UNIVERSITY OF CALIFORNIA
RIVERSIDE

Parallel Routing for Field Programmable Gate Arrays With Sparse Intra-Cluster Routing Crossbars

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

by

Yehdhih Ould Mohamed Moctar

June 2014

Dissertation Committee:

Dr. Philip Brisk, Chairperson
Dr. Walid Najjar
Dr. Frank Vahid
Dr. Qi Zhu

The Dissertation of Yehdhih Ould Mohamed Moctar is approved:

_____

_____

_____

_____
Committee Chairperson

University of California, Riverside

ACKNOWLEDGMENTS

Thanks are due first to my advisor, Philip Brisk, for the guidance, technical advice, and moral support he provided throughout my Ph.D. Philip's enthusiasm and confidence in me were invaluable during my research, and I learned a great deal about how to conduct research him. I would also like to thank the other members of my dissertation committee; Professor Walid Najjar, Professor Frank Vahid, and Dr. Qi Zhu for accepting to be on my dissertation committee and for giving me valuable suggestions.

I would also like to thank the other members of Philip's group – Daniel Grissom, Jeff Mc Daniel, Joseph Tarango, Kennet O'Neal -- for the many insightful discussions they provided. Special thanks are due to Professor Walid Najjar's students; Robert Halsted, Roger Moussaoui, and Robert Skyler for giving me access to their GPU and Multicore servers; that were critical for parallelizing the FPGA router.

I would also like to thank my research collaborators; Professor Guy Lemieux from the University of British Columbia, Professor Keshav Pingali and his student Donald Nguyen from the University of Texas, Austin, and Professor Paolo Ienne and his students Hadi-Parandeh Afshar and Nithin George from l'Ecole Polytechnique Federale de Lausanne (EPFL) for their help in using the Galois deterministic scheduler. I am also very thankful to Dr. Wolfgang Roesner (IBM Fellow) and Vasantha Vuyyuru and the rest of their IBM Electronic Design Automation Group, for giving me the opportunity to present my research and for their valuable advice; and to Dr. Kees Vissers from Xilinx for his insightful advice.

I am very grateful for the support, encouragement and patience of my wife, Oum El Banina, and my son Mohamed Moctar El Yadaly, throughout the years I worked toward my Ph.D. Many thanks are due to my brothers and sisters, and all the members of my extended family. I owe my friends and my extended family a debt of gratitude for their support and encouragement throughout all my years of schooling.

Finally, I appreciate the financial support for this project provided by the Department of Defense in the form of a SMART scholarship.

DEDICATION

To my parents; Aicha Mint Sidiya & Mohamed Moctar El Yadaly. Who always stood behind me and knew I would succeed. Gone now but never forgotten. I will miss them always and love them forever. Thanks for all you did.

ABSTRACT OF THE DISSERTATION

Parallel Routing for Field Programmable Gate Arrays With Sparse Intra-Cluster Routing Crossbars

by

Yehdhih Ould Mohamed Moctar

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2014
Dr. Philip Brisk, Chairperson

Routing is the most time consuming step of the process of synthesizing an electronic design on a Field Programmable Gate Array (FPGA). It involves the creation of a Routing Resource Graph (RRG); a large data structure representing the physical architecture of the FPGA. In this work, we first introduce two scalable routing heuristics for FPGAs with sparse intra-cluster routing crossbars: SElective RRG Expansion (SERRGE), which compresses the RRG, and dynamically decompresses it during routing, and Partial Pre-Routing (PPR), which locally routes all nets in each cluster, and routes global nets afterwards. Our experiments show that: (1) PPR and SERRGE converge faster than a traditional router using a fully-expanded RRG; (2) they both achieve better routability than the traditional router, given a limited runtime budget; and

(3) PPR uses far less memory and runs much faster than SERRGE, making it ideal for high capacity FPGAs.

We then introduce a new dynamic-multiplexing based hybrid logic blocks that can be configured to operate as regular configurable logic blocks, or to implement shifting operations required for mantissa alignment and normalization in floating point operations. We show that: (1) the number of CLBs required for shifting operations is reduced by 67%, and if shifting is not required, these hybrid logic blocks can be configured for normal operation, so no functionality is sacrificed; (2) the area overhead incurred by these modifications is small, and (3) there is no negative impact in terms of clock frequency or routability for benchmarks that do not use floating point shifting.

Finally, we investigate the parallelization of FPGA routing on Multicore, shared memory CPUs, using a speculation-based approach. The router is a parallel implementation of PathFinder, which is the basis for most commercial FPGA routers. Our results demonstrate scalability for large benchmarks and that the amount of available parallelism depends primarily on the circuit size, not the inter-dependence of signals. Our experimental results show an average speedup of approximately 5.5x in comparison to the single threaded router implemented in the publicly available Versatile Place and Route (VPR) framework [46].

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. INTRODUCTION

1.1 MOTIVATION

Since their inception in the mid-80s, Field Programmable Gate Arrays (FPGAs) have experienced an exponential growth at a rate faster than the rest of the semiconductor industry [14]. They evolved from as little as 64 LUTs (Xilinx XC2064)[11] to as much as over a million LUTs, and a large number of hard-wired macro blocks (embedded memories, DSP blocks, and embedded processors), and high speed IOs in the latest ALTERA and XILINX devices (Stratix 10 and Virtex 7) [3] [76]; an increase of more than 10,000 times in their logic capacity.

These FPGA devices are being used across many industries to implement highly complex system-on-chip (SoC) designs [14], and in many cases as accelerators for many computationally intensive applications. There is also considerable interest in using FPGAs to accelerate scientific applications that are dominated by floating-point computations. Commercial FPGAs currently use dedicated hard blocks such as DSPs and embedded processors for floating point operations. However, applications that are not floating-point intensive will be unable to use these blocks. In this thesis, we investigate an alternative strategy that leverages the abundant spatial parallelism in FPGAs, and provides an optimized floating-point datapath to minimize the size of each operator, as doing so maximizes the number of operators that can be synthesized onto a device of fixed size; which, in turn, maximizes throughput.

To support the design of these complex FPGA architectures, computer-aided design (CAD) tools play a decisive role in delivering high-performance, high-density, and low power design solutions using these high-end FPGAs. However, the process of synthesizing an industrial-scale circuit on a high-capacity commercial FPGA can easily take hours, days or even weeks, depending on the size of the circuit and the target device. This long runtime is one of the biggest concerns to FPGA architects and circuit designers, and a major impediment to the adoption of FPGAs as mainstream accelerators of many computationally intensive applications. Prior research in this area has identified Routing to be the most time consuming step of the CAD process[24]. Consequently, in this thesis we investigate two aspects of the routing problem; routing for FPGAs that employ sparse crossbars in their intra-cluster routing, and parallelizing the router on multicore, shared memory systems.

