct*

D-nodes, such that $w(P_G(S,K))$ is minimum, where

$$w(P_G(S,K)) = \sum_{v \in P_G(S,K)} w_v$$

Further, impose the restriction that the path cannot use the same edge to both enter and exit any D-node.

We call a route that contains at least 'N' distinct D-nodes an 'N_D' route. R-nodes represent interconnect wire-segments and the IO pins of logic units in a pipelined FPGA architecture, while D-nodes represent registered switch-points. A registered switch-point (from this point on, we will use the terms 'registered switch-points' and 'registers' interchangeably) can be used to pick up 1 clock cycle delay, or no delay at all. Every node is assigned a cost, and an edge between two nodes represents a physical connection between them in the architecture. The cost of a node is a function of congestion, and is identical to the cost function developed for Pathfinder's NC algorithm. Under this framework, the routing problem for a simpler two-terminal signal is to find the lowest cost route between source and sink that goes through at least N ($N \geq 1$) distinct D-nodes (N is the number of clock cycles that separates the source from the sink). Note that in this version a lowest cost route can be self-intersecting i.e. R-nodes can be shared in the lowest cost route. In [42], we show that the two terminal N_D problem is NP-Complete via a reduction from the Traveling Salesman Problem with Triangle Inequality.

9.1 One-Delay (1_D) Router

Although the general two terminal N_D problem is NP-Complete, we now show that a lowest cost route between a source and sink that goes through at least one D-node can be found in polynomial time. On a weighted undirected graph, Dijkstra's algorithm is widely used to find the lowest cost route between a source and sink node. The remainder of this section evaluates several modifications of Dijkstra's algorithm that can be used to find a lowest cost 1_D route. Our first modification is *Redundant-Phased-Dijkstra*. In this algorithm, a phase 0 wavefront is launched at the source. When the phase 0 exploration hits a D-node, it is locally terminated there (i.e. the phase 0 exploration is not allowed to continue through the D-node, although the phase 0 exploration can continue through other R-nodes and runs simultaneously with the phase 1 search), and an independent phase 1 wavefront is begun instead. When commencing a phase 1 wavefront at a D-node, we impose a restriction that disallows the phase 1 wavefront from exiting the D-node along the same edge that was used to explore it at phase 0. This is based on the assumption that it is architecturally infeasible for the D-node that originates the phase 1 wavefront to explore the very node that is used to discover it at phase 0. When a phase 1 wavefront explores a D-node, the D-node is treated like an R-node, and the phase 1 wavefront propagates through the D-node.

If the number of D-nodes that can be explored at phase 0 from the source is 'F', up to F independent phase 1 wavefronts can co-exist during *Redundant-Phased-Dijkstra*. The search space of the phase 1 wavefronts can overlap considerably due to the fact that each R-node in the graph can be potentially explored by up to F independent phase 1 wavefronts. Consequently, the worst-case run-time of *Redundant-Phased-Dijkstra* is F+1 times that of the conventional Dijkstra's algorithm. Since F could potentially equal the total number of interconnect registers in a pipelined FPGA, the worst-case run-time of *Redundant-Phased-Dijkstra* may get prohibitive.

An alternative to *Redundant-Phased-Dijkstra* that can be used to find a lowest cost 1_D route is *Combined-Phased-Dijkstra*. This algorithm attempts to reduce run-time by combining the search space of the phase 1 wavefronts that originate at D-nodes. The only difference between *Redundant-Phased-Dijkstra* and *Combined-Phased-Dijkstra* is that the latter algorithm allows each R-node to be visited only once by a phase 1 wavefront. As a consequence, the run-time of *Combined-Phased-Dijkstra* is only double that of Dijkstra's algorithm. In both *Redundant-Phased-Dijkstra* and *Combined-Phased-Dijkstra*, the phase 1 search begins at a cost equal to the path up to the D-node that starts the wavefront. The final route is found in two steps. In the first step, the phase 1 segment of the route is found by backtracing the phase 1 wavefront to the D-node that initiated the wavefront. The phase 0 segment of the route is then found by backtracing the phase 0 wavefront from the D-node back to the source.

A step-by-step illustration of how *Combined-Phased-Dijkstra* works is shown in Figure 9-2. For the sake of simplicity, assume all nodes in the example graph have unit cost. The source S is explored at phase 0 at the start of the phased exploration. The number 0 next to S in Figure 9-2 (a) indicates that S has been explored by a phase 0 wavefront. In Figure 9-2 (b), the neighbors of S are explored by the phase 0 wavefront initiated at S. The 2^{nd}-level neighbors of S are explored by phase 0 in Figure 9-2 (c), one of which is D-node D1. Note that we make a special note of D1's phase 0 predecessor here, so that we do not explore this predecessor by means of the phase 1 wavefront that is commenced at D1. In Figure 9-2 (d), the neighbors of D1 (excluding R1) are explored at phase 1. The phase 0 exploration also continues simultaneously, and note how both phase 0 and phase 1 wavefronts have explored nodes R4 and R7. Finally, in Figure 9-2 (e), the sink K is explored by the phase 1 wavefront initiated at D1. The route found by *Combined-Phased-Dijkstra* is shown in boldface in Figure 9-2 (e), and is in fact an optimal route between S and K.

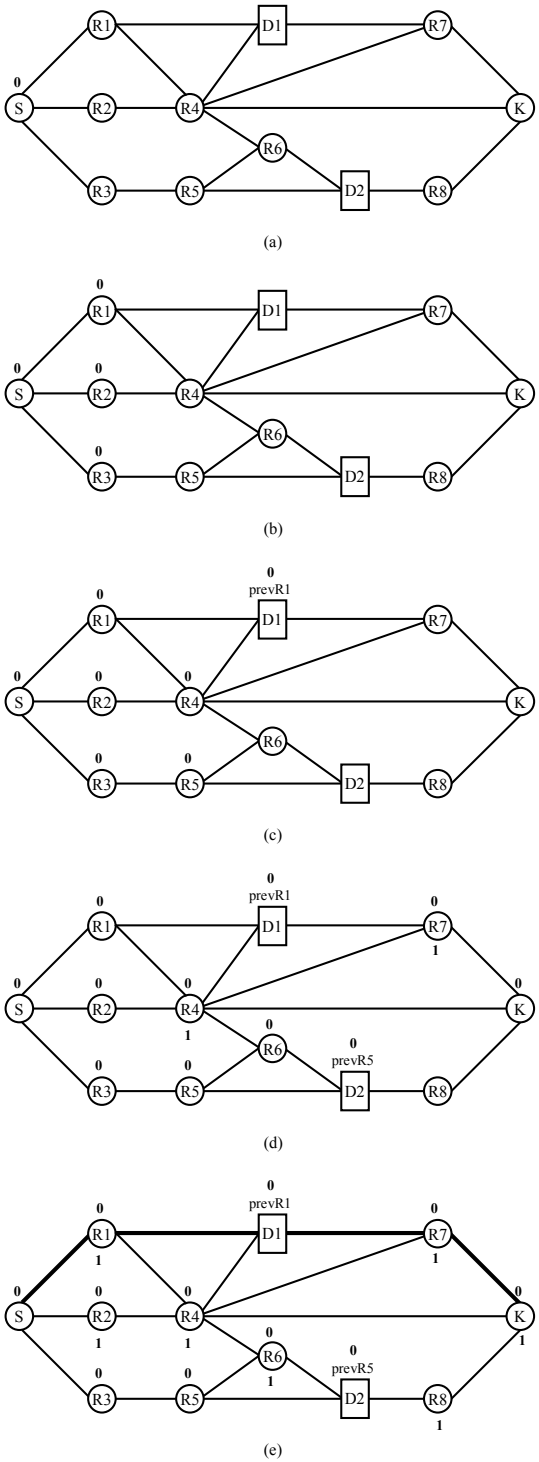

Figure 9-2: A step-by-step illustration of Combined-Phased-Dijkstra.

Unfortunately, *Combined-Phased-Dijkstra* fails to find a lowest cost route on some graph topologies. An example of a failure case is shown in Figure 9-3. Here the node S is both the source and sink of a signal, and each node is unit cost. *Combined-Phased-Dijkstra* will fail to return to S at phase 1 because R-nodes on each possible route back to S have already been explored by the phase 1 wavefront. In effect, *Combined-Phased-Dijkstra* isolates nodes S, R1, R2, D1 and D2 from the rest of the graph, thus precluding the discovery of any route back to S at all.

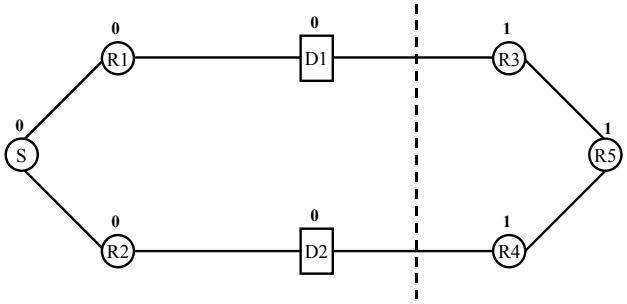

Figure 9-3: A case in which phased exploration fails. Observe how the phase 1 exploration has got isolated from the phase 0 exploration.

The reason for the failure of *Combined-Phased-Dijkstra* is that a node on the phase 1 segment of the lowest cost route is instead explored by a phase 1 wavefront commenced at another D-node. For example, in Figure 9-3 we consider the route S-R1-D1-R3-R5-R4-D2-R2-S to be lowest cost. Node R4 is explored by the phase 1 wavefront commenced at D2, thus precluding node R4 from being explored by the phase 1 wavefront started at D1. However, if we slightly relax *Combined-Phased-Dijkstra* to allow each node in the graph to be explored by at most two phase 1 wavefronts that are independently started at different D-nodes, then the phase 1 wavefronts started at D1 and D2 will now be able to overlap, thus allowing the lowest cost route to be found.

An important consequence of the nature of the transition from phase 0 to phase 1 at a D-node is shown in Figure 9-4. In this case, S is the source of the signal, and K is the sink. Observe that a phase 0 exploration explores D1 from R1. Consequently, the phase 0 exploration is precluded from exploring D1 from R4. This prevents the optimal 1_D route to K from being found. To address this problem, we allow any D-node to be explored at most two times at phase 0. In Fig. 4, D1 can be explored at phase 0 from R1 and R4, thus allowing the optimal 1_D path S-R2-R3-R4-D1-R1-K to be found.

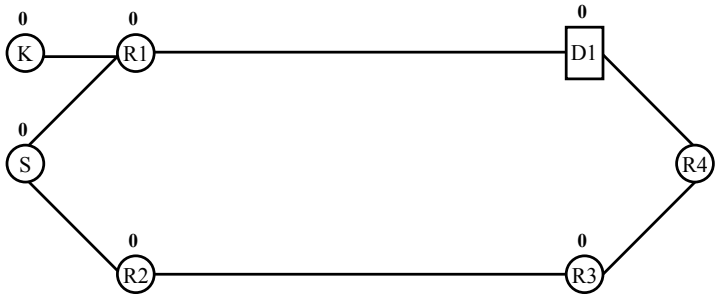

Figure 9-4: D1 is explored at phase 0 from R1, thus precluding the discovery of the 1_D path to the sink K.

Figure 9-5 shows pseudo code for the algorithm **2Combined-Phased-Dijkstra** that finds an optimal 1_D route between a source **S** and sink **K**. At the start of the algorithm, a phase 0 exploration is commenced at the source by initializing the priority queue **PQ** to **S** at phase 0. The phase 0 wavefront is expanded in a manner similar to that of Dijkstra's algorithm. Each time a node **lnode** is removed from **PQ**, its phase is recorded in the variable **phase**. The cost of the path from **S** to **lnode** is stored in **path_cost**. The variable **node_type** indicates whether **lnode** is an R-node or D-node. The fields **lnode.num_ex0** and **lnode.num_ex1** record the number of times **lnode** has been explored at phase 0 and 1 respectively, and are both initialized to 0. A node is marked *finally_explored* at a given phase when it is no longer possible to expand a wavefront through that node at the given phase. For each **lnode** that is removed from **PQ**, the following possibilities exist:

- **phase**==0 and **node_type** is R-node: R-nodes can be explored at phase 0 only once, and thus **lnode** is marked *finally_explored* if **x0**==1. The routine **AddNeighbors**(**PQ,lnode,path_cost,p**) is used to add the neighbors of **lnode** to **PQ** at phase **p**, where **p**==0 in this case.
- **phase**==0 and **node_type** is D-node: D-nodes can be explored at phase 0 twice, and thus **lnode** is marked *finally_explored* if **x0**==2. A phase 1 exploration is begun at this D-node by adding its neighbors to **PQ** at phase 1.
- **phase**==1: Since both R-nodes and D-nodes can be explored twice at phase 1, **lnode** is marked *finally_explored* at phase 1 if **x1**==2. If we are not done (i.e. **lnode** is not the sink **K**) the neighbors of **lnode** are added to **PQ** at phase 1.

```
2Combined-Phased-Dijkstra(S,K){
  Init PQ to S at phase 0;
  LOOP{
    Remove lowest cost node lnode from PQ;
    if(lnode == NULL){
      1_D path between S and K does not exist;
      return 0;
    }
    if(lnode is finally_explored at phase 0 and phase 1)
      continue;
    path_cost = cost of path from S to lnode;
    phase = phase of lnode;
    node_type = type of lnode;
    if(phase == 0){
      lnode.num_ex0++;
      x0 = lnode.num_ex0;
    }
    else{
      lnode.num_ex1++;
      x1 = lnode.num_ex1;
    }
    if(phase  == 0){
      if(node_type == R-node){
        if(x0 == 1)
          Mark lnode finally_explored at phase 0;
        AddNeighbors(PQ,lnode,path_cost,0);
      }
      else{
        if(x0 == 2)
          Mark lnode finally_explored at phase 0;
        AddNeighbors(PQ,lnode,path_cost,1);
      }
    }
    else{
      if(lnode == K)
        return backtraced 1_D path from S to K;
      else{
        if(x1 == 2)
          Mark lnode finally_explored at phase 1;
        AddNeighbors(PQ,lnode,path_cost,1);
      }
    }
  }END LOOP
}

AddNeighbors(PQ,lnode,path_cost,p){
  Foreach neighbor neb_node of lnode{
    neb_cost = cost of neb_node;
    neb_path_cost = neb_cost + path_cost;
    Add neb_node to PQ with phase p at cost neb_path_cost;
  }
}
```

Figure 9-5: Pseudo code for the 2Combined-Phased-Dijkstra algorithm.

9.1.1 Proof of Optimality

The optimality of `2Combined-Phased-Dijkstra` can be demonstrated by means of a proof by contradiction in which we show that `2Combined-Phased-Dijkstra` will always find an optimal 1_D path between S and K, if one exists. Before presenting a sketch of the proof, we introduce some terminology. `2Combined-Phased-Dijkstra` explores multiple paths through the graph via a modification to Dijkstra's algorithm. We state that the algorithm explores a path "P" up to a node "N" if the modified Dijkstra's search, in either phase 0 or phase 1, reaches node "N" and the search route to this node is identical to the portion of the path P from the source to node N. Further, a path A is "more explored" than path B if the cost of the path on A from the source to A's last explored point is greater than the cost of the path on B from the source to B's last explored point. For purposes of the proof sketch, we define the "goodness" of a path in the following way:

- If the cost of one path is lower than another's, it is "better" than the other. Thus, an optimal path is always better than a non-optimal path.
- If the costs of two paths C and D are the same, then C is "better" than D if C is more explored than D.

From these definitions, the "best" path is an optimal path. If there is more than one optimal path, the best path is the most explored optimal path.

***Initial Assumption*:** Assume that Figure 9-6 shows the *most explored* optimal 1_D path between S and K. In other words, the path shown in the figure is the best 1_D path between S and K, with a single clock-cycle delay picked up at D-node D_L. Note that there are no D-nodes on the path S-D_L, although there could be multiple D-nodes on D_L-K. This is because we assume that in case the best 1_D path between S and K goes through multiple D-nodes, then the D-node nearest S is used to pick up one clock-cycle delay.

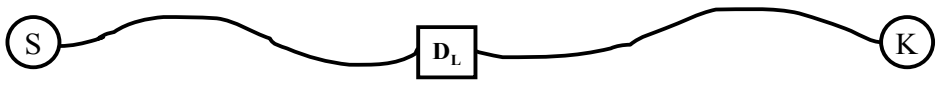

Figure 9-6: The initial assumption is that the most explored lowest cost 1_D route between S and K goes through D-node D_L.

Although it appears that the paths S-D_L and D_L-K in Figure 9-6 are non-intersecting, note that the R-nodes on the path S-D_L can in fact be reused in the path D_L-K. In all diagrams in this section, we use the convention of showing paths without overlaps (Figure 9-7), even though they may actually overlap

(Figure 9-8). Our proof does not rely on the extent of intersection between hypothetical paths (which are always shown in gray) and the known best 1_D path.