The focus of this thesis is therefore on investigating logic block architecture to provide better support for floating-point shifters, and exploring different ways of parallelizing and reducing the run time and memory footprint of the routing model for FPGAs with intra-cluster routing crossbars.

1.2 CHALLENGES

We are interested in island style FPGAs, consisting of an array of configurable logic blocks, which implement the logic of the circuit, and the programmable routing which allows the logic blocks to be inter-connected. In this thesis we investigate two issues

related to the FPGA routing model, and one issue related to the logic block architecture.

The first issue we investigate is routing in FPGAs that employ sparse crossbars in their intera-cluster routing. The process of routing a design on an FPGA is often lengthy and memory-intensive. In particular, the *Routing Resource Graph (RRG)* of a commercial-grade FPGA can be very large, due to the inordinate quantity of uniquely programmable routing resources that are present in the architecture.

One of the significant contributors to overall RRG size is the presence of sparse intra-cluster routing crossbars within the FPGA routing network [20][42][43][73]. In early FPGA generations, intra-cluster routing crossbars were fully connected, which allowed the RRG to implicitly represent them. When the crossbars become sparse, the implicit representation is no longer accurate, so the need to explicitly enumerate their connectivity significantly enlarges the overall RRG size.

Prior research in this area addressed the sparness of crossbars and attempted to provide a model for estimating the routability of sparse crossbars, using analytical and architectural approaches. It didn't however, investigate how routing is performed in FPGAs with intra-cluster routing crossbars.

The second issue we investigate is the enhancement of the configurable logic block to provide better support for floating-point shifting operations. This involves the introduction of dynamic multiplexing in the intra-cluster routing fabric of certain logic blocks. Dynamic multiplexing gives the FPGA programmer the ability to configure the SRAM bits of a multiplexer, after they have been set by the routing model. Even tough this technique can be beneficial for circuits that require a significant amount of

multiplexing, it comes at a non-negligible cost, and creates new challenges for physical design tools. The following issues must be addressed in order to justify the inclusion of dynamic multiplexers in an FPGA fabric:

(1) Given that FPGA routing networks consume as much as 90% of on-chip area [7], is the area overhead of replacing static multiplexers justifiable?

(2) When multiplexers are configured for dynamic control, how can the router overcome the lack of flexibility arising from the fact that input signals must be routed to multiplexer inputs in a pre-specified order? How is routability achieved in the general case?

(3) How are the dynamic control bits generated, and how are they routed?

Our work address these issues using a CAD-driven architectural approach that evaluates the tradeoffs in terms of area and delays associated with the introduction of dynamic multiplexing into the intra-cluster routing of the FPGA.

The last issue addressed in this thesis is the reduction of CAD runtime through the parallelization of FPGA routing. Among the different stages in a typical CAD flow, *routing* is often the most significant in terms of runtime and performance, since it directly affects the achievable clock frequency. Practically, all commercial FPGA routers have their origins in the *PathFinder* algorithm, introduced in 1995 by McMurchie and Ebeling [50]. PathFinder employs an algorithmic approach called *negotiated congestion*, in which individual nets in the user circuit are allowed to share FPGA routing resources; as the algorithm proceeds, the negotiation process ensures that at most one net is routed along each resource.

The challenges associated with parallelizing PathFinder depend primarily on the strategy used, and the underlying hardware architecture. At a coarse-grained level, one can route the nets in parallel and use a shared congestion map to control the negotiation process. The biggest challenge of this approach is to introduce an efficient mechanism to handle contention and communication among concurrent threads. At the fine-grained level, nets can be routed serially, while parallelizing the maze expansion of individual nets. As the maze expansion is typically a directed breadth-first or A* search on the RRG, the main concern is finding a mechanism for implementing the priority queue such that it is optimized for multithreading operations.

In this work we investigate parallelizing at both levels (coarse-grained & fine-grained) on multicore, shared memory CPU architectures.

1.3 THESIS APPROACH & ORGANIZATION

The research approach used in this thesis is experimental; we have selected circuits from the 2005 IWLS benchmark suite [28]. We used ABC for logic optimization and technology mapping [6]. We then incorporated our enhancements and modifications into the academic Versatile Place and Route (VPR) physical design tool [46], and use it to pack, place and route our design circuits. We evaluate our approaches using FPGA architectures from the iFAR repository [33][34] made public by the University of Toronto, and report area models and delay estimates provided by VPR.

The next chapter provides background information and details some of the previous work in both the relevant areas of CAD and FPGA architecture. Chapter 3 describes the

two approaches we introduced to perform routing in FPGAs that employ sparse crossbars in the intra-cluster of their logic blocks. In Chapter 4 we detail the enhancement we have made to dynamically reconfigure the logic blocks in order to reduce the cost of floating point mantissa alignment and normalization in FPGAs. Chapter 5 presents our speculation-based model for parallelizing an FPGA router on multicore, shared memory CPU architecture. The final chapter summarizes the thesis conclusions and provides suggestions for future work.

Chapter 2. BACKGROUND AND RELATED WORK

This chapter provides background information about FPGA architecture, and the CAD flow used to automatically map circuits into FPGAs that employ sparse crossbars in their Configurable Logic Blocks (CLBs). It also presents the fundamental concepts and techniques of parallel computing essential to parallelizing an FPGA router on multi-core, shared memory CPU architectures, and briefly describes the prior work relevant to this thesis.

2.1 FPGA ARCHITECTURE

The architecture of a modern FPGA as depicted in **Figure 2-1**, consists of an array of inter-connected programmable logic blocks surrounded by programmable input/output (I/O) blocks. The main components of this architecture are the configurable logic blocks (CLBs), the memory and multiplier blocks, the I/O blocks, and the programmable routing fabric. Mapping a design on an FPGA involves configuring the programmable logic blocks to implement the logic required by the design and using the I/O blocks as input and output pads for interfacing with external devices. The programmable routing fabric allows the logic blocks and I/O blocks to be programmatically interconnected.

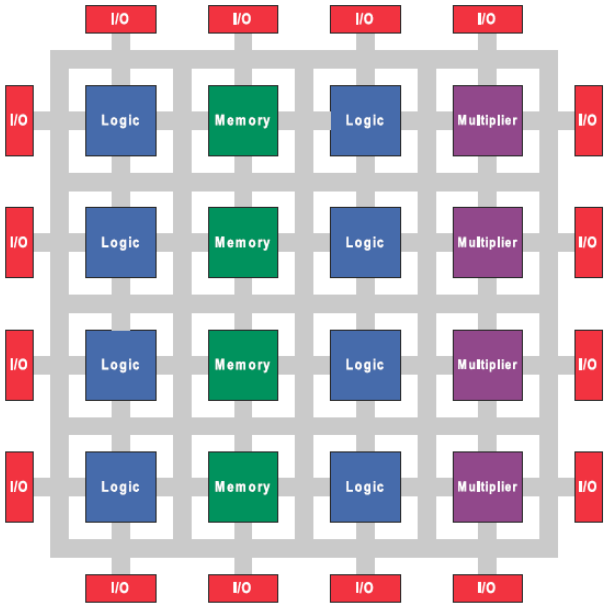


Figure 2-1: Generic Island-style FPGA[1].

FPGAs are programmed by configuring the switches of the CLBs and the programmable routing fabric, using one of three techniques; SRAM-based, Flash-based, or Anti-fuse [7]. Among these techniques, SRAM-based is the most popular, and it is the technology of choice of Xilinx and Altera; the two major commercial FPGA vendors[2][75]. Consequently, in this thesis we will only investigate the SRAM-based FPGAs. Basically, this technology makes FPGAs programmable by using SRAM cells to control pass transistors, multiplexers and tri-state buffers in order to configure the programmable routing and logic blocks as required. These SRAM cells are stored in static memory as an array of latches. **Figure 2-2** shows these SRAM-based switches. The next section briefly describes how the logic blocks are configured using the SRAM cells. Sections 2.1.2 and 2.1.3 then describe some of the different architectures and prior

research into FPGA logic blocks and routing, respectively.

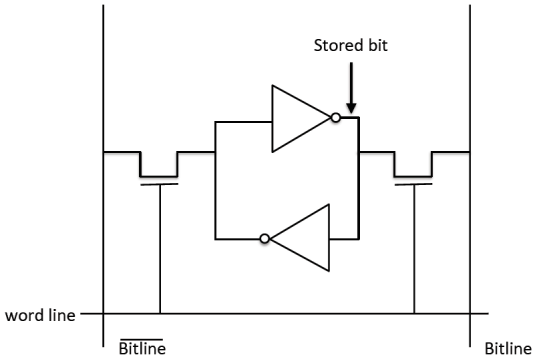


Figure 2-2: SRAM bit cell.

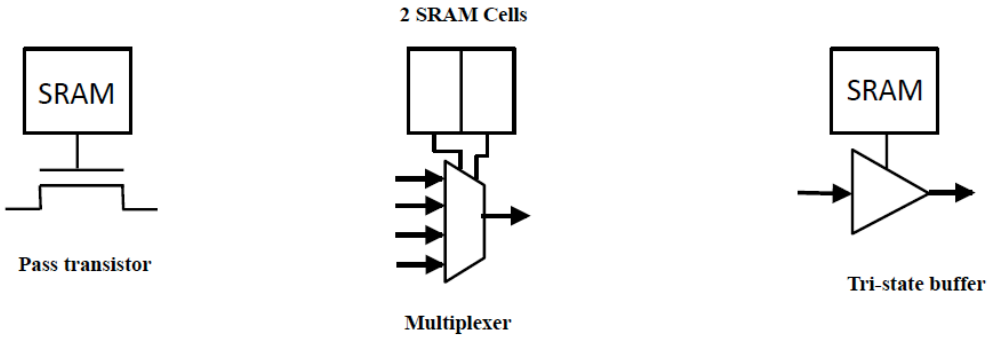


Figure 2-3: Types of Programmable switches used in SRAM-based FPGAs[7].

2.1.1. CONFIGURABLE LOGIC BLOCK (CLB)

The purpose of a logic block in an FPGA is to provide the basic computation and storage elements used in digital logic systems. Even though many different logic blocks have been used to provide this functionality, most current commercial FPGAs are using configurable logic blocks based on look-up tables (LUTs)[3][4][76]. In this CLB

architecture, the atomic unit is a K-input LookUp Table (K-LUT); **Figure 2-3** shows how a 2-input LUT can be implemented in an SRAM-based FPGA, a k-input LUT requires 2^k SRAM cells and 2^k-input multiplexers. A k-input LUT can be configured to implement any K-input, 1-output logic function; one simply programs the 2^k SRAM cells to be the truth table of the desired function[7].

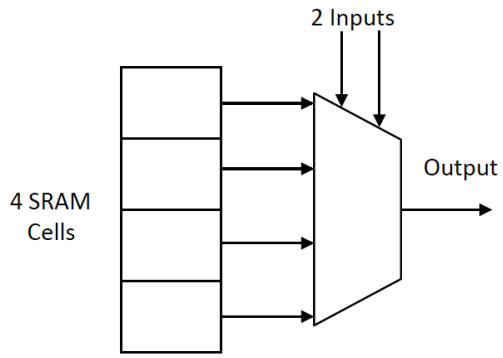


Figure 2-4: SRAM-based implementation of a 2-inpout lookup table (LUT)

A Basic Logic Element (BLE) is a K-LUT coupled with a bypassable flip-flop, as shown in Figure 2-5. BLEs are clustered in groups called Configurable Logic Blocks (CLBs). Each CLB contains N BLEs, along with an intra-cluster routing crossbar. In early FPGAs, the intra-cluster routing crossbar was fully connected; in more recent devices, it has become sparse [43]. Section 2.1.3 provides detailed discussion about the architecture of the routing fabric; including the intra-cluster routing crossbars. The next section presents some modifications to the intra-cluster routing multiplexers, which can reduce the number of CLBs required to implement floating point adders by 67%.

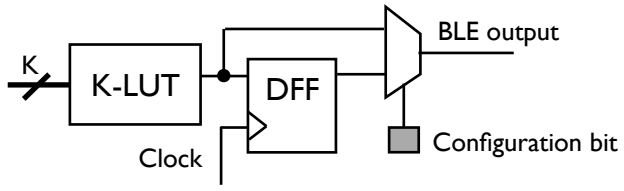


Figure 2-5: The Basic Logic Element (BLE) of an FPGA.