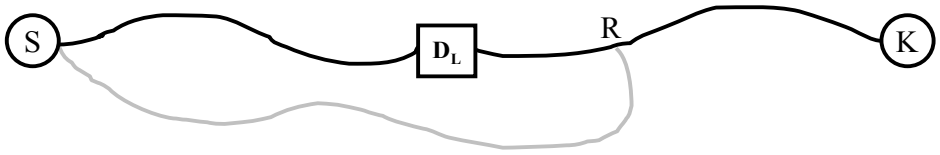

Figure 9-7: Representation of a path from S to node R shown in gray.

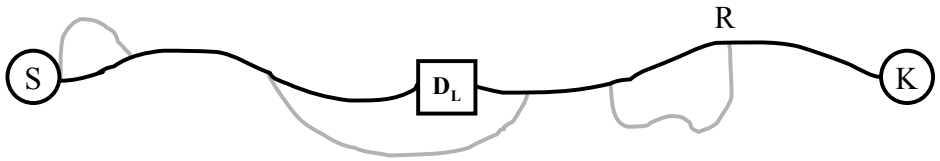

Figure 9-8: The path from S to R could actually intersect with the paths S-D_L and D_L-K.

There are three distinct cases in which `2Combined-Phased-Dijkstra` could fail to find the best path S-D_L-K shown in Figure 9-6:

CASE 1: An R-node on the path S-D_L gets explored at phase 0 along a path other than S-D_L.

CASE 2: The D-node D_L gets explored at phase 0 along two paths other than S-D_L.

CASE 3: A node on the path D_L-K gets explored at phase 1 along two paths other than D_L-K.

Figure 9-9 shows why **CASE 1** can never occur. For **CASE 1** to occur, the cost of the gray path S-G-R would have to be less than or equal to the cost of path S-R. In this case, the path S-G-R-D_L-K would be better than the known best path, which is a contradiction of our initial assumption.

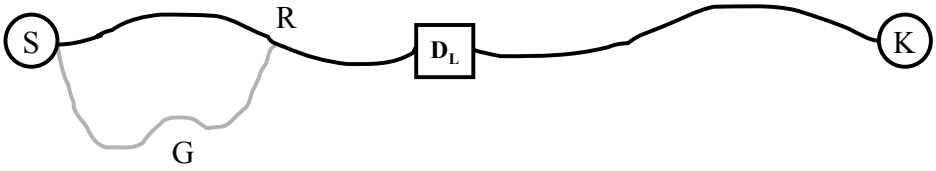

Figure 9-9: The case in which an R-node on the path S-D_L gets explored at phase 0 along some other path.

Figure 9-10 shows an instance of **CASE 2**. The cost of each of the paths S-G1-D_L and S-G2-R2-D_L is less than or equal to the cost of path S-D_L. In this case, the path S-G1-D_L-R2-K would be better than the known best path S-D_L-K, thus contradicting our initial assumption.

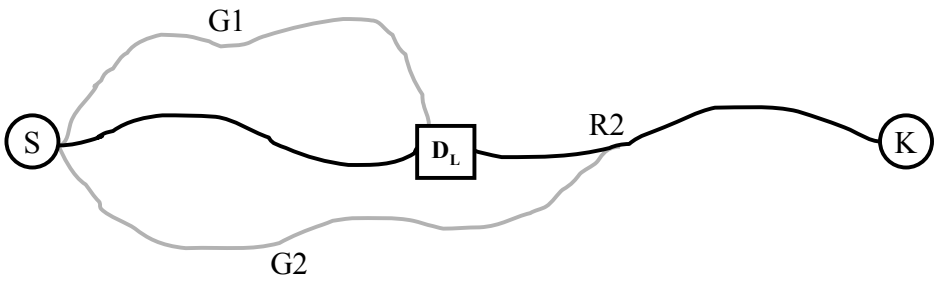

Figure 9-10: D_L gets explored at phase 0 along paths S-G1-D_L and S-G2-R2-D_L.

Figure 9-11 illustrates an example of **CASE 3**, in which a node X on the path D_L-K gets explored at phase 1 along two paths other than D_L-K. There are two possibilities here:

The cost of path S-G1-R1-D-X is less than or equal to the cost of the path to X along the known best path. In this case, the path S-G1-R1-D-X-K would be better than the known best path, which is a contradiction of our initial assumption.

The cost of path S-G2-D-X is less than or equal to the cost of the path to X along the known best path. This means that the path S-G2-D-X-K is better than the known best path, which contradicts our initial assumption.

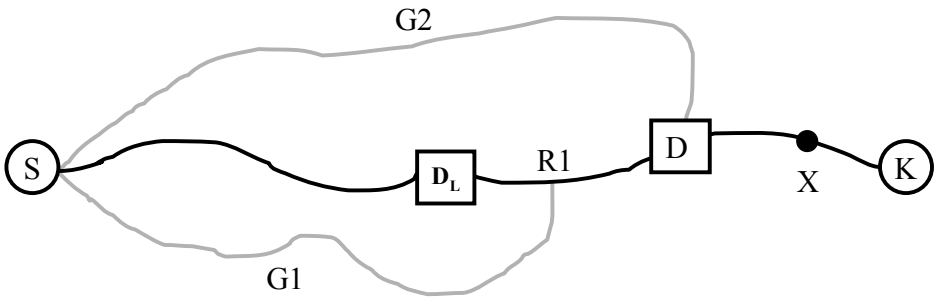

Figure 9-11: Node X can get explored at phase 1 along either S-G2-D-X or S-G1-R1-D-X.

A more detailed case-by-case analysis of the proof of optimality of 2Combined-Phased-Dijkstra can be found in [43]. In this study, we enumerate all the possible sub-cases of **CASE 1**, **CASE 2** and **CASE 3** and separately show that each of the sub-cases contradicts our initial assumption. Consequently, none of **CASE 1**, **CASE 2** or **CASE 3** can occur, implying that 2Combined-Phased-Dijkstra is optimal.

9.2 N-Delay (N_D) Router

In this section, we present a heuristic that uses the optimal 1_D router to build a route for a two terminal N_D signal. This heuristic greedily accumulates D-nodes on the route by using 1_D routes as building blocks. In general, an N_D route is recursively built from an $(N-1)_D$ route by successively replacing each segment of the $(N-1)_D$ route by a 1_D route and then selecting the lowest cost N_D route. Figure 9-12 is an abstract illustration of how a 3_D route between S and K is found. In the first step, we find a 1_D route between S and K, with D11 being the D-node where we pick up a register. At this point, we increase the sharing cost [33] of all nodes that constitute the route S-D11-K. In the second step, we find two 1_D routes, between S and D11, and D11 and K. The sequence of sub-steps in this operation is as follows:

- Decrease sharing cost of segment S-D11.
- Find 1_D route between S and D11 (S-D21-D11). Store the cost of route S-D21-D11-K.
- Restore segment S-D11 by increasing the sharing cost of segment S-D11.
- Decrease sharing cost of segment D11-K.
- Find 1_D route between D11 and K (D11-D22-K). Store the cost of route S-D11-D22-K.
- Restore segment D11-K by increasing the sharing cost of segment D11-K.
- Select the lowest cost route, either S-D21-D11-K or S-D11-D22-K.

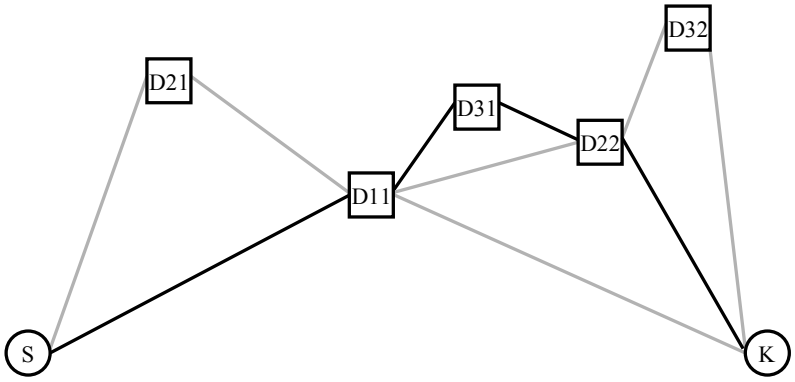

Figure 9-12: Building a 3_D route from 1_D routes.

Suppose the lowest cost 2_D route is S-D11-D22-K. We rip up and decrease sharing due to the segment D11-K in the original route S-D11-K, and replace it with segment D11-D22-K. Finally, we increase sharing of the segment D11-D22-K. The partial route now is S-D11-D22-K. The sequence of sub-steps in step three is similar. Segments S-D11, D11-D22 and D22-K are successively ripped up, replaced with individual 1_D segments, and for each case the cost of the entire 3_D route between S and K is stored.

The lowest cost route is then selected. In Fig. 12, the 3_D route that is found is shown in dark lines, and is S-D11-D31-D22-K.

The number of 1_D explorations launched for the 3_D route that we just discussed is 1 + 2 + 3 = 6. For the general N_D case, the number of 1_D explorations launched is 1 + 2 + ... + N = N(N+1)/2.

9.3 Multi-Terminal Router

The previous section described a heuristic that uses optimal 1_D routes to build a two-terminal N_D route. The most general type of pipelined signal is a multi-terminal pipelined signal. A multi-terminal pipelined signal has more than one sink, and the number of registers separating the source from each sink could differ across the set of sinks. A simple example of a multi-terminal pipelined signal ***sig*** was shown in Figure 9-1. The sinks K1, K2 and K3 must be separated from the source S by 3, 4 and 5 registers respectively. We now demonstrate how a route for a multi-terminal signal can be found by taking advantage of the 1_D and N_D routers.

In a manner similar to the Pathfinder algorithm, the routing tree for a multi-terminal pipelined signal is built one sink at a time. Each sink is considered in non-decreasing order of register separation from the source of the signal. The multi-terminal router starts by finding a route to a sink that is the least number of registers away from the source. Since finding a route to the first sink is a two-terminal case, we use the two-terminal N_D router to establish a route between the source and first sink. The remainder of this section examines the task of expanding the route between the source and the first sink to include all other sinks.

We explain the multi-terminal router via a simple example. Assume a hypothetical signal that has a source S and sinks K2 and K3. K2 must be separated from S by 2 registers, whereas K3 must be separated by 3 registers. Sink K2 is considered first, and the N_D router is used to find a 2_D route between S and K2. In Figure 9-13 (a), the route S-D1-D2-K2 represents the 2_D route between S and K2, and constitutes the *partial_routing_tree* of the signal. In general, the *partial_routing_tree* of a multi-terminal pipelined signal can be defined as the tree that connects the source to all sinks that have already been routed.

After a route to K2 is found, the router considers sink K3. As was the case in the N_D router, we accumulate registers on the route to K3 one register at a time. Thus, we start by finding a 1_D route to K3, then a 2_D route, and finally a 3_D route to K3. It can be seen that a 1_D route to K3 can be found either

from the 0_D segment S-D1 by going through another D-node, or from the 1_D segment D1-D2 directly. However, it is not necessary to launch independent wavefronts from segments S-D1 and D1-D2. This is because both wavefronts can be combined into a single 1_D search in which segment S-D1 constitutes the starting component of the phase 0 wavefront, and segment D1-D2 constitutes the starting component of the phase 1 wavefront. Setting up the 1_D search in such a way could find a 1_D path from S-D1 or a 0-delay path from D1-D2, depending on which is of lower cost. Assume that P1-D_A-K3 is the 1_D route found to K3 (Figure 9-13 (b)). After the 1_D route to K3 is found, the sharing cost of the nodes that constitute P1-D_A-K3 is increased. The segment P1-D_A-K3 is called the *surviving_candidate_tree*. The *surviving_candidate_tree* can be defined as the tree that connects the sink (K3 in this case) under consideration to some node in the *partial_routing_tree* every time an N_D route ($1 \le N \le 3$ in this case) to the sink is found. Thus, a distinct *surviving_candidate_tree* results immediately after finding the 1_D, 2_D, and 3_D routes to K3.

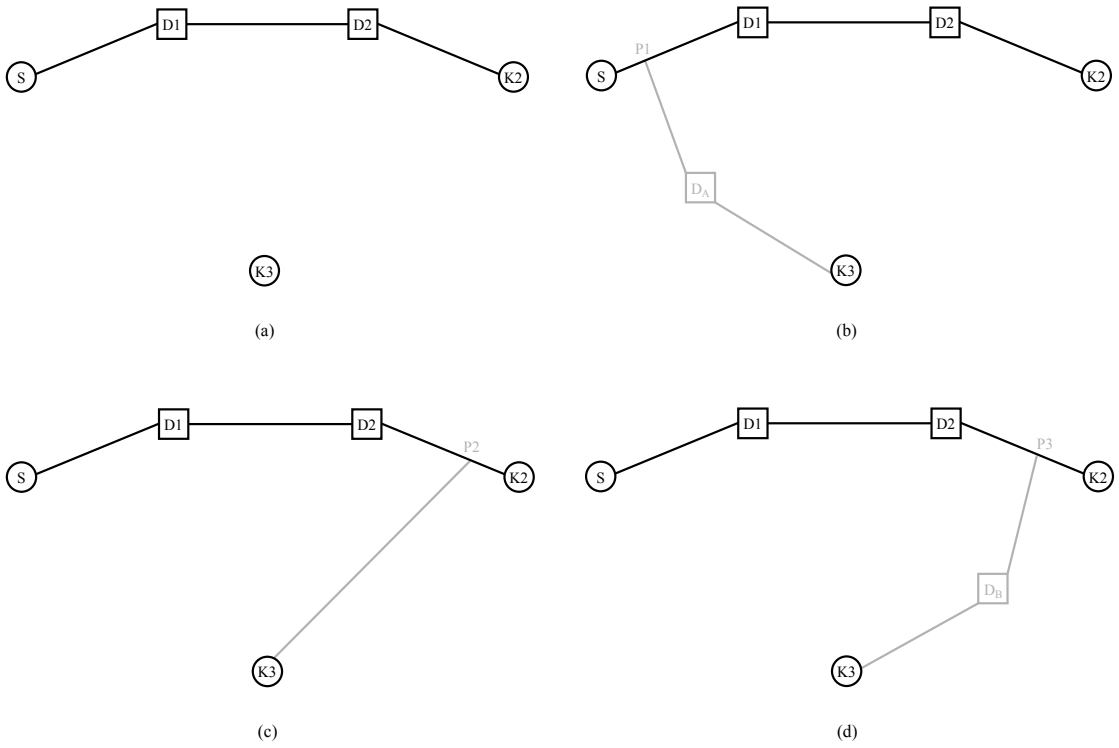

Figure 9-13: (a) 2_D route to K2 using the two-terminal N_D router. S-D1-D2-K2 is the *partial_routing_tree*. (b) 1_D route to K3. P1-D_A-K3 is found by launching a 1_D exploration that starts with segment S-D1 at phase 0 and segment D1-D2 at phase 1. P1-D_A-K3 is the *surviving_candidate_tree*. (c) 2_D route to K3. P2-K3 is now the *surviving_candidate_tree*. (d) P3-D_B-K3 is the final *surviving_candidate_tree*, and this tree is joined to the *partial_routing_tree* S-D1-D2-K2 to complete the route to K3.

Next, we attempt to find a 2_D route to K3. Before explaining specifics, it is important to point out here that while finding an N_D route to a sink we try two options. The first is to alter the *surviving_candidate_tree* to include an additional D-node as was done in the two terminal N_D router. The second option is to use the N_D and $(N-1)_D$ segments in the *partial_routing_tree* together to start a 1_D exploration. The lower cost option is chosen, and this becomes the new *surviving_candidate_tree*.

For finding a 2_D route to K3, we first modify P1-D_A-K3 to include another D-node much in the same way that a two terminal 2_D route is built from an already established 1_D route. The segments P1-D_A and D_A-K3 are each separately replaced by optimal 1_D routes, and the lowest cost route is stored. To evaluate the second option, we rip up the segment P1-D_A-K3 (Figure 9-13 (b)) and launch a 1_D search using segments D1-D2 at phase 0 and D2-K2 at phase 1. The cost of the resultant 1_D route is also stored. The lower cost route amongst the two options is chosen, and the sharing cost of the nodes that constitute this route is increased. This selected route becomes the new *surviving_candidate_tree*. In Figure 9-13 (c), assume that the lower cost route that is selected is the segment P2-K3 shown in gray.

Finally, the segment P2-K3 is ripped up and a 1_D exploration from the segment D2-K2 is launched at phase 0 to complete the 3_D route to K3 (Figure 9-13 (d)). Figure 9-14 presents pseudo code for the multi-terminal routing algorithm. **`Net`** is the multi-terminal signal that is to be routed. Without loss of generality, we assume that **`Net`** has at least two sinks, and each sink is separated from **`Net`**'s source by at least one D-node. $\mathbf{P_{RT}}$ contains the *partial_routing_tree* during the execution of the algorithm, and $\mathbf{C_{RT}}$ contains the *surviving_candidate_tree*. $\mathbf{Src_{Net}}$ is the source of the signal **`Net`**, while $\mathbf{S_K}$ is an array that contains **`Net`**'s sinks. **N_D`-Router`** is the N-Delay router presented in the previous section.

```
Multi-Terminal-Router (Net){
  P_RT = ϕ; C_RT = ϕ;
  Sort elements of S_K in non-decreasing order
    of Dnode-separation from Src_Net;
  Use the two-terminal N_D-Router to find
    route R from Src_Net to S_K[1];
  Add R to the partial routing tree P_RT;
  Foreach i in 2…|S_K| {
    k_i = S_K[i];
    d_i = num Dnodes between Src_Net and k_i;
    Foreach j in 1…d_i {
      Use the two-terminal N_D-Router to find
        a j_D route called RN_j by altering the
        (j-1)_D route contained in C_RT;
      Use 2Combined-Phased-Dijkstra to build
        a j_D route called RD_j from the (j-1)_D
        and j_D segments of the route contained
        in P_RT;
      if cost(RD_j) < cost(RN_j) {
        C_RT = RD_j;
      }
      else {
        C_RT = RN_j;
      }
    }
    Add surviving candidate tree C_RT to partial
      routing tree P_RT;
  }
  return the route contained in P_RT;
}
```

Figure 9-14: Pseudo code for the multi-terminal routing algorithm.

9.4 Multiple Register-Sites

The PipeRoute algorithm described in Sections 9.1, 9.2 and 9.3 assumes that register sites (D-nodes) in the interconnect structure can only provide zero or one register. Also, the algorithm does not address the fact that the IO terminals of logic units may themselves be registered. Since pipelined FPGA architectures [15,52] do in fact provide registered IO terminals and multiple-register sites in the interconnect structure, we developed a greedy pre-processing heuristic that attempts to maximize the number of registers that can be acquired at registered IO terminals and multiple-register sites. We present the details of this heuristic in three parts:

9.4.1 Logic Units with Registered Outputs

We try to greedily pick up the maximum allowable number of registers at the source of each pipelined signal. The maximum number of registers that can be picked up at the source is capped by the sink that is separated by the least number of registers from the source. Consider the example in Figure 9-15. The pipelined signal shown has a source S and two sinks K1 and K2 that must be separated from S by two

and five registers respectively. Assuming that up to three registers can be acquired at S, both registers that separate S and K1 can be picked up at S itself, thus eliminating the need to find a 2_D route between S and K1 in the interconnect structure. Instead, we now only need to find a simple lowest-cost route from S to K1, and a 3_D route to K2.

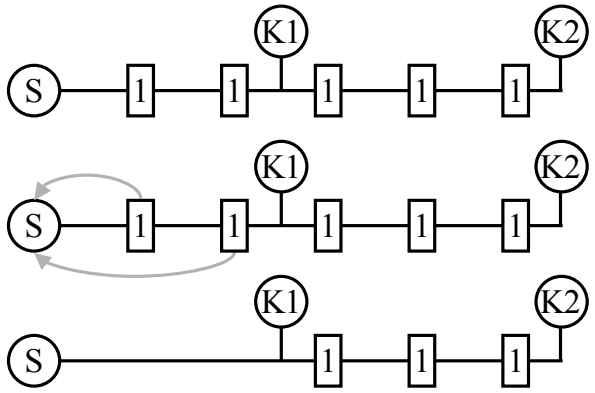

Figure 9-15: Assuming that S can provide up to three registers locally, both the registers between S and K1 can be picked up at S.

9.4.2 Logic Units with Registered Inputs

In this case, we push as many registers as possible into each sink of a pipelined signal. In Figure 9-16, if we again assume that each sink can provide up to three registers locally, both registers between S and K1 can be moved into K1, while three registers between S and K2 can be moved into K2. This leaves us with the task of finding a simple lowest-cost route to K1 and a 2_D route to K2.

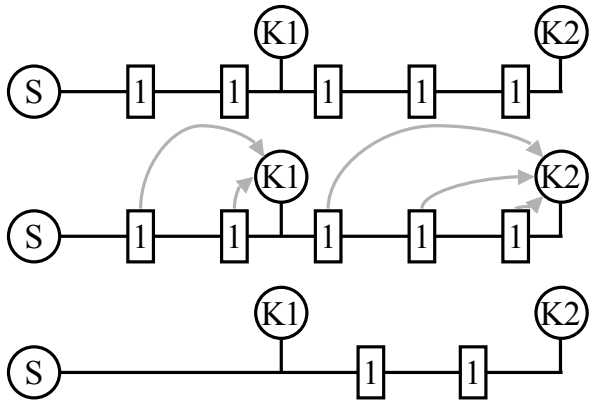

Figure 9-16: Assuming that the sinks K1 and K2 can locally provide up to three registers, both registers between S and K1 and three of the five registers between S and K2 can be picked up locally at the respective sinks.

9.4.3 Multiple-Register Sites in the Interconnect Structure

Multiple-register sites in the interconnect structure provide an opportunity to significantly improve the routability of pipelined signals. In Figure 9-17 for example, if we assume that each register site (D-node) in the interconnect can provide up to three registers, the task of finding a two terminal 9_D route simplifies to finding a route that with at least three D-nodes. For a multi-terminal pipelined signal, every time an N_D route to the new sink is to be found, we use all existing N_D, $(N-1)_D$, $(N-2)_D$, and $(N-3)_D$ segments in the current partially built routing tree to start an exploration that finds a single D-node. Since each D-node can be used to pick up between zero and three registers, we use all segments within the current, partially built routing tree that are less than or equal to three registers away from the new sink.

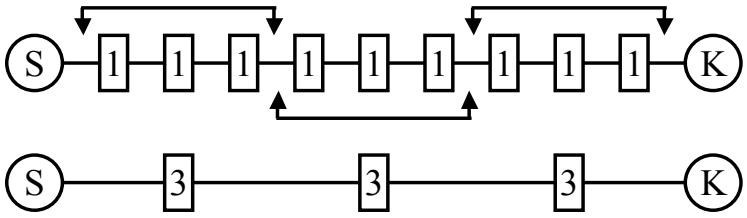

Figure 9-17: Finding a 9_D route between S and K can effectively be transformed into a 3_D pipelined routing problem.

The intuition behind the development of the greedy heuristics in this section is to aggressively reduce the number of register-sites that need to be found in the interconnect structure. The heuristic is clearly routability-driven, since reductions in the number of interconnect registers favorably impact the routability of pipelined signals. Due to the finite nature of an FPGA's interconnect structure, any place-and-route heuristic must consider routability to ensure that a placement can be successfully routed.

A shortcoming of the greedy heuristic is that long segments of a pipelined signal may get unpipelined because of the removal of registers from the interconnect structure. This phenomenon is illustrated in Figure 9-18. Assume that a maximum of four registers can be picked up at the sinks K1 – K8. In this case, one interconnect register will be moved into K1, two into K2, three into K3, and four into K4-K8. This process effectively unpipelines a long segment, which in turn may increase the critical path delay of a netlist.

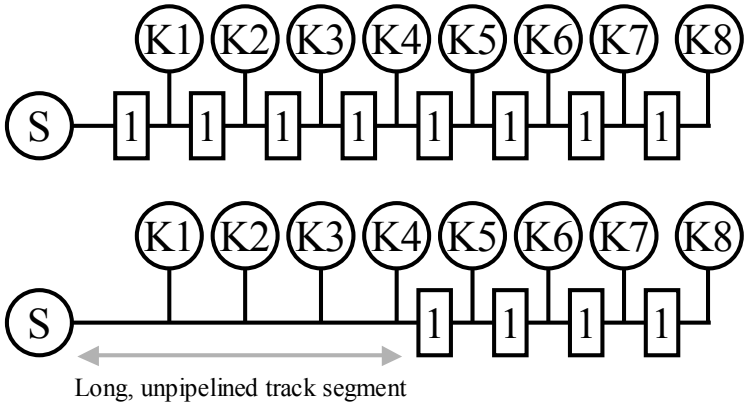

Figure 9-18: Pushing registers from the interconnect structure into functional unit inputs sometimes results in long, unpipelined track segments.

9.5 Timing-Aware Pipelined Routing

Since the primary objective of pipelined FPGAs is the reduction of clock cycle time, it is imperative that a pipelined routing algorithm maintains control over the criticality of pipelined signals during routing. In making PipeRoute timing aware, we draw inspiration from the Pathfinder algorithm. While routing a signal, Pathfinder uses the criticality of the signal in determining the relative contributions of the congestion and delay terms to the cost of routing resources. If a signal is near critical, then the delay of a routing resource dominates the total cost of that resource. On the other hand, if the signal's criticality is considerably less than the critical path, the congestion on a routing resource dominates.

In the case of pipelined routing, the signal's route may go through multiple D-nodes. Consequently, the routing delay incurred in traversing the route from source to sink may span multiple clock cycles. Also, the location of D-nodes on the route may be different across routing iterations. This is because PipeRoute may have to select different routes between the source and sink of a signal to resolve congestion. In Figure 9-19 for example, the 2_D route between S and K may go through different D-nodes at the end of iterations i, i+1 and i+2 respectively.

To address these problems, we treat D-nodes like normal registers during the timing analysis at the end of a routing iteration. Once the timing analysis is complete, we are faced with making a guess about the overall criticality of a pipelined signal. Note that different segments of a pipelined signal's route could be at different criticalities (Figure 9-19). Our solution is to make a pessimistic choice. Since we know the individual criticalities of signals sourced at each D-node, we make the criticality of the pipelined signal equal to the criticality of the most critical segment on the route. Thus, when the pipelined signal

is routed during the next iteration, the most critical segment of the signal's previous route determines the delay cost of routing resources.

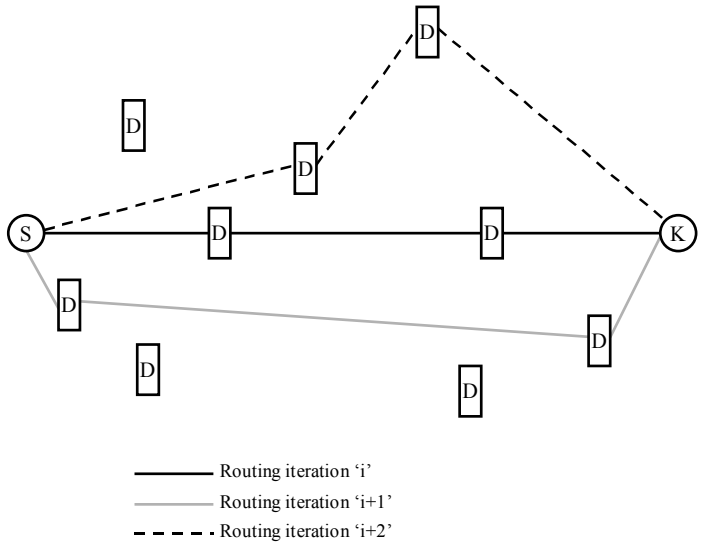

Figure 9-19: The route between source S and sink K of a signal may go through different D-nodes at the end of successive routing iterations. Also, since each D-node on the route is used to pick up a register, different segments on the route may be at different criticalities.

9.6 Placement Algorithm

The placement of a netlist is determined using a Simulated Annealing [26,38] algorithm. The cost of a placement is formulated as a linear function of the maximum and average *cutsize*, where *cutsize* is the number of signals that need to be routed across a vertical partition of the architecture for a given placement. Since the RaPiD interconnect structure provides a fixed number of routing tracks, the cost function must be sensitive to changes in maximum cutsize. At the same time, changes in average cutsize also influence the cost of a placement. This is because average cutsize is a measure of the total wirelength of a placement.

Pipelining information is included in the cost of a placement by mapping each pipelining register (a pipelining register is a register that must be mapped to an interconnect register) in the netlist to a unique BC in the interconnect structure. Our high-level objective in mapping pipelining registers to BCs is to place netlist components such that the router is able to find a sufficient number of BCs in the interconnect structure while routing pipelined signals. A more detailed discussion of the placement algorithm can be found in [40] and [43].

Since pipelining registers are explicitly placed, it might be possible to solve the register allocation problem during placement. The pipelining registers in a netlist could be mapped to registered switch-points in the architecture, and a simulated annealing placement algorithm could determine a placement of the pipelining registers. After the placement phase, a conventional FPGA router (Pathfinder) could be used to route the signals in the netlist. While this approach is attractive for its simplicity and ease of implementation, it has a serious shortcoming. A placement of a netlist that explicitly maps pipelining registers to registered switch-points eliminates portions of the routing graph. This is because a registered switch-point that is occupied by a particular pipelining register cannot be used by signals other than the signals that connect to that pipelining register. As a consequence, the search space of a conventional FPGA router is severely limited, and this results in solutions of poor quality.

To validate our hypothesis, we ran an experiment on a subset of the benchmark netlists. The objective of the experiment was to find the size of the smallest RaPiD array needed to route (using a pipelining-unaware router Pathfinder) placements produced by the algorithm described in this section. Note that pipelining registers were explicitly mapped to BCs in the interconnect structure, and the post-placement routing graph was modified to reflect the assignment of pipelining registers to BCs.

Table 9-1: Overhead incurred in using a pipelining-unaware router (Pathfinder) to route netlists.

NETLIST	NORM. AREA
firtm	1
sobel	1
fft16	1.6
imagerapid	FAIL
cascade	FAIL
matmult4	FAIL
sort_g	FAIL
sort_rb	FAIL
firsymeven	FAIL

Table 9-1 presents the results of this experiment. Column 1 lists the netlists in our benchmark set. Column 2 lists the minimum-size array required to route each netlist using Pathfinder. The entries in column 2 are normalized to the minimum-size RaPiD array needed to route the netlists using PipeRoute. A “FAIL” entry in column 2 means that the netlist could not be routed on any array whose

normalized size was between 1.0 – 2.0. Table 9-1 shows that Pathfinder was unable to route a majority of netlists on arrays that had double the number of logic and routing resources needed to route the placements using PipeRoute. This result clearly showed that pipelining register allocation is best done during the routing phase.

9.7 Experimental Setup and Benchmarks

The set of benchmark netlists used in our experimentation includes implementations of FIR filters, sorting algorithms, matrix multiplication, edge detection, 16-point FFT, IIR filtering and a camera imaging pipeline. While selecting the benchmark set, we included a diverse set of applications that were representative of the domains to which the RaPiD architecture is targeted. We also tried to ensure that the benchmark set was not unduly biased towards netlists with too many or too few pipelined signals. Table 9-2 lists statistics of the application netlists in our benchmark set. Column 1 lists the netlists, column 2 lists the total number of nets in each netlist, column 3 lists the percentage of nets that are pipelined, column 4 lists the maximum number of registers needed between any source-sink terminal pair in the netlist (this number is similar to the latency of the application), and column 5 lists the average number of registers needed across all source-sink terminal pairs in the netlist.

Table 9-2: Benchmark application netlist statistics.

NETLIST	NUM NETS	% PIPELINED	MAX DEPTH	AVG DEPTH
firtm	158	3	16	5.5
fft16	94	29	3	0.74
cascade	113	40	21	3.88
matmult4	164	44	31	4.62
sobel	74	44	5	1.44
imagerapid	101	51	12	3.46
firsymeven	95	54	31	6.98
sort_g	70	65	35	4.98
sort_rb	63	71	35	5.42

While the size of the netlists in Table 9-2 might seem small, remember that a single pipelined signal represents multiple routing problems. An example of a pipelined signal in the netlist *sort_rb* has 38 sinks. The number of registers that separate the 38 sinks from the source is evenly distributed between 0 registers and 35 registers. Although this signal is counted as a single signal in Table 9-2, finding a route for this signal may require hundreds of individual routing searches. Thus, routing the pipelined signals

in the benchmark netlists clearly represents a problem of reasonable complexity. Also note that RaPiD is a coarse-grained architecture. Thus, a single net represents a 16-bit bus.

Applications are mapped to netlists using the RaPiD compiler, and the architecture is represented as an annotated structural Verilog file. Area models for the RaPiD architecture are derived from a combination of the current layout of the RaPiD cell, and transistor-count models. The delay model is extrapolated from SPICE simulations. Each netlist is placed using the algorithm presented in Section 9.6. The placement algorithm places pipelining registers into BC positions in order to model the demands of pipelining. However, the BC assignments are removed before routing to allow PipeRoute full flexibility in assigning pipelining registers.

The netlists are routed using timing-aware PipeRoute that can handle multiple-register IO and interconnect sites. A netlist is declared unrouteable on an architecture of a given size (where size is the number of RaPiD cells that constitute the architecture) if PipeRoute fails to route the netlist in 32 tracks.