2.1.2. HYBRID CONFIGURABLE LOGIC BLOCKS

We have introduced a new type of logic blocks that can be configured to operate as regular configurable logic blocks, or to implement shifting operations required for mantissa alignment and normalization in floating point operations. If the logic block is configured for shifting operations, the routing MUXes of the intra-cluster routing crossbar are used in conjunction with the LUTs of the logic block to implement shifting operations. This configuration is made possible by the use of dynamic multiplexing; a technique that gives the FPGA programmer a direct control over the intra-cluster routing MUXes. This is a deviation from traditional FPGA Multiplexing that place the Multiplexers under static control, i.e. once configured after routing, they cannot be reconfigured until the FPGA is reprogrammed. The next section describes this technique and give a motivating example showing the potential area savings resulting from using these hybrid blocks to implement floating point shifters.

2.1.2.1. STATIC VS DYNAMIC MULTIPLEXING

Statically controlled MUX (S-MUX) is a multiplexer controlled by the FPGA bitstream, and is not generally accessible to the programmer. In contrast, dynamically

controlled MUX (D-MUX) is a multiplexer that can be configured by the programmer. An SD-MUX is a multiplexer that can be configured by the programmer to be either S-MUX or a D-MUX. One way to make S-MUX accessible to the FPGA programmer is to add dynamic configurability as shown in **Figure 2-6** (b). Figure **Figure 2-6**(a) shows a traditional FPGA MUX that is statically controlled by the configuration bitstream; in Figure **Figure 2-6** (b) the static MUX is extended so that it can be dynamically configured as an S-MUX or a D-MUX. In the next section we show that this hybrid configuration of the MUXes can convert the CLB to implement a shifter. So, converting the S-MUXes of the intra-cluster routing of a CLB into SD-MUXes results in a new, hybrid CLB architecture accessible by the programmer. The next section presents a motivating example showing how this hybrid CLB can be configured to implement a floating-point shifter significantly reducing the number of CLBs.

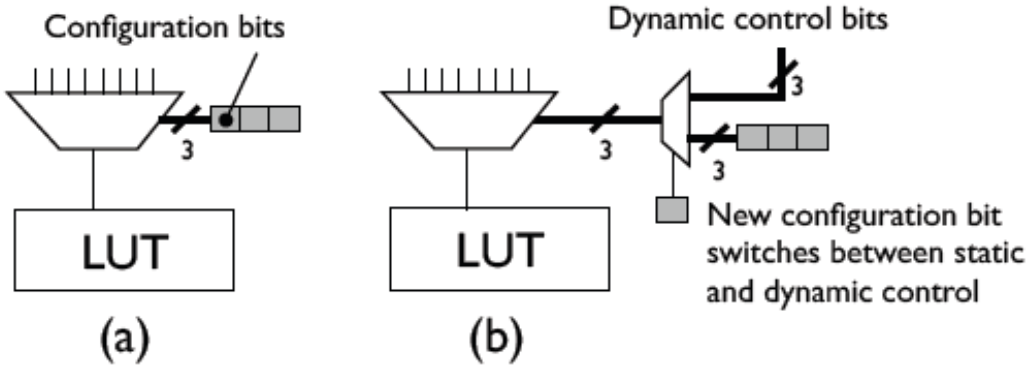


Figure 2-6: A Multiplexer in a traditional FPGA routing network.

2.1.2.2. CONVERTING HYBRID BLOCKS TO IMPLEMENT SHIFTERS

Shifters implemented as a Multiplexer-based logic maps inefficiently onto LUTs [52]. This inefficiency can however be reduced significantly by using the intra-CLB routing MUXes in conjunction with LUTs to implement the shifters required for mantissa alignment and normalization in floating-point addition. As a motivating example, consider the 8:1 multiplexer shown in **Figure 2-6**(a), which drives one input of a LUT; the other LUT inputs are driven by similar multiplexers, which are not shown. Three FPGA configuration bits drive the multiplexer's selection inputs. The purpose of this multiplexer is to provide some flexibility to the FPGA CAD tools—in particular, the router -when synthesizing a circuit onto the FPGA. In this case, there are 8 physical wires within the FPGA that can connect to this LUT input, via the multiplexer. One signal must route to that particular LUT input, and the router is given 8 possible wires to use. Once the route is complete, the configuration bits are set to select the chosen wire. This configuration is *static,* i.e., it does not change until the FPGA is reprogrammed. As there is no possibility to dynamically drive the selection inputs of this multiplexer, there is no possibility for the user to utilize it as an actual 8:1 multiplexer. As it is not architecturally visible, the typical user—who is not an FPGA architect—will be completely unaware of its existence.

As illustrated in **Figure 2-6** (b), a *Static-Dynamic Multiplexer (SD-MUX)* can be configured for either static or dynamic control. A 2:1 multiplexer now drives the configuration inputs of the 8:1 multiplexer. The 2:1 multiplexer can select either the control bits or a set of wires that are available to the user to provide dynamic control. An

extra configuration bit drives the selection input of the 2:1 multiplexer, thus allowing the user to configure the 8:1 multiplexer to provide either static or dynamic control. This basic idea easily generalizes to a multiplexer with any number of inputs, as long as a sufficient number of control bits are provided. When the SD-MUX is configured to provide static control, one signal can be routed to any of the 8 multiplexer inputs, and the configuration bits are set accordingly, as noted earlier, this provides flexibility to the router, as there is fierce competition for routing resources. When the SD-MUX is configured as a dynamic multiplexer, as shown in **Figure 2-7**(b), 8 signals are routed to the 8 multiplexer inputs in pre-specified order; e.g., if the user logic expects the multiplexer to select signal *x* when the selection bits are 010, then *x* must be routed to multiplexer input 010 in order to preserve this functionality; thus, the flexibility afforded to the router in the static case is sacrificed.

If we assume that the multiplexers in the routing network are 27:1 or larger, then 24 of them can implement mantissa alignment, and 27 can implement normalization. If we ignore the other LUT inputs, and configure the LUT to implement the identity function, then these two shifters can be implemented using 51 LUTs: a savings of 66.7% over the LUT-based implementation. Chapter 2 presents a solution that realizes this best-case savings.