9.8 Results

9.8.1 Experiment 1

The objective of our first experiment was to quantify the area overhead incurred in routing the benchmark netlists on an optimized RaPiD architecture [44]. The logic units in this architecture have registered input terminals (the original RaPiD architecture has registered output terminals), and between 0 – 3 registers can be acquired at each input terminal and BC. Also, unlike the original RaPiD architecture, the optimized RaPiD architecture in [44] has nine GPRs in every RaPiD cell.

We acquired area numbers by running the entire set of benchmarks through two place-and-route flows. The first is a pipelining-unaware flow that treats netlists as if they were unpipelined. Specifically, all pipelined signals in a netlist are treated like normal, unpipelined signals (Figure 9-20). The pipelining-unaware placement tool attempts to reduce only maximum and average cutsize. The pipelining-unaware router attempts only connectivity routing, since there are no registers to be found in the interconnect structure. The pipelining-unaware place and route flow provides a lower-bound on the size of the smallest architecture needed to successfully route the benchmark netlists. This is because the best area that we can expect from a pipelining-aware flow would be no better than a pipelining-unaware flow that ignores pipelining altogether.

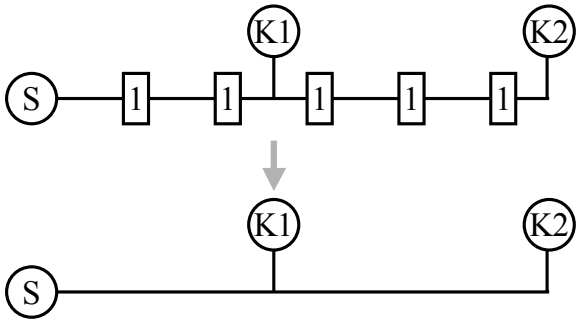

Figure 9-20: Unpipelining a pipelined signal. The pipelined signal (top) is transformed into an unpipelined signal (bottom).

The second flow is the pipelining-aware flow described in this paper. Netlists are placed using the algorithm described in Section 9.6, and routed using purely congestion-driven PipeRoute. For both approaches, we recorded the area of the smallest architecture required to successfully route each netlist. Table 9-3 lists the smallest areas found for each benchmark netlist using both pipelining-aware and pipelining-unaware flows. The area overhead varied between 0% (for the netlists *fft16*, *matmult4* and *sobel*) and 44% for the netlist *firsymeven*. Overall, the geometric mean of the overhead incurred across the entire benchmark set was 18%. We regard this a satisfactory result, since a pipelining-aware flow incurs less than a 20% penalty over a likely unachievable lower-bound.

Table 9-3: Experiment 1 – Area comparison between pipelining-aware and pipelining-unaware place and route flows.

NETLIST	PIPELINING UNAWARE AREA (um^2)	PIPELINING AWARE AREA (um^2)
sort_g	3808215	5183743
sort_rb	3808215	5752143
fft16	5712322	5712322
imagerapid	6664376	8025125
firsymeven	6897972	9949143
firtm	7257201	7616430
matmult4	7616430	7616430
cascade	7616430	8753230
sobel	9039119	9039119
GEOMEAN	**6247146**	**7347621**

9.8.2 Experiment 2

The objective of our second experiment was to investigate the performance of timing-aware PipeRoute vs. timing-unaware PipeRoute. For both approaches, we separately obtained the post-route critical path delays of benchmark netlists routed on the smallest possible RaPiD architecture. Table 9-4 shows the results that we obtained. Across the entire benchmark set, timing-aware PipeRoute produced an 8% improvement in critical path delay compared to timing-unaware PipeRoute.

Table 9-4: Experiment 2 – Delay comparison between timing-aware and timing-unaware PipeRoute.

NETLIST	TIMING-UNAWARE (ns)	TIMING-AWARE (ns)
firtm	6.73	6.63
matmult4	8.27	8.57
sort_rb	9.65	12.61
firsymeven	10.97	9.96
sort_g	11.44	6.07
fft16	13.14	11.6
sobel	14.24	13.25
imagerapid	14.36	12.62
cascade	15.45	15.42
GEOMEAN	**11.2112**	**10.29194**

9.8.3 Experiment 3

Our final experiment was to study whether there is any relationship between the fraction of pipelined signals in a benchmark netlist and the area overhead incurred in successfully routing the netlist on a minimum size architecture. The area overhead is a measure of the pipelining 'difficulty' of a netlist and is quantified in terms of the following parameters:

- A_L – The area of the smallest architecture required to successfully route the netlist using a pipelining-unaware place and route flow.
- A_P – The area of the smallest architecture required to successfully route the netlist using a pipelining-aware place and route flow.
- PIPE-COST – The ratio A_P / A_L. This is a quantitative measure of the overhead incurred.

Figure 9-21 shows a plot of PIPE-COST vs. the fraction of pipelined signals in a netlist. The data points represent the PIPE-COST of each netlist in the benchmark set. It can be seen that an increase in the percentage of pipelined signals in a netlist tends to result in an increase in the PIPE-COST of that

netlist. This observation validates our intuition that the fraction of pipelined signals in a netlist roughly tracks the combined architecture and CAD effort needed to successfully route the netlist.

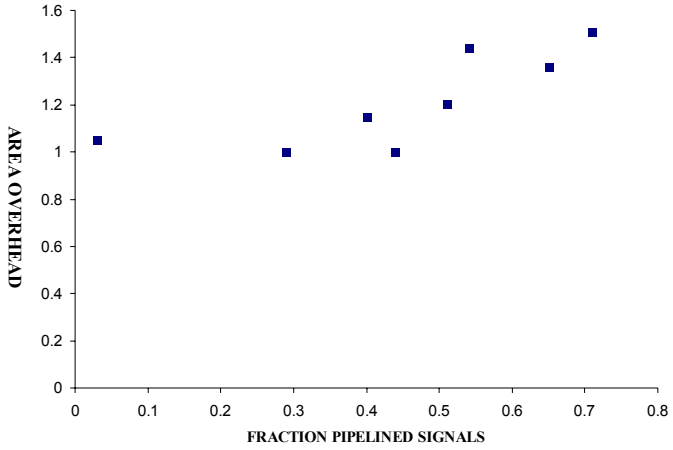

Figure 9-21: Experiment 3 – The variation of PIPE-COST vs. fraction pipelined signals.

9.9 Summary

The main focus of this chapter was the development of an algorithm that routes logically retimed netlists on the RaPiD architecture. We developed an optimal 1_D router, and used it in formulating an efficient heuristic to route two-terminal N_D pipelined signals. The algorithm for routing general multi-terminal pipelined signals borrowed from both the 1_D and N_D routers. Congestion resolution while routing pipelined signals was achieved using Pathfinder. Our results showed that the architecture overhead (PIPE-COST) incurred in routing netlists on the RaPiD architecture was 18%, and that there might be a correlation between the PIPE-COST of a netlist and the percentage of pipelined signals in that netlist.

Chapter 10: Exploring RaPiD's Interconnect Structure

Pipelined FPGA architecture design poses a number of challenges, not the least of which is the composition of the interconnect structure. Earlier work [6] has shown that the design of FPGA interconnect structures involves tradeoffs amongst different parameters like segment-length, switch-box types and layout considerations. However, the interconnect structure of a pipelined FPGA is different. Unlike conventional architectures, the interconnect structure of a pipelined FPGA may include a large number of registers. The number and location of interconnect registers plays an important role in determining the performance of applications mapped to pipelined FPGAs. If the number of interconnect registers is too few, the benefits of pipelining may get lost in long, circuitous routes. On the other hand, the area penalty due to too many interconnect registers may reduce the impact of improvements in clock cycle time.

In this chapter, we parameterize and explore the performance of RaPiD's pipelined interconnect structure. Specifically, we try to answer the following questions:

- What are the benefits of registering the IO terminals of logic units? Note that both RaPiD and HSRA provide register banks at IO terminals.
- How many sites in the interconnect structure should be registered? A related question is how many registers should a single site provide?
- How long should the segments of registered routing tracks be?
- How does the flexibility of the interconnect structure affect the performance of pipelined FPGAs?

To the best of our knowledge, the only previous work that explored pipelined interconnect structures can be found in [46]. In that work, the authors present a limited study that demonstrates speed-ups by adding registers to routing switches. The authors do not explore multiple-register interconnect sites, segment lengths of registered tracks, or the flexibility of the interconnect structure. This work expands the pipelined interconnect exploration space to include these parameters.

10.1 Characterizing Pipelined Interconnect Structures

We now present our interpretation and analysis of the trends that we observed while exploring RaPiD's pipelined interconnect structure. Our primary measure of the quality of a given point in the exploration

space is the post place-and-route geometric mean of the area-delay product across the benchmark set. Netlists are placed using the placement algorithm described in Section 9.6, and routed using timing-aware PipeRoute. The area-delay product of a netlist is measured from the minimum number of RaPiD cells required to route a netlist in less than thirty-two tracks. Area models for the RaPiD architecture are derived from a combination of the current layout of the RaPiD cell and transistor-count models. The delay model is extrapolated from SPICE simulations.

Before presenting our results, we briefly explain the effects of certain important interconnect features on the area and delay of a netlist:

Track Count: The track count of a netlist is the minimum number of tracks required to route the netlist. Track count directly affects area in two ways. First, the area of the IO multiplexers and demultiplexers depends on the number of tracks that connect to them. Second, the number of BCs in the architecture is directly proportional to the number of tracks.

BCs: The frequency and number of BCs in the interconnect structure affects both area and delay. A large number of BCs provide an abundance of interconnect register sites. Consequently, a BC-rich interconnect structure improves the routability of pipelined signals. Routability improvements generally result in track count reductions, and if the area benefit due to such reductions is greater than the area-penalty of a large number of BCs, an overall area win may result. The number and location of BCs in the interconnect structure also influences the delay characteristics of a netlist. One reason is the effects of segmentation on the critical path delay of a netlist [6]. Another reason is that the number of BCs determines the quality of the routes of pipelined signals. Recall that the number of BCs is a direct measure of the number of interconnect registers. In BC-poor architectures, the pipelined router finds long, circuitous routes for heavily pipelined signals. Such poor-quality routes result in a deterioration of the delay characteristics of a netlist.

Based on our observations on the impact of track count and BCs on area and delay, we identified the following parameters that may play an important role in determining the overall performance of RaPiD's pipelined interconnect structure:

- *Registered IO Terminals* – The number of registers that can be locally acquired at the IO terminals of logic units directly affects the number of registers that need to be located in the interconnect structure. The ability to acquire registers at IO terminals may reduce the overall routing effort expended in locating pipelining registers.

- *Bus Connectors* – The number of BCs in the interconnect structure directly impacts both area and delay. As mentioned earlier in this section, a large number of BCs might improve track count and delay. However, the area hit due to too many BCs might offset improvements due to track count and delay reductions.
- *Multiple-Register Bus Connectors* – The number of registers in a BC is a measure of the number of pipelining registers that can be acquired at a single pipelining site. Increasing the number of registers that can be acquired at a BC reduces the total number of BCs that have to be found when routing pipelined signals.
- *Short / Long Track Ratio* – Since BCs can only be found on long tracks, the ratio between the number of short and long tracks affects the total number of BCs that can be found in the interconnect structure. Further, if there are too many short tracks compared to long tracks, then long connections that would have otherwise used long tracks may be forced to use short track segments. This might affect track counts and delay adversely.
- *Datapath Registers (GPRs)* – The GPRs provided in the RaPiD datapath structure offer an important degree of flexibility. Any unoccupied GPRs in the datapath can be used to switch tracks during routing. The ability to switch tracks might improve routability.

The remainder of this section presents our interpretation of the results we obtained while exploring RaPiD's pipelined interconnect structure. The results were obtained by sweeping individual parameters of an experimentally optimized RaPiD architecture.

10.1.1 Registered IO Terminals

Our first step is to explore the possible benefits of logic units that have 'registered' input or output terminals. An IO terminal of a logic unit is registered if the terminal can be connected to the interconnect structure through a local register bank. Local register banks allow pipelined signals to pick up registers at the logic unit, thus reducing the number of registers that have to be found in the interconnect structure.

Figure 10-1 shows the area and delay numbers that we obtained on mapping the benchmark netlists to architectures with registered input, registered output and unregistered terminals. Surprisingly, the effect of registered IO terminals on area is negligible. This is because the area penalty of adding registers to IO terminals nullifies the area benefits attributable to the track count reductions shown in Figure 10-2.

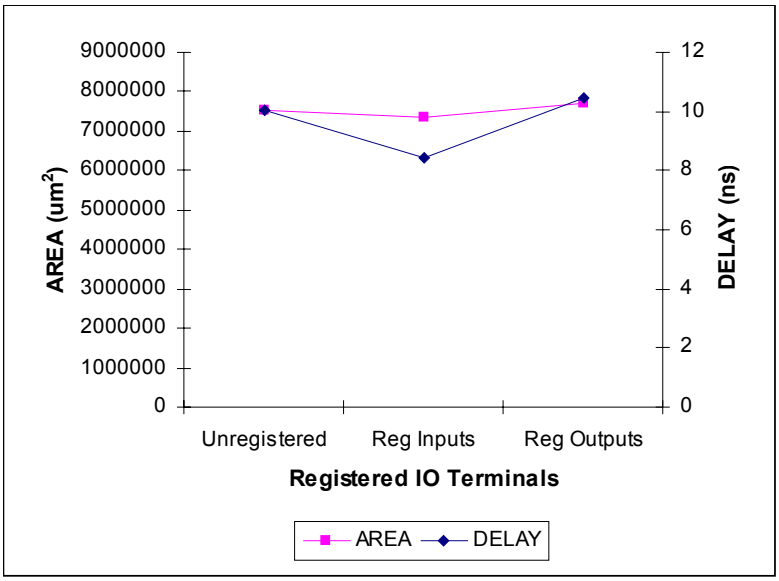

Figure 10-1: Area and delay numbers for architectures with registered outputs, registered inputs and unregistered IO terminals. The "Unregistered" point on the x-axis represents logic units that have no IO registers, the "RegInputs" point represents logic units that have registers at the input terminals, and the "RegOutputs" point represents logic units that have registers at the output terminals.

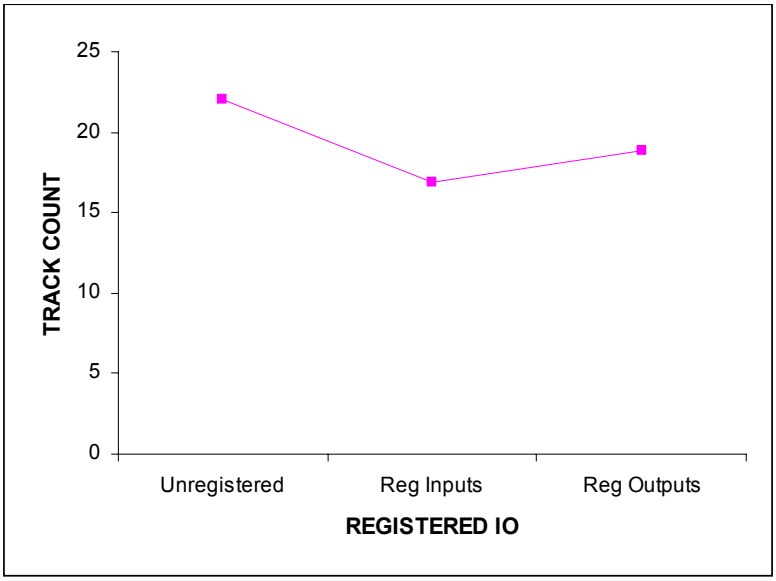

Figure 10-2: Track counts for architectures that have registered input, registered output and unregistered IO terminals. The "Unregistered" point on the x-axis represents logic units that have no IO registers, the "RegInputs" point represents logic units that have registers at the input terminals, and the "RegOutputs" point represents logic units that have registers at the output terminals.

While area is insensitive to registered IO terminals, the delay results of architectures with registered inputs is clearly better. This is because the preprocessing heuristic mentioned in Section 9.4 moves a large number of registers from the interconnect structure in to the inputs of logic units. Consequently, the pipelined router has to find fewer registers in the interconnect structure, which in turn may improve delay due to tighter pipelined routes. Interestingly, architectures with registered outputs show no delay improvement when compared to architectures that have unregistered IO. This is probably because the number of interconnect registers that are moved in to the outputs of logic units is an insignificant fraction of the total number of interconnect registers that have to be found during pipelined routing. Overall, architectures with registered input terminals proved to be the best choice in terms of area-delay product.

10.1.2 Bus Connectors (BCs)

BCs serve as buffered registered switches in the RaPiD interconnect structure The total number of BCs in the interconnect structure plays a major role in determining the overall area and delay of a netlist mapped to the RaPiD architecture. Since a single BC may provide multiple registers, the number of BCs directly impacts the number of pipelining registers available in the interconnect structure. The number of BCs in the interconnect structure is varied by changing the number of BCs per long track in a RaPiD cell. (Hereafter, 'BCs per long track' will simply be called BCs per track).

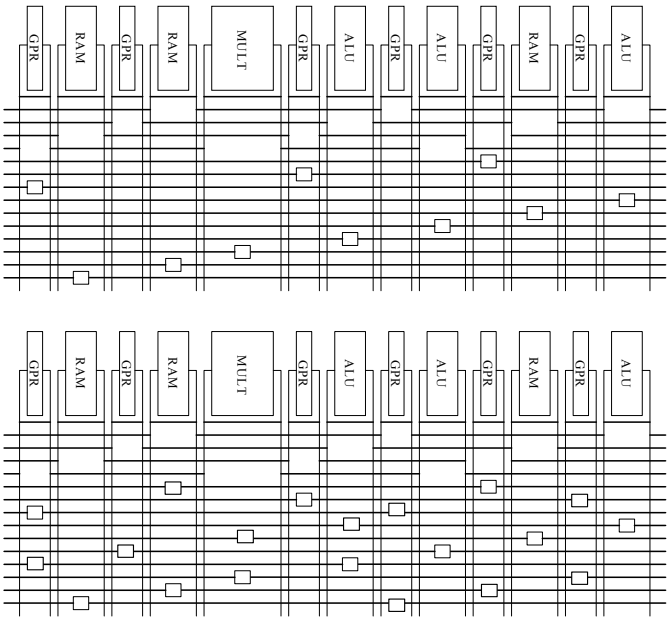

Figure 10-3: A RaPiD cell that has 1 BC per track (top), and a RaPiD cell that has 2 BCs per track (bottom).

For example, the RaPiD cell shown in Figure 10-3 (top) has one BC per track, while the cell shown in Figure 10-3 (bottom) has two BCs per track. Varying the number of BCs per track not only changes the number of interconnect register sites, but also the length of long track segments. Long track segments in Figure 10-3 (top) span thirteen logic units, while long track segments in Figure 10-3 (bottom) span six or seven logic units.

Figure 10-4 shows the area and delay numbers that we obtained as a result of varying the number of BCs per track (the number 0.5 on the x-axis implies architectures that had a single BC per track for every two RaPiD cells). There is a marked improvement in delay when going from half to a single BC per track. This is because at half BC per track, track segments are too long and there are relatively few BCs available for pipelined signals. When we increase the number of BCs per track past one, the delay gradually goes back up. This is because the delay incurred in traversing an increased number of BCs along a long track more than offsets improvements due to shorter track segments and tighter pipelined routes.

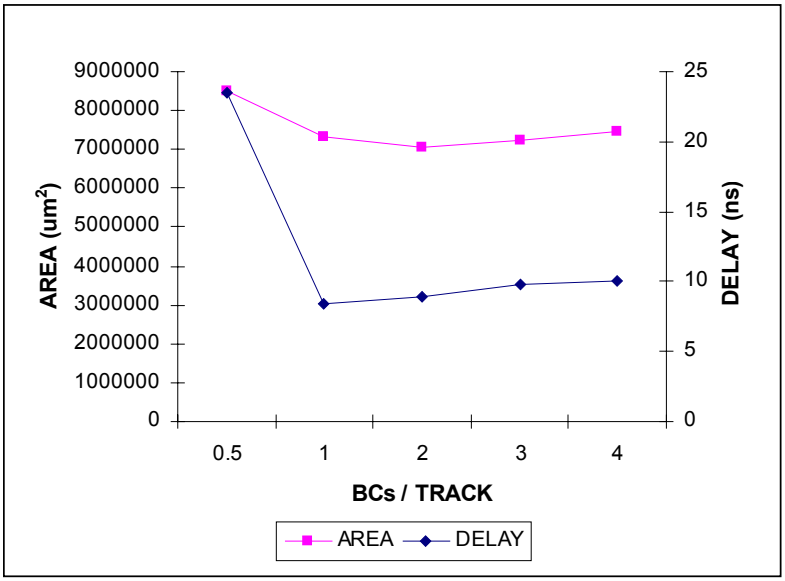

Figure 10-4: The effect of varying number of BCs per track on area and delay.

In terms of area, Figure 10-4 shows a benefit as the number of BCs per track is increased to two. This is consistent with the 45% reduction in track count when the number of BCs per track is increased from half to two (Figure 10-5). The area gradually increases after that due to the fact that the area cost of adding more BCs per track exceeds any improvements in track count.

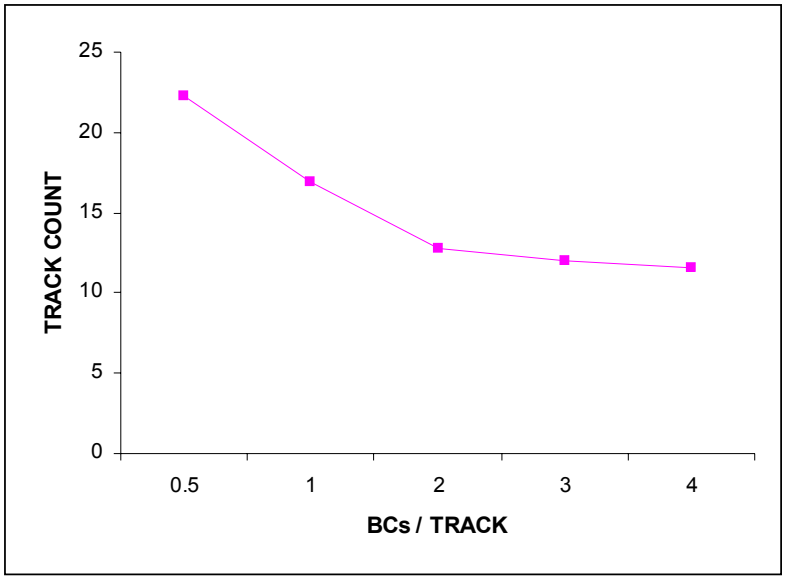

Figure 10-5: The effect of varying number of BCs per track on track count.

Figure 10-6 shows the area-delay product trend that we obtained. The area-delay products at one and two BCs per track are within 1% of each other, which leads us to believe that anywhere between one and two BCs per track is a good architectural choice. We selected one BC per track for our experiments.

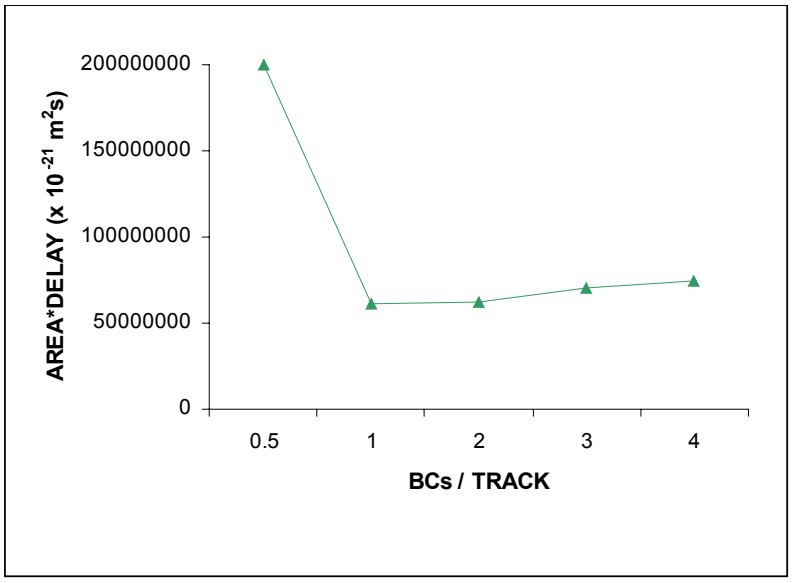

Figure 10-6: The effect of varying number of BCs per track on the area-delay product.

10.1.3 Multiple-Register Bus Connectors

The number of registers in a BC is another parameter that influences the overall area-delay performance of a circuit. An increase in the number of registers per BC allows pipelined signals to pick up a greater number of registers at a single interconnect site. This improves track count because a reduced number of BCs have to be found while routing pipelined signals. At the same time, the delay characteristics of the netlists may also get better due to a reduction in the long, circuitous routes that are found while routing pipelined signals on architectures that have register-poor BCs. Figure 10-7 shows area and delay trends when the number of registers per BC is varied between one and seven.

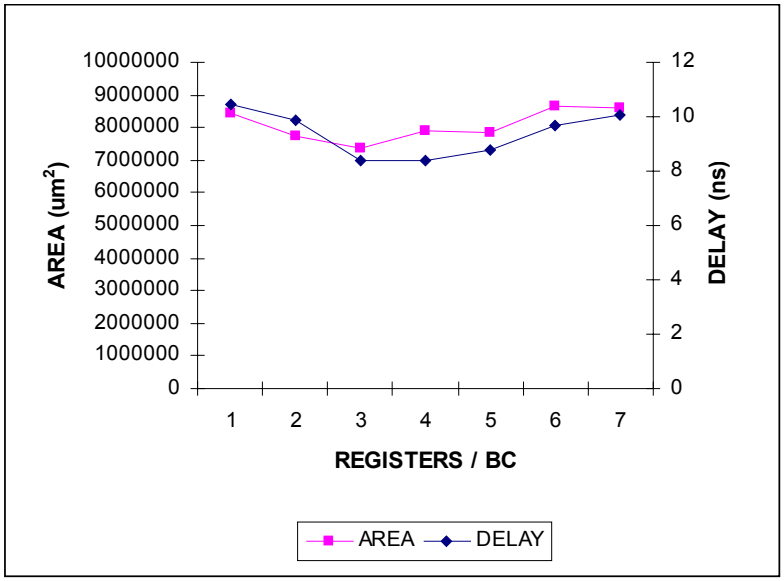

Figure 10-7: The effect of varying number of registers per BC on area and delay.

There is an improvement in area as the number of registers per BC is increased to three. However, the area goes back up as the number of registers per BC is increased past that point. This is because increases in BCs area exceed any area improvements attributable to track-count reductions (Figure 10-8).

At first sight, the delay trend in Figure 10-7 seems surprising. While there is an expected improvement in delay as the number of registers per BC is increased to four, the delay unexpectedly goes back up past that point. A possible reason for this behavior is the greedy manner in which the preprocessing heuristic pushes interconnect registers into logic unit input terminals. While conducting experiments, we assume that the number of registers in a BC is equal to the maximum number of registers that can be

picked up at the inputs of logic units (we made this assumption to limit the number of axes that we explored to a practical number). Thus, if the number of registers per BC is large, so is the number of registers that can be moved into the sinks of a pipelined signal. A shortcoming of this assumption is that long segments of a pipelined signal may get unpipelined because of the removal of registers from the interconnect structure. This phenomenon is illustrated in Figure 10-9. Assume that a maximum of four registers can be picked up at the sinks K1- K8. In this case, one interconnect register will be moved into K1, two into K2, three into K3, and four into K4-K8. This process effectively unpipelines a long-track segment, which in turn may increase the critical path delay of a netlist.

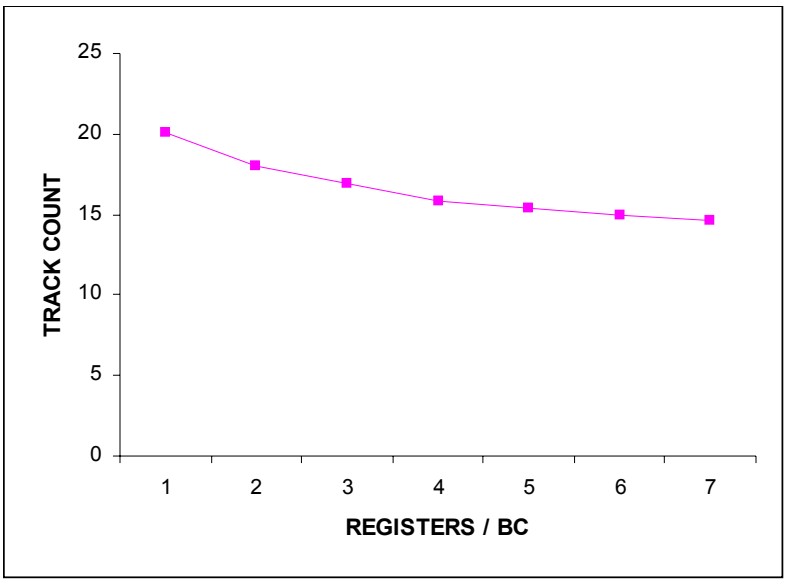

Figure 10-8: The effect of varying number of registers per BC on track count.

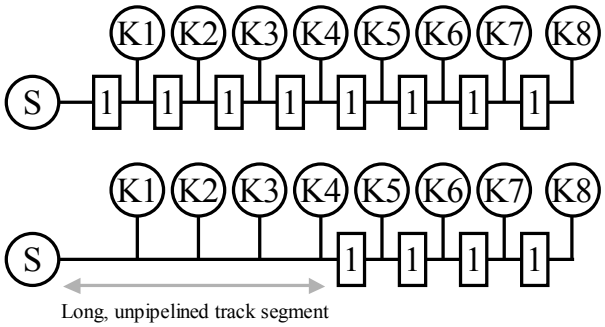

Figure 10-9: Pushing registers from the interconnect structure into logic unit inputs sometimes results in long, unpipelined track segments.

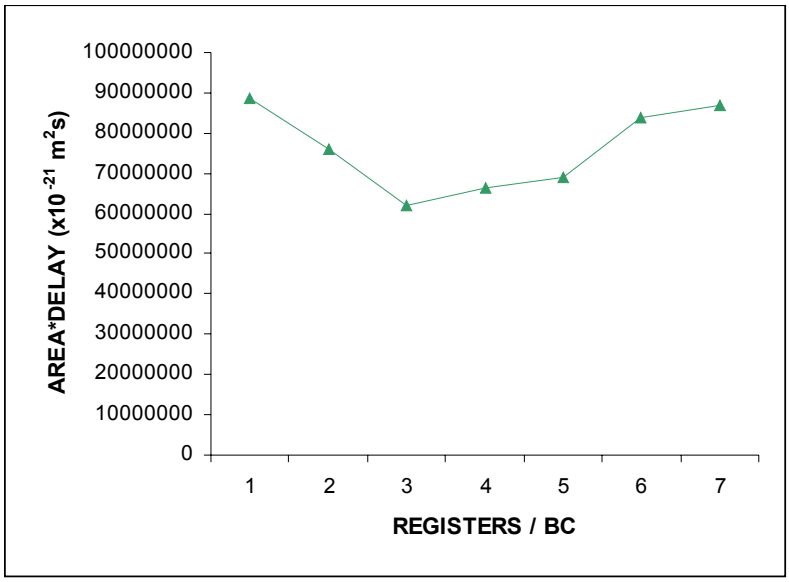

Figure 10-10: The effect of varying number of registers per BC on the area-delay product.

Figure 10-10 shows the area-delay product vs. number of registers per BC. A clear sweet spot can be observed at three registers per BC.

10.1.4 Short / Long Track Ratio

RaPiD's interconnect structure is a mix of short tracks and long tracks. Short tracks achieve local connectivity between logic units. Long tracks are used to traverse longer distances along the datapath, and are segmented by means of BCs. In addition to serving as bidirectional switches, BCs also play the role of interconnect register sites.

We demonstrated earlier that the combined area-delay product of the benchmark netlists is sensitive to the number of BCs per track. Varying the number of BCs per track changes the distribution and total number of BCs in the interconnect structure. Another factor that directly affects the number of BCs is the ratio between short and long tracks. Figure 10-11 shows the area and delay trends that we observed on varying the fraction of short tracks in the architecture. Notice that the delay is higher for architectures that have short-track fractions < 0.28. This trend may be due to the fact that short-track poor architectures force signals to use long-track segments to establish connections that could otherwise have been routed on short-track segments.[9]

[9] In general, the routing delay of a long-track segment exceeds that of a short-track segment. A long segment has more resistance due to its length, and greater fanout capacitance.