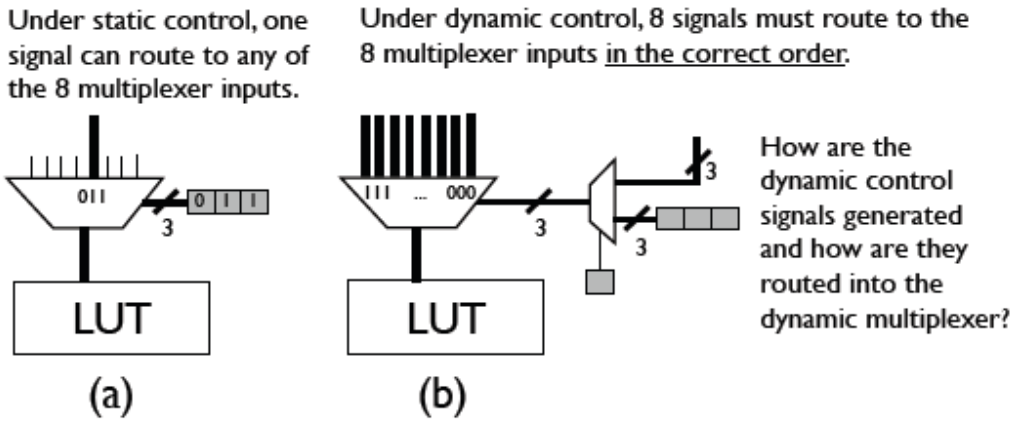


Figure 2-7: Routing challenges of dynamic multiplexers.

The goal of converting CLBs to implement shifters is to reduce the cost of mantissa alignment and normalization in floating-point operations. One alternative is to integrate floating-point units as hard blocks [5][30][36]; however, applications that are not floating-point intensive will be unable to use these blocks. To date, FPGA vendors do not sell device families with dedicated blocks for floating-point applications.

Beauchamp et al. [5] advocate integrating hard shifters or 4:1 multiplexors in parallel with FPGA logic; however, when the shifters are not used, the nearby routing resource are wasted; and when the 4:1 multiplexors are used, significant routing resources are still required to form large shifters.

Shifters and multiplexers can be synthesized onto multipliers in the DSP blocks [30][21], and Xilinx has added 17-bit barrel shifters to their DSP48E1 blocks [72]; however, a DSP block used for shifting, cannot perform other operations. Benchmarks that require multiplication and shifting can still benefit from FPGAs containing DSP blocks and macro-cells. Floating-point datapath compilers use arithmetic transformations

to synthesize floating-point operations efficiently on FPGAs [18][36][37]; reducing the cost of normalization is one of their goals. These compilers achieve better performance and logic density than using 2-input operators, but they sacrifice IEEE compliance. Our approach is amenable to IEEE-compliant operators. A patent by Kaviani (*Xilinx*) [32] exposes the selection bits of C block multiplexers to the programmer; the idea is similar to Xilinx Virtex FPGAs, which do not have intra-cluster routing. No CAD tools are described, so the affect on routability is unknown. The next section presents the FPGA routing architecture, and illustrates the approach employed to model the intra-cluster routing resources in order to realize the hybrid CLB architecture.

2.1.3. FPGA ROUTING ARCHITECTURE

The routing fabric of an island-style FPGA can be viewed as a network of interconnected MUXes that is divided into two parts: the Intra-CLB routing, which is used to route signals from CLB input pins and feedbacks (i.e., BLE outputs in the same cluster) to LUT inputs; and the *inter-CLB routing*, which is used to route signals from BLE outputs to their destination clusters (or specifically, the routing tracks that drive the destination clusters). In this thesis we adhere to the inter-CLB routing architecture used in VPR [46], but present an intra-CLB routing model completely different from VPR, that share some similarities with the work of Guy and Lewis in [43], and Feng and Kaptanoglu [20]. The inter-CLB routing is a single connected structure for the whole device, while there is one intra-CLB routing for each cluster. **Figure 2-8** depicts an island style FPGA in which the Configurable Logic Blocks (CLBs) are surrounded by routing channels of pre-fabricated wiring segments on all four sides. A Connection Block (C

Block) of programmable switches connects each CLB input or output pin to a subset of the wires in the adjacent routing channel. **Figure 2-8** illustrates the FPGA floorplan. Switch Blocks (S Blocks) are programmable intersections between horizontal and vertical routing channels. They are simply a set of programmable switches that allow some of the wire segments incident to the switch block to be connected to other segments. By turning on the appropriate switches, short wire segments can be connected together to form longer connections.

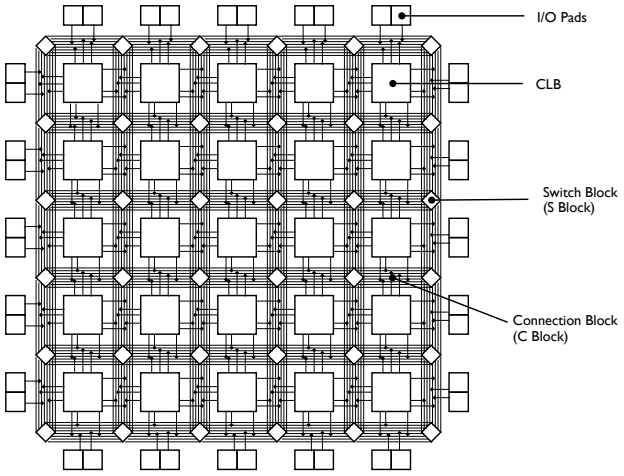


Figure 2-8: Generic Island-style FPGA.

The multiplexers, shown on the right-hand side of **Figure 2-9**, are implemented in the S Blocks, which are shown (without detail) in **Figure 2-9**. **Figure 2-9** depicts inputs coming in from the left hand side of the CLB and outputs leaving to the right; in actuality, inputs and outputs may enter and exit from all four sides.

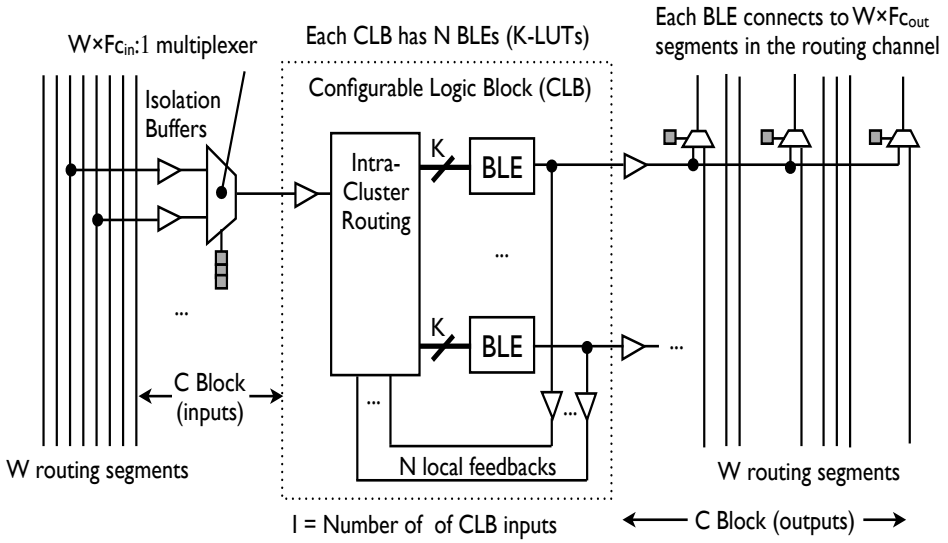


Figure 2-9: CLB and its adjacent routing channels.

Configurable Logic Block (CLB) contains several BLEs with fast local interconnect provided by the intra-cluster routing crossbar; the Connection Block (C Block) inputs and outputs interface the CLB with the global routing network **Figure 2-9**.

We use the same notation employed in VPR [46], originally introduced by Brown and Rose [9], for describing some of the parameters of an FPGA's routing architecture. The number of LUT inputs (LUT size) is denoted K, and N is the number of LUTs per CLB (Cluster size). I is the number of CLB input pins, and W denotes the number of segments per routing channel. The number of wires in each channel to which a logic block pin can connect is called the connection block flexibility, or F_c. The number of wires to which each incoming wire can connect in a switch box is called the switch block flexibility, or F_s. Each C Block input multiplexer in **Figure 2-9** selects one of $W \times Fc_{in}$ wires, and each

BLE drives $W \times Fc_{out}$ segments in the adjacent routing channels. Most FPGAs use single driver routing [7], so the C Block output is a conceptual description of the routing topology.

Prior work conducted by Vaughn Betz and Jonathan Rose in [7] and implemented in VPR [46], has extensively investigated the inter-CLB routing architecture, including the tradeoffs between the different parameters of this architecture, but only considered fully connected intra-cluster routing crossbars. In this work, we use their inter-CLB model but investigate the use of sparse crossbars in the Intra-cluster routing of the FPGA. The next section presents our proposed sparse crossbar model and section 2.3 describes the routing approaches we introduce for FPGAs with sparse intra-cluster routing crossbars.

2.1.4. INTRA-CLB ROUTING CROSSBARS

Intra-cluster routing crossbar can be viewed as a network of MUXes connecting the CLB pins to the LUT pins. Fully connected crossbars guarantee that there is a path in the intra-cluster routing network from each CLB pin to each LUT pin. When the crossbars are sparse this full connectivity is no longer guaranteed. **Figure 2-10** show the Intra-cluster connectivity pattern; **Figure 2-10**(a) depicts a fully connected crossbar, while **Figure 2-10**(b) shows a sparse crossbar.

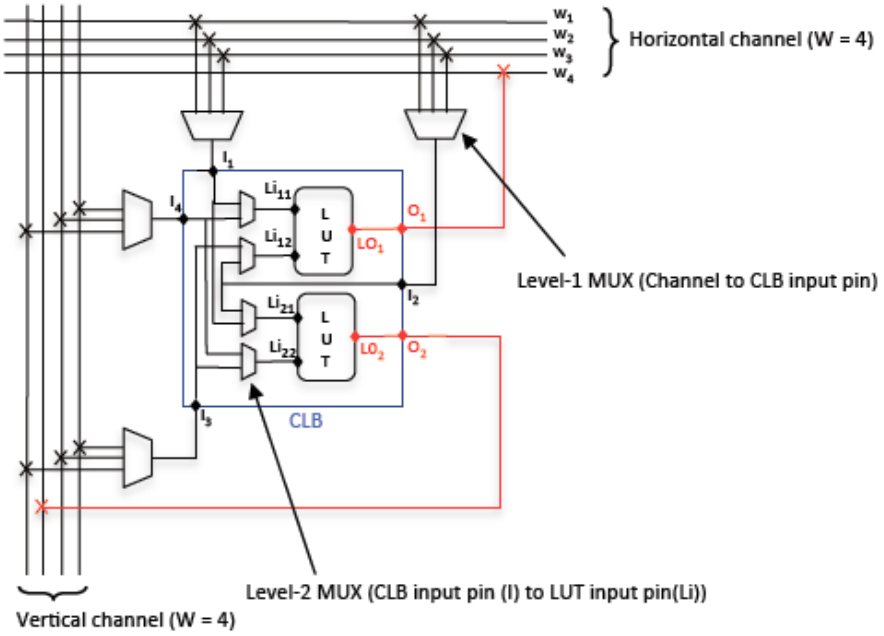


Figure 2-10: CLBs with sparsely connected intra cluster routing crossbars.

We model this intra-cluster routing connectivity as a 2-dimensional binary matrix B, with $I+N$ columns and KN rows. Each column corresponds to an input (a CLB input pin or a local feedback from a BLE in the cluster), and each row corresponds to a BLE input. $B(i,j)=1$ if a signal can route from input i to BLE input j, and *0* otherwise. It is important to note that B simply models the CLB-input-to-BLE-input connectivity of the crossbar, but does not model its internal architecture.

As an example, we model a CLB with $N=2, K=2$ (e.g., it contains two 2-LUTs); the four BLE inputs are denoted b_{00}, b_{01}, b_{10}, and b_{11}. The CLB has three input pins, I_0, I_1, and I_2, and two local feedbacks from the BLEs, O_0 and O_1:

$$B = \begin{array}{c} \begin{matrix} I_0 & I_1 & I_2 & O_0 & O_1 \end{matrix} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{matrix} b_{00} \\ b_{01} \\ b_{10} \\ b_{11} \end{matrix}$$

In this example, there is a connection from CLB input pin I_0 to LUT input pins b_{00} and b_{01}, but not b_{10} and b_{11}; also, the local feedbacks are not used.

We used a tool developed by Lemieux et al. [42] to generate routable sparse crossbars with a user-provided density function *p*. The tool generates matrix B such that each row, column, and the entire matrix all have a population percentage of approximately *p*, i.e.:

$$\sum_j B(i,j) = \lceil p(I+N) \rfloor \pm 1 \quad \text{.....(2.1)}$$

$$\sum_i B(i,j) = \lceil pkN \rfloor \pm 1 \text{ , and } \quad \text{.....(2.2)}$$

$$\sum_{i,j} B(i,j) = \lceil pk(I+N) \rfloor \pm 1 \quad \text{.....(2.3)}$$

Now that we have a logical mapping between the CLB pins and LUT pins we can model crossbars of any density and extend the routing resource graph to include the routing resources of these crossbars.