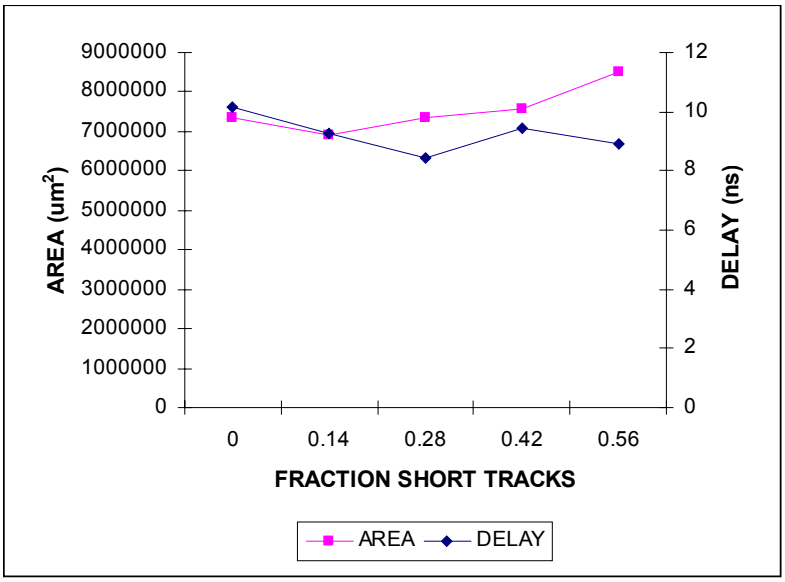

Figure 10-11: The effect of varying fraction of short tracks on area and delay.

For short-track fractions > 0.28, the delay again increases because of two reasons. First, long-track poor architectures force signals to use multiple short-track segments to establish connections that may have otherwise used a single long-track segment[10]. Second, the reduction in the number of BCs increases the need for long, circuitous routes for heavily pipelined signals.

The area curve has a minimum at 0.14. Architectures that are relatively poor in short tracks pay an area penalty due to an excessive number of BCs and an increased track count (Figure 10-12). The track count increases because signals that could have been routed on segments on the same short track have to use segments on different long tracks. As the short-track fraction is increased past 0.14, the area goes back up. This is again due to an increase in track count (Figure 10-12). This time however, the track count increases because fewer BCs are available to pick up registers in the interconnect structure.

[10]Note that unoccupied GPRs in the datapath can be used by signals to switch tracks arbitrarily.

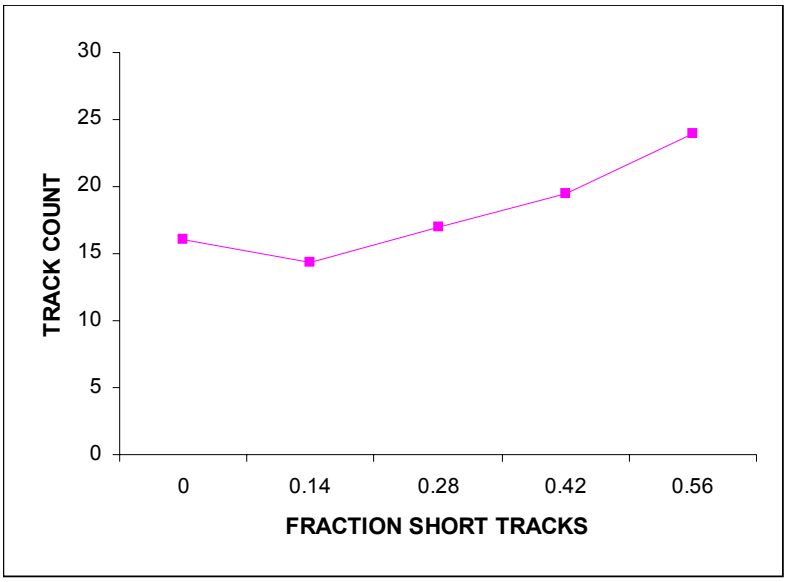

Figure 10-12: The effect of varying fraction of short tracks on track count.

The area-delay trend vs. the fraction of short tracks in Figure 10-13 shows a clear minimum at 0.28.

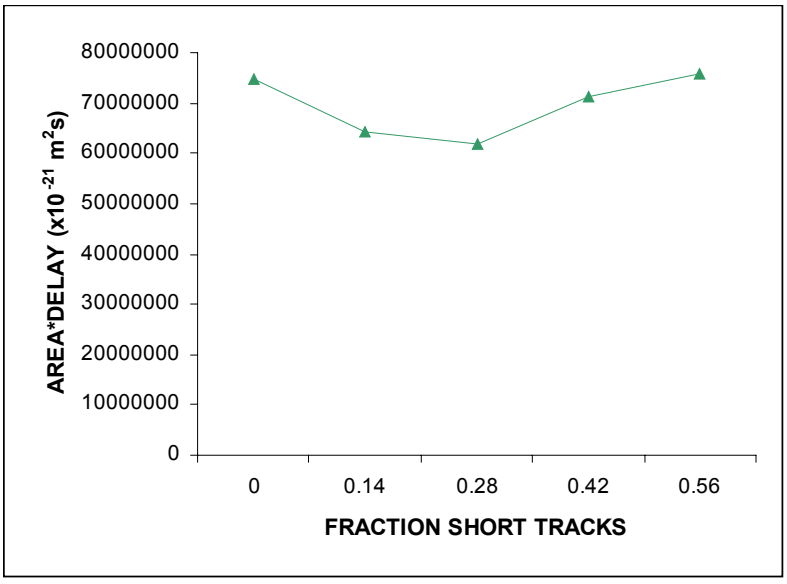

Figure 10-13: The effect of varying fraction of short tracks on the area-delay product.

10.1.5 Datapath Registers (GPRs)

The main purpose of GPRs in the RaPiD architecture is to serve as pipelining sites in the datapath structure. However, any unoccupied GPR units can also be used by signals to switch tracks in the interconnect structure. A large number of unoccupied GPRs in the datapath structure increases the flexibility of the interconnect structure. Consequently, the total number of GPRs in the architecture plays a role in determining the routability of netlists that are mapped to the RaPiD architecture. This role may be especially pronounced in netlists that occupy a large percentage of GPRs in the datapath. Figure 10-14 shows area and delay trends when the number of GPRs per RaPiD cell is varied between five and ten (the number 6 on the x-axis corresponds to the number of GPRs provided in the original RaPiD cell shown in Figure 10-3 (top)).

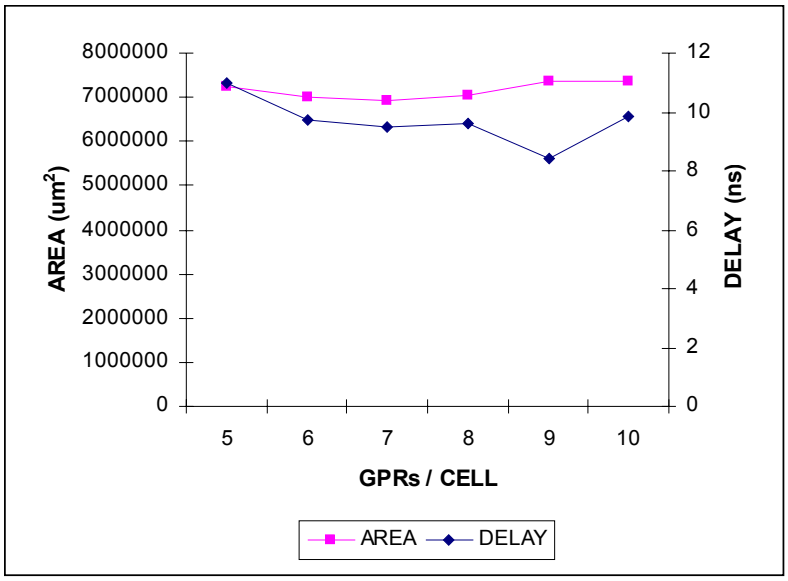

Figure 10-14: The effect of increasing the number of extra GPRs / RaPiD cell on area and delay.

Figure 10-14 shows that varying the number of GPRs per RaPiD cell produces marginal area benefits while going from five to seven GPRs per cell. This is consistent with the reduction in track count shown in Figure 10-15. When the number of GPRs / cell is increased past seven, the area goes back up due to the penalty of adding extra GPRs to the architecture. Notice in Figure 10-15 that track count remains relatively constant past seven GPRs.

The delay curve in Figure 10-14 has a minimum at nine GPRs per cell. Architectures that have fewer than nine GPRs per cell do not have sufficient switching sites. Consequently, the pipelined router is

forced to find potentially longer routes for pipelined signals. The delay goes back up past nine GPRs per cell because the delay of track segments increases. This increase can be attributed to the greater fanout capacitance per segment that results when the number of GPRs per cell is increased. Figure 10-16 shows that the area-delay product is minimum for architectures that have nine GPRs per RaPiD cell.

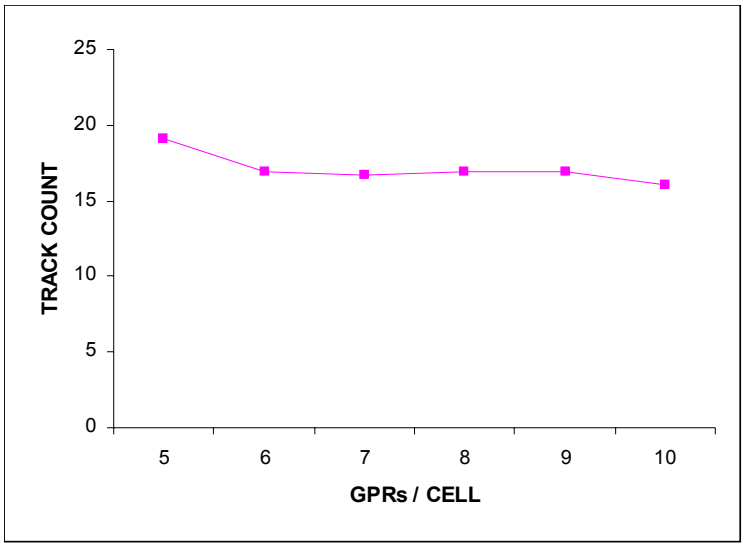

Figure 10-15: The effect of increasing the number of extra GPRs / RaPiD cell on track count.

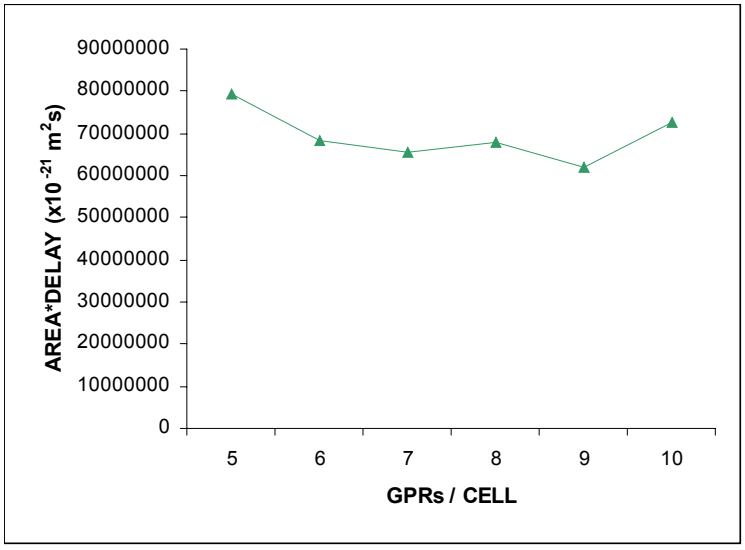

Figure 10-16: The effect of increasing the number of extra GPRs / RaPiD cell on area-delay product.

10.2 Quantitative Evaluation

In this section we quantify the benefits of exploring RaPiD's pipelined interconnect structure. We reproduce the RaPiD cell from Figure 10-3 (top) in Figure 10-17. The RaPiD cell has registered outputs, a single BC per track, three registers per BC and 28% short tracks.

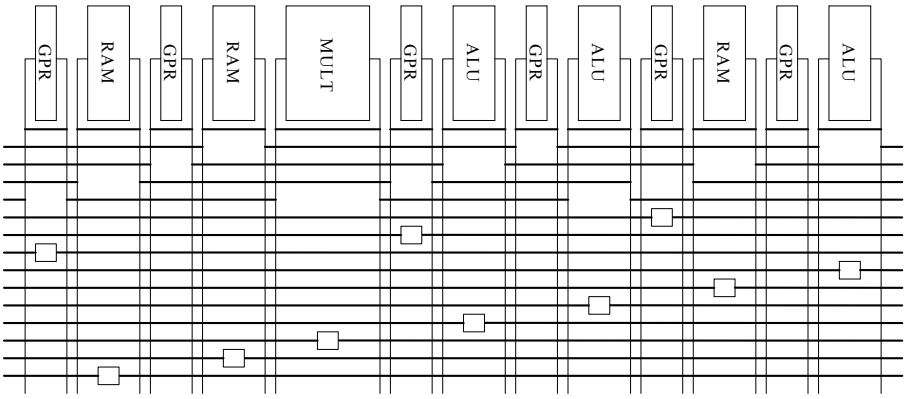

Figure 10-17: A RaPiD cell. Several cells can be tiled together to form a representative architecture.

We first note that the choices of a single BC per track, three registers per BC and 28% short tracks are in fact consistent with the findings of our exploration in Section 10.1. At the same time, there are differences between RaPiD and our findings. First, RaPiD has registered outputs. Our exploration found that registered inputs are a better choice. Second, we found that the number of GPRs per RaPiD cell is insufficient, and that there should be nine GPRs per RaPiD cell (three more than the six GPRs shown in Figure 10-17).Table 10-1 presents a comparison between the original RaPiD architecture and the best post-exploration architecture that we found. Column 1 lists the benchmark netlists, column 2 lists area-delay products (all area-delay product values are $x10^{-21}m^2s$) measured from the post-exploration architecture, column 3 lists area-delay products measured from RaPiD, column 4 lists percentage improvements, and column 5 lists the fraction of pipelined signals in each netlist. RaPiD outperforms the post-exploration architecture for netlists that have less than 30% pipelined signals, while the post-exploration architecture performs better than RaPiD for netlists that have more than 54% pipelined signals. Overall, the post-exploration architecture's area-delay product is 19% better than that of the RaPiD architecture.

Table 10-1: A quantitative comparison of RaPiD with the post-exploration architecture.

Netlist	Post-Explore	RaPiD	% Improve	fracn pipe
firtm	53315010	42731038	- 25%	0.03
fft16	69861704	58841648	- 19%	0.29
cascade	124208334	131567582	+ 6%	0.4
matmult4	43718308	69887073	+ 37%	0.44
sobel	149055072	111187648	- 34%	0.44
imagerapid	108339199	90049942	- 20%	0.51
firsymeven	66659262	102737262	+ 35%	0.54
sort_g	19128011	41412889	+ 54%	0.65
sort_rb	31809351	88791876	+ 65%	0.71
	61850995.6	**76281065.26**		

10.3 Summary

The primary objective of this chapter was to identify and explore various interconnect parameters that affect the overall performance of applications that are mapped to RaPiD. A summary of our findings:

1. Adding registers to the inputs of logic units may improve the performance of pipelined netlists (Section 10.1.1). However, if the number of registers is large, greedily pushing the maximum number of registers into inputs may result in a deterioration of the delay of a netlist (Section 10.1.3).
2. The number and distribution of registered interconnect sites greatly influence overall performance. If there is an insufficient number of interconnect register sites, the pipelined router is forced to find long, circuitous routes that adversely affect both track count and delay (Section 10.1.2). On the other hand, peppering the interconnect structure with register sites may result in an unacceptable area penalty.
3. For reasons similar to those in 2, the number of registers per interconnect site also has to be carefully selected (Section 10.1.3).
4. The flexibility of the interconnect structure has a bearing on the performance of netlists. In Section 10.1.5, we show that architectures that are GPR-poor do not perform well. This is because of increased track counts and longer pipelined routes. On the other hand, architectures that have too many GPRs suffer from an excessive area-penalty.

Chapter 11: Conclusions and Future Work

The subject of this dissertation was the development of placement and routing algorithms that may play important roles in FPGA architecture advancement. The work presented in this document can be divided into two major contributions. The first was the development of a universal FPGA placement algorithm (Independence) that adapts to the target FPGA's interconnect structure. The second contribution was the development of an algorithm (PipeRoute) that can be used to route netlists on pipelined FPGAs. We present our assessment, conclusions, and the scope for future work separately for each contribution.

11.1 Independence

The primary motivation for Independence was the lack of an FPGA placement algorithm that truly adapts to the target FPGA's interconnect structure. We thought that FPGA architecture development efforts would benefit from an adaptive placement algorithm that could be used both as an early evaluation mechanism, as well as a quality goal during CAD tool development. Since the primary goal of an FPGA placement algorithm is to produce a routable placement, our solution to architecture adaptive FPGA placement was centered on using an architecture-adaptive router (Pathfinder) to guide a conventional simulated annealing placement algorithm. Specifically, we used Pathfinder in the simulated annealing inner loop to maintain a fully routed solution at all times. As a result, our cost calculations were based on actual routing information instead of architecture-specific heuristic estimates of routability.

The results presented in Chapter 6 clearly demonstrated Independence's adaptability to island-style FPGAs, a hierarchical FPGA architecture (HSRA), and a domain-specific reconfigurable architecture (RaPiD). The quality of the placements produced by Independence was within 2.5% of the quality of VPR's placements, 21% better than the placements produced by HSRA's place-and-route tool, and within 1% of RaPiD's placement tool. Further, our results also showed that Independence successfully adapts to routing-poor island-style FPGA architectures. When considered together, these results were a convincing validation of using an architecture adaptive router to guide FPGA placement.