In the past decade, there has been some interest in investigating FPGAs that employ sparse crossbars in their intra-cluster routing, Lemieux et Lewis presented an algorithm to generate and evaluate routable sparse crossbars [42], and later proposed their usage for FPGA with intra-cluster routing; to improve routability they added spare CLB input pins [43]. Later work by Feng and Kaptanoglu [20] used entropy counting to design intra-cluster routing crossbars that offer greater routability; however, there is concern that CLB inputs and local feedbacks cannot reach fast inputs for LUTs with non-uniform delay [20].

Ye [73] showed how the equivalence of LUT inputs can be leveraged to reduce the population density of the intra-cluster routing crossbar without compromising routability; however, it is unclear if this approach is compatible with more advanced logic block features such as fracturable LUTs and carry chains, where LUT inputs can no longer be treated as logically equivalent. Chin and Wilton [16] extended Ye's work to investigate high-capacity hierarchical CLBs with multi-layer sparse crossbar interconnects, and showed that this approach reduced the placement and routing problem sizes significantly, thereby yielding faster and more robust CAD algorithms.

In terms of commercial FPGAs, Xilinx employs a C-block (**Figure 2-1**) without an intra-cluster routing crossbar, while Altera and Microsemi (formerly Actel) employ an intra-cluster routing crossbar in conjunction with a C-block. We presume that Xilinx's C-block is much denser than Altera's or Microsemi's, although no formal comparative study has been published, to the best of our knowledge.

Altera has disclosed that their Stratix-series devices employ a sparse intra-cluster routing crossbar [43]; although no details regarding the topology were presented. Microsemi disclosed that their FPGAs employ a 3-layer Clos network, where the third layer is subsumed by LUTs [42].

None of the previous work, however, investigated how routing is accomplished in the presence of sparse crossbars; section 2.3 briefly describes the proposed routing heuristics, and Chapter 3 presents the routing algorithms we employed to perform routing in FPGAs with intra-cluster routing crossbars.

2.2 CAD FOR FPGAS

2.2.1. OVERVIEW

Computer-Aided Design (CAD) has played a key role in the advancement and adoption of FPGAs across many industries. Due to the complexity of FPGA devices, the use of CAD tools has become an integral part of the process of synthesizing circuit designs onto an FPGA. A typical CAD flow takes as input a user circuit specified using a hardware description language (HDL) or a schematic along with a description of the target FPGA device. The CAD software then converts the high-level description into a binary file specifying the state of every configuration bit in the FPGA. This process is too complex to be modeled as one monolithic problem; it is therefore broken into a series of interdependent sub-problems, which are solved in sequence, as shown in **Figure 2-11**.

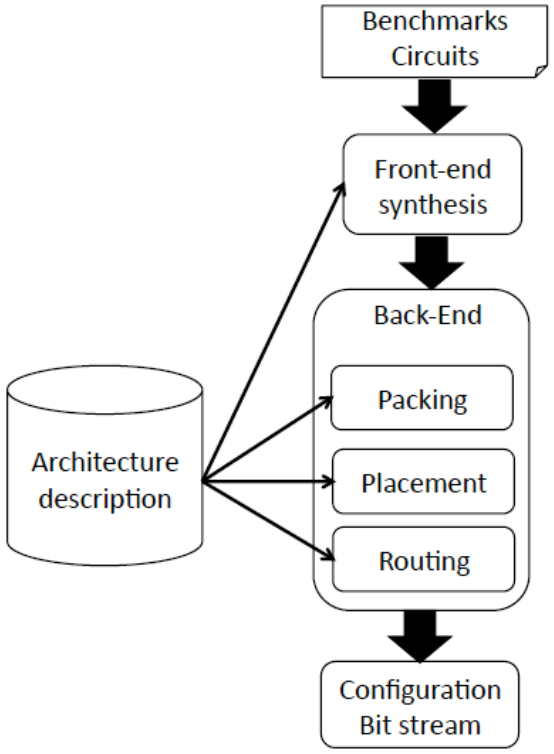


Figure 2-11: Typical CAD flow for an FPGA.

The first stage of this flow is called front-end synthesis. This step typically involves HDL elaboration, logic optimizations, and technology-mapping which maps the logic of the circuit into the LUTs of the FPGA. Front-end synthesis produces a netlist of FPGA logic blocks optimized in terms of logic blocks count and/or circuit speed.

The netlist then passes through the back-end flow, which involves packing, placement and routing. In the packing stage, the logic blocks of the technology-mapped netlist are clustered together, to determine which blocks are clustered together in the same CLBs; clustering logic blocks together creates opportunities for many types of efficiencies, such as CLB input sharing (if a net fans out to multiple logic blocks clustered in the same CLB), and the usage of the fast local feedbacks within the CLB,

rather than using the global FPGA routing network. The placement stage assigns each clustered logic block in the netlist to a physical CLB in the FPGA. Routing determines the paths in the routing fabric for individual signals in the netlist. The output of this flow is a configuration bitstream necessary to program the circuit on the FPGA.

A substantial amount of work has been published on the different stages of this CAD flow. Despite this work, publicly available CAD tools cannot model the hybrid logic blocks described earlier and offer limited to no capability for modeling sparse crossbars. Furthermore, synthesizing an industrial-scale circuit on a high-capacity commercial FPGA can easily take hours, days or even weeks, depending on the size of the circuit and the target device. The major contributions of this thesis in terms of FPGA CAD tool enhancement are mostly related to physical design (placement and routing). Specifically, we investigate the placement and routing of macro-cells, routing for FPGA with sparse crossbars, and parallel FPGA routing on multi-core, shared memory CPU architectures. The following two sections describe the placement and routing problems in the context of sparse crossbar routing model, and hybrid CLB architecture. Section 2.5 presents our parallel model targeting multi-core CPUs, and the final section summarizes this chapter.

2.2.2. PLACEMENT

Placement algorithms assign each clustered logic block to a physical CLB in the FPGA. This mapping has significant impact on the performance and routability of the circuit; as it determines the amount of interconnect in the FPGA, which is the major bottleneck of circuit performance. To minimize the impact of this bottleneck, the placer is often optimized for wire-length, routability, and timing. The mostly widely recognized

approach for FPGA placement is iterative improvement, specifically via simulated annealing; this is the approach taken by VPR [7].

```
S = RandomPlacement ( );
T = InitialTemperature ( );
R_Limit = InitialR_Limit ( );
while ( ExitCriterion ( ) == False) {                    /*"Outer loop"*/
    while ( InnerLoopCriterion ( ) == False) {           /*"Inner loop"*/
        S_new = GenerateViaMove (S, R_Limit);
        ΔC = Cost(S_new) − Cost(S)
        r = random (0, 1);
        if ( r < e^(−ΔC/T) ) {
            S = S_new;
        }
    }            /* End "Inner loop"*/
    T = UpdateTemperature ( );
    R_Limit = UpdateR_Limit ( );
}                /* End "Outer loop"*/
```

Figure 2-12: Pseudo-code for the FPGA placement algorithm.