In our opinion, Independence's main weakness is its runtime. The algorithm pays a stiff runtime penalty for using a graph-based router in the simulated annealing inner loop. In Chapter 7, we presented ideas

on speeding up Independence (and FPGA routing in general) using the A* algorithm. Again, to preserve adaptability, we concentrated on developing an approach that would work across different FPGA architectures. Memory considerations quickly eliminated a straightforward approach that would pre-compute and store A* estimates for every sink terminal at each interconnect wire. The central idea behind our approach was to cluster interconnect wires that have similar A* estimates, so that all wires that belong to the same cluster could share an entry in the A* estimate table. Thus, the memory requirements of the A* estimate table produced by our clustering technique were comfortably manageable when compared to the straightforward approach.

We evaluated the efficacy of our clustering-based technique on an island–style architecture and a hierarchical architecture (HSRA). The quality of the A* estimates produced by our technique was within 11% of heuristic estimates on the island-style architecture, and 7% better than heuristically calculated estimates for HSRA. We also observed that a low-effort clustering technique might produce estimates that are comparable in quality to both heuristic and clustering-based estimates.

Currently, we see at least three important problems that need to be addressed in the future. We now describe each problem, and present preliminary ideas on how these problems may be solved.

Congestion Weighting Parameter λ: Independence's cost function (Equation 11.1) includes a congestion cost term that represents the extent to which the routing resources are congested in a given placement. The congestion weighting parameter λ is used to vary the importance of changes in congestion with respect to changes in wire cost. In Chapter 6, we studied the effect of the congestion weighting parameter λ on the quality of placements produced by Independence. We reproduce our results in Figure 11-1. While it can safely be concluded that a non-zero value of λ produces placements that either match or are better than placements produced at λ=0, there is no compelling evidence that points to a "magic" value of λ that produces high-quality placements across different architectural styles. Furthermore, the nature of the quality curves also differs on a per-architecture basis.

Equation 11.1: $\Delta C = \Delta WireCost / prevWireCost + \lambda * \Delta CongestionCost / CongestionNorm$

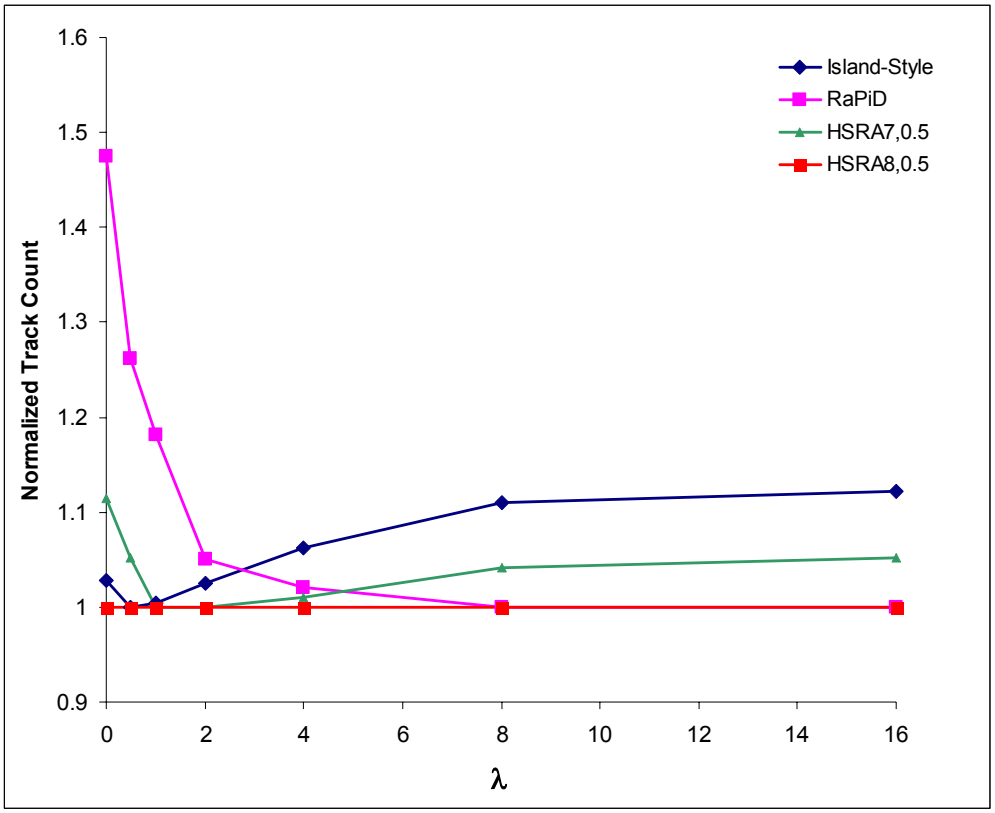

Figure 11-1: The effect of weighting parameter λ on the quality of the placements produced by Independence. The Island-Style curve shows the results of our experiments on an island-style architecture, the RaPiD curve shows results on the RaPiD architecture, the HSRA7,0.5 curve shows results on HSRA, and the HSRA8,0.5 curve shows results on a high-stress instance of the HSRA architecture. The x-axis represents increasing values of the congestion weighting parameter λ, and the y-axis represents normalized track-counts.

Based on the results presented in Figure 11-1, λ is a tuning parameter that needs to be empirically determined on a per-architecture basis. Tuning λ on a per-architecture basis weakens Independence. The underlying premise of Independence is adaptability, and expecting FPGA architects to spend time and resources in determining a sweet spot for λ is probably not consistent with the adaptability premise. Thus, it is imperative that future research focuses on techniques that auto-determine λ.

Our first idea for solving the auto-determination problem for λ is based on dynamic parameter tuning techniques presented in [50]. Specifically, we first select a small representative sub-set of the benchmark netlists. For each netlist in this sub-set, we determine the value of λ which yields the best placement while recording the congestion profile as the placement algorithm progresses. We then use

statistical techniques to determine a target congestion value at the end of each temperature iteration. If the congestion at the end of a temperature iteration exceeds the target value, then we increase λ by an amount proportional to the difference between actual and target congestion. Otherwise, λ is decreased by an amount proportional to the difference between the target and actual congestion. This approach is rooted in feedback-based control systems, and proved very effective in dynamically controlling cost function weighting parameters in [50].

Another technique that may prove useful in auto-determining λ is based on dynamically varying λ in proportion to the difference between the netlist and interconnect bandwidth. At the end of a temperature iteration, the target device is binned into approximately equal sized bins. Each bin is assigned a value of λ proportional to the difference between the region's incident interconnect bandwidth (or Rent's parameter) and the locally resident sub-netlist's Rent's parameter. In the next temperature iteration, the λ used in calculating the change in cost due to a logic block move is equal to the λ assigned to the bin in which the logic block resided. In case the logic blocks involved in the move belong to different λ bins, then a simple, easily calculated function of both λs can be used in cost calculations. The significant difference between the previous dynamic parameter tuning technique and the binning technique is that λ varies *spatially* as well as over time in the binning technique. An efficient technique for estimating the Rent's parameter of an FPGA interconnect structure can be found in [37]. Techniques for estimating the Rent's parameter at the sub-netlist level can be found in [53].

A potentially significant shortcoming of the binning technique is selecting bin-size on a per-architecture basis. We want to ensure that bin-size does not become a tuning parameter itself. To alleviate this problem, we could use a pre-processing step that uses a binary search on a representative sub-set of benchmark netlists to calculate a reasonable bin size. This bin size can then be used as a constant parameter for experiments on the actual benchmark set.

Timing-Driven Cost Function: Independence's current cost function is routability-driven, and does not explicitly consider critical path delay information during the placement process. Timing can be incorporated into Independence's cost formulation by including a timing cost term that is very similar to VPR's *TimingCost* [30]. Independence's cost function can be modified to include timing information as follows:

Equation 11.2: $$\Delta C = \frac{\Delta WireCost}{prevWireCost} + \lambda * \frac{\Delta CongestionCost}{CongestionNorm} + \mu * \frac{\Delta TimingCost}{TimingCost}$$

The timing cost of a placement is calculated using the cost functions in Equation 11.3 and Equation 11.4.

Equation 11.3: $$TimingCost(i, j) = Delay(i, j) * Criticality(i, j)$$

Equation 11.4: $$TimingCost = \sum_{\forall i,j \subset circuit} TimingCost(i, j)$$

In Equation 11.3, *TimingCost(i,j)* represents the timing cost of a net that connects a source-sink pair *(i,j)*, *Delay(i,j)* is the delay of the net, and *Criticality(i,j)* is the criticality of the net. During the placement process, the delay of a net is directly obtained from the currently routed placement. Note that VPR uses a pre-computed delay table to lookup the delay of a net during placement. The pre-computed delay values are calculated using a congestion-unaware timing-driven router, and may be inaccurate in the presence of congestion. Since Independence uses a congestion-driven cost function and maintains a fully routed placement at all times, accurate delay calculations can easily be done on the fly during placement. Further, we update the criticality of moved nets using an incremental static timing analysis approach similar to the approach presented in [35]. Again, this is a difference between our timing-driven technique and VPR's timing-driven placement algorithm. VPR performs static timing analysis only at the beginning of temperature iteration, which may result in undesirably stale criticalities during a temperature iteration. Independence's incremental criticality update approach avoids stale criticalities.

The value of μ in Equation 11.2 can be dynamically determined using a technique that is similar to the dynamic parameter tuning approach for auto-determining congestion parameter λ.

Runtime Considerations: An enhanced implementation of Independence that uses the A* algorithm is currently between 640 – 5000 times slower than VPR. This gap is undesirably large, and calls for further runtime improvements.

Our technique for runtime improvements is based on dividing simulated annealing into two phases over time. We hypothesize that a large share of the annealing runtime is spent in producing routability improvements primarily through wirelength reductions. Thus, during the first part of the annealing process (called the "initial" phase), the cost of a placement is purely a function of wirelength. Full-blown Pathfinder-driven Independence is kicked in only during the second part of the annealing process (called the "final" phase). In the final phase, we use Independence's congestion- and timing-driven cost function to aggressively reduce congestion while producing potential improvements in timing.

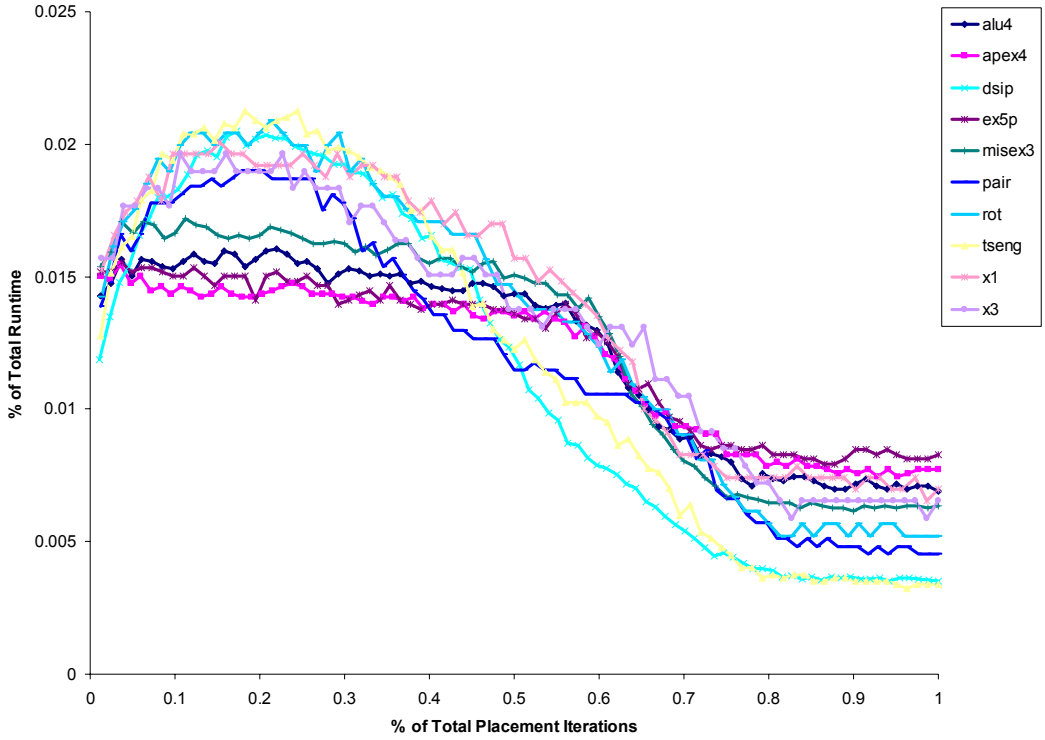

Figure 11-2: Runtime distributions for the ten largest netlists in our island-style benchmark set. The x-axis shows the percentage of the total number of annealing iterations required by Independence to place an individual netlist. The y-axis shows the fraction of the total annealing runtime spent in an annealing iteration.

The point at which the initial phase transitions to the final phase may be determined by running full-blown Independence on a subset of the benchmark netlists and plotting execution time vs. annealing progress. Figure 11-2 shows the variation in per-iteration placement runtime for the ten largest island-style benchmark netlists. The charts in Figure 11-2 show that a large share of Independence's runtime is

due to the first 80 – 90% of the annealing iterations. Based on these trends, we expect that the initial phase will consist of 80 – 90% of the annealing iterations, while the final phase will consist of the remaining 10 – 20% of the annealing iterations. In our current implementation of Independence, the last 10 – 20% of the annealing iterations require 5 – 11% of the total runtime. Assuming that we can develop an architecture-adaptive wirelength-based placer whose runtime is similar to VPR, we could reduce Independence's runtime by potentially 10 – 20 times.

The transition from the initial to the final phase need not be determined only on the basis of runtime. A second approach that may prove useful in determining the phase transition point is based on monitoring quality curves produced by full-blown Independence and the wirelength-based placement algorithm. Specifically, we run Independence and the wirelength-based placer separately on a subset of benchmark netlists. During execution, we periodically halt the placement algorithm and route the current placement using Pathfinder. Note that this will be a complete multi-iteration Pathfinder run that attempts to find a fully routed solution for the current placement. At the end of a Pathfinder run, we record routing data (number of congested resources, number of iterations required by Pathfinder to converge etc) that is representative of the quality of the routing solution. This process of halting placement and attempting routing is repeated throughout the execution of both placement algorithms. At the end of a placement run, the routing data produced by both placement algorithms is plotted vs. annealing progress. Based on our hypothesis, we would expect that the quality curves produced by Independence and the wirelength-based placer would diverge at some point during the annealing process. This divergence point might be a good opportunity to transition from a wirelength-based placer to full-blown Pathfinder-driven Independence.

The bedrock of our approach will be the architecture-adaptive wirelength estimation technique used during the initial phase. Clearly, we do not have the luxury of semi-perimeter estimates, and need to develop techniques that are reasonably accurate and architecture-adaptive. In our opinion, architecture-adaptive wirelength estimates can only be obtained using actual routing information. The important question is when and how often to run the router. Clearly, we cannot use an incremental rip-up and reroute strategy after every placement move without degenerating to Independence's runtime. Instead, we could use an all-pair shortest-path algorithm to pre-compute the cost of a shortest path between any two logic-blocks in the target device and store the costs in a lookup table. This lookup table could then be used during the initial phase to estimate the wirelength of a net.

Note that lookup table-based estimation of the wirelength of a multi-terminal net is not a trivial process. It will be necessary to develop a quick method to approximate the cost of a Steiner tree for the net based on shortest-path costs between any two logic blocks. Approximating Steiner trees using MSTs may be a good starting point, since the number of vertices in the target graph (built from the lookup-table) is equal to the number of logic blocks, and not the number of routing wires and IO terminals in the FPGA's interconnect structure. Still, MST calculation is expensive and the runtime improvement produced by an initial phase that uses MSTs to estimate wirelength might not be compelling. Thus, it might be necessary to develop a faster heuristic for approximating Steiner trees using the pre-computed lookup table.

Wirelength estimation can be approached from a different direction if we relax the adaptability constraint to an architecture-independent constraint. Specifically, we could develop a wirelength estimation technique that is largely independent of the target FPGA's interconnect structure. There are a number of research efforts that have used statistical techniques [11], Rent's rule based techniques [48,61], and flexibility analysis [24] to estimate the wirelength and/or congestion of a placement. It might be possible to use these techniques as a good starting point for developing an architecture-independent wirelength estimation approach. A good survey of the applicability of these techniques can be found in [23].

As a final note, there are a couple important considerations that directly influence the usefulness of our approach. First, the wirelength estimation technique used in the initial phase must allow fast, incremental calculation of changes in cost. Otherwise, the technique might not be appropriate for a simulated annealing placement algorithm that relies on the quick calculation of incremental changes in cost. Second, the transition from the initial phase to the final phase may affect the annealing schedule, since the placement cost profile will abruptly switch from one space to another. Both these issues need to be carefully considered during the development of this potentially faster technique.

11.2 PipeRoute

The second major contribution of the work discussed in this dissertation is the PipeRoute algorithm. In Chapter 9 we pointed out that the routing problem for pipelined FPGAs is different from the conventional FPGA routing problem. In a nutshell, the pipelined routing problem is to find minimum cost routes that satisfy register constraints at sink terminals. The two-terminal pipelined routing problem is NP-Complete, and PipeRoute was the first algorithm that attempted to solve the pipelined routing problem. The algorithm's core is an optimal 1-Register router that is used in a heuristic manner

to build general two-terminal and multi-terminal routes. During PipeRoute's development, we actively tried to minimize the algorithm's reliance on architecture-specific features by using architecture independent abstractions and heuristics. For example, by expressing the target FPGA's interconnect structure as a routing graph, we were able to leverage Pathfinder's congestion resolution mechanism while routing pipelined signals.

PipeRoute's performance was evaluated on the RaPiD architecture. Overall, PipeRoute was able to successfully route pipelined netlists on the RaPiD architecture, and the overhead incurred was less than 20% above a realistic lower bound. We also combined PipeRoute and a RaPiD-specific placer together in an exploratory CAD flow. Chapter 10 presents the results that we obtained on using the CAD flow to explore RaPiD's pipelined interconnect structure. The main findings of the exploration were that the number and location of registers in the interconnect structure may have a significant impact on area-delay product, and that logic units with registered inputs may be a good choice for the RaPiD architecture. Overall, the post-exploration architecture that we found was 19% better than the original RaPiD architecture.

PipeRoute's primary weakness is probably its inability to distribute latency along a route. PipeRoute's greedy heuristics tend to clump registers at the source end of a pipelined signal, resulting in long unpipelined track segments. This weakness may adversely affect critical path delay, and thus it is necessary to explore techniques that explicitly consider the timing effects of register assignment during the routing process.

Clearly, the development of a timing-driven pipelined CAD flow is a strong candidate for future work. However, timing-driven register assignment must be made manageable, possibly by breaking up the task into smaller problems. In our opinion, timing-driven register assignment during the FPGA routing phase is an untenably difficult problem. To begin with, conventional FPGA routing is an NP-Hard problem. Further, even if sharing and timing constraints are entirely disregarded, the two-terminal N_D pipelined routing problem is also NP-Hard. Developing an effective, unified approach that simultaneously finds pipelined routes *and* eliminates sharing *and* minimizes critical-path delay is probably just too hard. We propose the following approaches as potential solutions to timing-driven register assignment:

Register Assignment During Placement: In this approach, register assignment is performed during the placement phase. Pipelining registers in the netlist are freely assigned to both logic units and

interconnect register-sites. The placement algorithm used by this technique is a timing-driven version of Independence. The primary benefits of using Independence are its adaptability and an explicit congestion resolution mechanism. Thus, Independence can be used to solve the register assignment problem on any pipelined FPGA architecture. Furthermore, on routing-poor architectures like RaPiD, Independence's congestion-driven formulation can counter reductions in routing flexibility due to the assignment of pipelining registers to interconnect register-sites.

At the end of the placement phase, the assignments of pipelining registers to logic and interconnect sites are preserved, and Pathfinder is used to route the netlist. This methodology is in direct contrast to the place-and-route flow developed for RaPiD (Chapter 9). In RaPiD's case, the register assignments that were produced by a targeted cutsize-based placement algorithm were in fact removed prior to routing the netlist using PipeRoute. Recall that the register assignments were of extremely poor quality (Table 9-1), and Pathfinder was unable to route most of netlists. We feel that the primary reason for this failure was the inability of our placement algorithm to produce sufficiently decongested placements on RaPiD's inflexible interconnect structure, which shifted the burden of register assignment to the routing algorithm. Now that we have a congestion-driven placement algorithm, it might be a good idea to revisit placement techniques that can solve the register assignment problem.

Decouple Timing From Congestion Resolution: This approach breaks up the timing-driven pipelined routing problem into two sub-problems. Specifically, we first use routability-driven PipeRoute to produce register assignments. We then transform the netlist to expose the register assignments, rip up the routing produced by PipeRoute, and use timing-driven Pathfinder to reroute the transformed netlist. Clearly, this two-step approach separates register assignment from timing improvements, and is more manageable from an implementation perspective.

Merely decoupling register assignment from timing improvements might not adequately address PipeRoute's register-clumping shortcoming. In order to distribute registers more evenly along pipelined routes, an incremental "de-clump" step that redistributes registers along pipelined routes might be necessary. Specifically, at the end of a successful PipeRoute run, the de-clump step pulls apart register clumps by spreading out register assignments on a pipelined route. Note that de-clump does not involve any rip-up and reroute; register re-assignments are localized to register-sites that are already part of a pipelined route. Clearly, the de-clump operation relies on the availability of extra, unused register-sites along a pipelined route. We feel that this assumption may be reasonable given that a number of FPGA

architectures have track-domains, and the switches in a track domain are either entirely registered or unregistered [15,52,46].

All in all, the overall flow of this technique is routability-driven PipeRoute + de-clump + timing-driven Pathfinder. Placements can be obtained using an Independence-based placement algorithm that exposes pipelining registers during placement. The only difference is that register assignments are removed post-placement, and routability-driven PipeRoute is used to reassign pipelining registers during the routing phase.

Bibliography

[1] M. Alexander, J. Cohoon, J. Ganley, G. Robins, " Performance-Oriented Placement and Routing for Field-Programmable Gate Arrays", *European Design Automation Conference*, pp. 80 – 85, 1995.

[2] Altera Inc., "Stratix™ Devices", available at http://altera.com/literature/lit-stx.jsp

[3] Altera Inc., "Stratix II™ Devices", available at http://altera.com/literature/lit-stx2.jsp.

[4] J. Beetem, "Simultaneous Placement and Routing of the LABYRINTH Reconfigurable Logic Array", In Will Moore and Wayne Luk, editors, *FPGAs*, pp. 232-243, 1991.

[5] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", *7th International Workshop on Field-Programmable Logic and Applications,* pp 213-222, 1997.

[6] V. Betz, J. Rose and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Boston, MA:1999.

[7] V Betz, "The FPGA Place-and-Route Challenge", at http://www.eecg.toronto.edu/~vaughn/

[8] G. Boriello, C. Ebeling, S Hauck, S. Burns, "The Triptych FPGA Architecture", *IEEE Transactions on VLS Systems*, Vol. 3, No. 4, pp. 473 – 482, 1995.

[9] S. Cadambi, S. Goldstein, "Efficient Place and Route for Pipeline Reconfigurable Architectures," *International Conference on Computer Design,* pp. 423 – 429, 2000.

[10] Y.W. Chang and Y.T. Chang, "An Architecture-Driven Metric for Simultaneous Placement and Global Routing for FPGAs", *ACM/IEEE Design Automation Conference*, pp. 567-572, 2000.

[11] C. Cheng, "RISA: Accurate and Efficient Placement Routability Modeling", *IEEE/ACM International Conference on Computer Aided Design*, pp. 690 – 695, 1994.

[12] K Compton, S Hauck, "Totem: Custom Reconfigurable Array Generation", *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp , 2001.

[13] K Compton, A Sharma, S Phillips, S Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", *International Conference on Field-Programmable Logic and Applications*, pp 59 – 68, 2002.

[14] T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA:1990.

[15] D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Twentieth Anniversary Conference on Advanced Research in VLSI,* pp 23-40, 1999.

[16] A. DeHon, “Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don’t really want 100% LUT utilization),” *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1999.

[17] C. Ebeling, D. Cronquist, P. Franklin, “RaPiD - Reconfigurable Pipelined Datapath”, *6th International Workshop on Field-Programmable Logic and Applications,* pp 126-135, 1996.

[18] C. Fiduccia, R. Mattheyses, "A Linear-Time Heuristic for Improved Network Partitions", *ACM/IEEE Design Automation Conference*, pp. 241-247, 1982.

[19] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Taylor, “PipeRench: A Reconfigurable Architecture and Compiler”, *IEEE Computer*, pp. 70 – 76, 2000.

[20] S. Hauck, T. Fry, M. Hosler, J. Kao, “The Chimaera Reconfigurable Functional Unit”, *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 87 – 96, 1997.

[21] S Hauck, *Multi-FPGA Systems*, PhD Thesis, University of Washington, Dept. of Computer Science and Engineering, 1995.

[22] N. Kafafi, K. Bozman, S Wilton, “Architectures and Algorithms for Synthesizable Embedded Programmable Logic Cores”, *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 3 – 11, 2003.

[23] P. Kannan, S. Balachandran, D. Bhatia, “On Metrics for Comparing Routability Estimation Methods for FPGAs”, *ACM/IEEE Design Automation Conference*, pp 70 – 75, 2002.

[24] P. Kannan, S. Balachandran, D. Bhatia, “fGREP: Fast Generic Routing Demand Estimation for Placed FPGA Circuits”, *International Conference on Field Programmable Logic and Applications*, pp 37 – 47, 2001.

[25] G. Karypis, Vipin Kumar, “Multi-level k-way Hypergraph Partitioning”, *ACM/IEEE Design Automation Conference*, pp. 343 – 348, 1999.

[26] S. Kirkpatrick, C. Gelatt Jr., M. Vecchi, “Optimization by Simulated Annealing”, *Science*, 220, pp. 671-680, 1983.

[27] C. Leiserson, and J. Saxe, “Retiming Synchronous Circuitry”, *Algorithmica*, 6(1):5-35, 1991.

[28] J. MacQueen, “Some Methods for Classification and Analysis of Multivariate Observations”, *5th Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281 – 297, 1967.

[29] P. Maidee, C. Ababei, K. Bazargan, “Fast Timing-Driven Partitioning-based Placement for Island Style FPGAs”, *ACM/IEEE Design Automation Conference*, pp. 598 – 603, 2003.

[30] A. Marquardt, V. Betz and J. Rose, “Timing Driven Placement for FPGAs”, *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 203 – 213, 2000.

[31] A. Marquardt, V. Betz and J. Rose, “Speed and Area Tradeoffs in Cluster-Based FPGA Architectures”, *IEEE Transactions on VLSI Systems*, Vol. 8, No. 1, pp. 84 – 93, 2000.

[32] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, B. Hutchings, “ A Reconfigurable

Arithmetic Array for Multimedia Applications", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 135 – 143, 1999.

[33] L. McMurchie and C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 111-117, 1995.

[34] C. Mulpuri, S. Hauck, "Runtime and Quality Tradeoffs in FPGA Placement and Routing", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 29 – 36, 2001.

[35] S. Nag and R. Rutenbar, " Performance-Driven Simultaneous Placement and Routing for FPGAs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, Vol. 17, No. 6, pp. 499 – 518, 1998.

[36] M. Papaefthymiou, "Understanding Retiming through Maximum Average-Weight Cycles", *ACM Symposium on Parallel Algorithms and Architectures*, pp. 272 – 277, 1991.

[37] G. Parthasarathy, M. Marek-Sadowska, A. Mukherjee, A. Singh, "Interconnect Complexity-Aware FPGA Placement Using Rent's Rule", *IEEE/ACM International Workshop on System Level Interconnect Prediction*, pp. 115 – 121, 2001.

[38] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, Kluwer Academic Publishers, Boston, MA: 1988.

[39] U. Seidl, K. Eckl, F. Johannes, "Performance-directed Retiming for FPGAs using Post-placement Delay Information", *Design Automation and Test in Europe*, pp. 770 – 775, 2003.

[40] A. Sharma, "Development of a Place and Route Tool for the RaPiD Architecture", *Master's Project, University of Washington*, December 2001.

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

[42] A. Sharma, C. Ebeling, S. Hauck, "PipeRoute: A Pipelining-Aware Router for Reconfigurable Architectures", to appear in *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 2006.

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

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

[45] N. Shenoy, R. Rudell, "Efficient Implementation of Retiming", *IEEE/ACM International Conference on Computer Aided Design*, pp. 226 – 233, 1994.

[46] D. Singh, S. Brown, "The Case for Registered Routing Switches in Field Programmable Gate Arrays", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 161-169, 2001.

[47] D. Singh, S. Brown, "Integrated Retiming and Placement for Field Programmable Gate Arrays", *ACM/SIGDA International Symposium on* Field-*Programmable Gate Arrays*, pp 67-76, 2002.

[48] D. Stroobandt and J Van Campenhout, "Accurate Interconnect Length Estimations for Predictions Early in the Design Cycle", *VLSI Design, Special Issue on Physical Design in Deep Submicron*, 10(1):1-20, 1999.

[49] J. Swartz, V. Betz and J. Rose, "A Fast Routability-Driven Router for FPGAs", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 140 – 149, 1998.

[50] W. Swartz and C Sechen. "New Algorithms for the Placement and Routing of Macrocells", *IEEE International Conference on Computer Aided Design*, pp. 336 – 339, 1990.

[51] N. Togawa, M. Yanigasawa, T. Ohtsuki, "Maple-opt: A Performance-Oriented Simultaneous Technology Mapping, Placement, and Global Routing Algorithm for FPGAs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, Vol. 17, No. 9, pp. 803 – 818, 1998.

[52] 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 International Symposium on Field-Programmable Gate Arrays*, 1999.

[53] H. Van Marck, D. Stroobandt and J. Van Campenhout, "Toward an Extension of Rent's Rule for Describing Local Variations in Interconnect Complexity", *International Conference for Young Scientists*, pp. 136 –141, 1995.

[54] B Von Herzen, "Signal Processing at 250 MHz Using High-Performance FPGAs", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 62 – 68, 1997.

[55] P. Wang and K. Chen, "A Simultaneous Placement and Global Routing Algorithm for an FPGA with Hierarchical Interconnection Structure", *International Symposium on Circuits and Systems*, pp. 659 – 662, 1996.

[56] N. Weaver, J. Hauser, J. Wawrzynek, "The SFRA: A Corner-Turn FPGA Architecture", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 3 – 12, 2004.

[57] N. Weaver, Y. Markovskiy, Y. Patel, J. Wawrzynek, "Post-Placement C-slow Retiming for the Xilinx Virtex FPGA", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 185 – 194, 2003.

[58] T. Wong, "Non-Rectangular Embedded Programmable Logic Cores", *M.A.Sc. Thesis*, *University of British Columbia*, May 2002.

[59] Xilinx Inc, "Virtex-4 Overview", at http://www.xilinx.com/products/virtex4/overview.htm.

[60] Xilinx Inc, "VirtexII Platform FPGA Features", at http://www.xilinx.com.

[61] X. Yang, R. Kastner, M Sarrafzadeh, "Congestion Estimation During Top Down Placement", *International Symposium on Physical Design*, pp. 164 – 169, 2001.

Vita

Personal
Akshay Sharma, born January 3, 1977.

Education
Ph.D., Electrical Engineering, University of Washington, Seattle WA, 2005.
Thesis: Place and Route Techniques for FPGA Architecture Advancement.
Advisors: Scott Hauck and Carl Ebeling.

M.S., Electrical Engineering, University of Washington, Seattle WA, 2001.
GPA 3.85.

B.E., Electronics & Communications Engineering, University of Delhi, New Delhi (India), 1999.
Grade 80% (With Distinction).

Recognition
University of Washington Electrical Engineering Outstanding Research Assistant Award, 2005.
Nominated for an Intel Fellowship by University of Washington Electrical Engineering, 2004.
Nominated for University of Washington Electrical Engineering Outstanding Teaching Award, 2004.

Publications
K. Compton, A. Sharma, S. Phillips, S.Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", *International Conference on Field Programmable Logic and Applications*, 2002.

A. Sharma, C. Ebeling, S. Hauck, "PipeRoute: A Pipelining-Aware Router for FPGAs", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2003.

A. Sharma, K. Compton, C. Ebeling, S. Hauck, "Exploration of Pipelined FPGA Interconnect Structures", *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2004.

S. Phillips, A. Sharma, S. Hauck, "Automating the Layout of Reconfigurable Subsytems Via Template Reduction", *International Conference on Field-Programmable Logic and Applications*, 2004.

A. Sharma, C. Ebeling, S. Hauck, "Architecture Adaptive Routability-Driven Placement for FPGAs", *International Conference on Field-Programmable Logic and Applications*, 2005.

K. Eguro, S. Hauck, A Sharma, "Architecture-Adaptive Range Limit Windowing for Simulated Annealing FPGA Placement", *ACM/IEEE Design Automation Conference*, 2005.

A. Sharma, C. Ebeling, S. Hauck, "PipeRoute: A Pipelining-Aware Router for Reconfigurable Architectures", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006.