Simulated annealing is a stochastic optimization method for finding the global minimum of a cost function; which may possess many local minima. It mimics the annealing process used to gradually cool molten metal to produce high-quality metal objects **[**71]. Pseudo-code for a generic simulated annealing-based placer is shown in **Figure 2-12**. The quality of any placement of logic blocks is evaluated by a cost function; which, for a wirelength-driven placement could be, the sum over al1 nets of the half-perimeter of their bounding boxes [7]. VPR placer creates an initial placement by assigning logic blocks randomly to the available locations in the FPGA.

The algorithm then performs a large number of moves to gradually improve the placement. The moves are evaluated based on the changes they instigate on the cost

function. Moves that decrease the cost are always accepted, if the cost would increase, the move is not automatically rejected. Rather, the acceptance of the move is decided using a probability of acceptance given by $r < e^{-\Delta C/T}$, where ΔC is the (positive) change in cost, and T the temperature of the annealing [71] that controls the likelihood of accepting moves that make the placement worse. VPR's placer sets T very high initially, so almost all moves are accepted; it gradually decreases T as the placement is refined, reducing the probability of accepting a move that negatively impacts the current placement solution. The algorithm terminates when the annealing process cannot generate better moves. VPR uses an adaptive annealing schedule that control the rate at which temperature is decreased, the exit criterion for terminating the anneal, the number of moves attempted at each temperature, and the generation of potential moves [7].

The work presented in this thesis uses the VPR 5.0 placement engine [46], with some modifications introduced in order to investigate the best approach for placing the macro-cells. The macro-cells are placed offline, prior to the rest of the circuit, and we attempted to optimize for routability and delay. More details on this will be provided in Chapter 2.

The next section covers the routing step, which constitutes the bulk of this work. We first present a general description of routing, then we show how routing is performed in a sparse crossbar, and macro-cells. Section 2.5 describes our parallelization of PathFinder on multi-core, shared memmory CPU architectures.

2.2.3. ROUTING

2.2.3.1. OVERVIEW

After the circuit placement has been chosen, routing is performed to configure the programmable switches of the routing fabric to connect the logic blocks and the I/O pads of the circuit. To establish a connection, the router must find a sequence of unused routing resources along a path from the source to the sink for each signal in the netlist. Routing usually involves the creation of a graph [7] representing the routing resources of the FPGA, commonly called the routing resource graph (RRG).

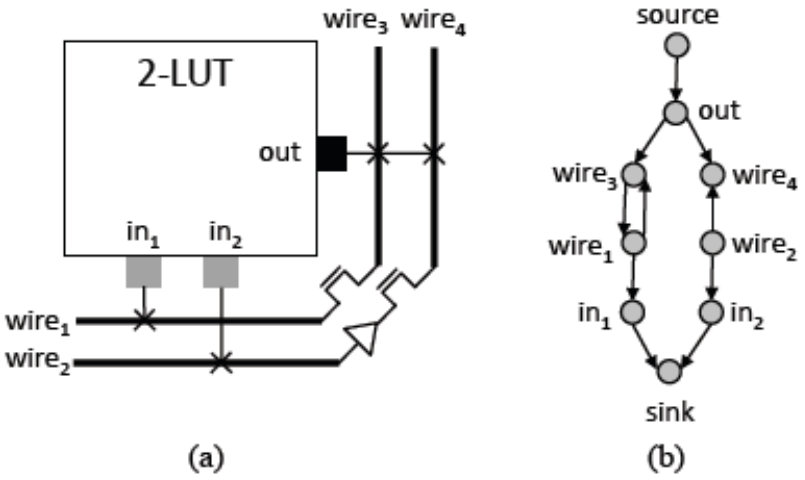


Figure 2-13: A small FPGA fragment (a) and its corresponding RRG (b) [8, Fig. 4].

As shown in **Figure 2-13**, the nodes of the RRG represent wires and pins of the FPGA and the edges represent switches or feasible connections between two nodes.

Each signal of the input circuit is represented by one source along with a subset of nodes corresponding to its sinks. Routing a signal involves assigning

routing resources such that all the sinks are reachable from the source. To minimize routing resources usage, this path has to be as short as possible. When routing a set of signals sequentially, the order in which the signals are routed may be critical since some routing resources needed by a signal may be occupied by signals that are routed earlier. For this reason, FPGA routers must employ a congestion avoidance mechanism to resolve contention for routing resources. Since most of the delay in FPGAs is due to routing, the critical path delay for a circuit should be kept minimal by using fast routing resources and shortest paths to route nets on or near the critical path. Routers that optimize the critical path delay are called timing-driven; others are generally classified as being routability-driven.

2.2.3.2. PROBLEM FORMULATION

C. Problem Formulation

FPGA routing is a technology-specific variation of the disjoint path problem from graph theory, which is one of Karp's original NP-complete problems [31]. In a graph, two paths are disjoint if they share no vertices or edges. **Figure 2-14** provides an example of disjoint and non-disjoint paths.

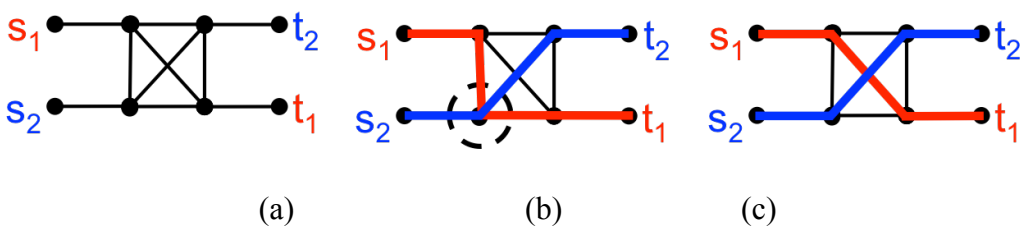


Figure 2-14: A simple instance of the disjoint path problem.

Figure 2-14 shows an instance of the disjoint path problem; **Figure 2-14**(a) shows a graph G(V, E) with sources $S = \{s_1, s_2\}$ and sinks $T = \{t_1, t_2\}$; an illegal solution, i.e., two non-disjoint paths that share a common vertex is shown in **Figure 2-14** (b); **Figure 2-14** (c) shows a legal solution, i.e., two disjoint paths that share no common vertices.

An instance of the disjoint path problem is a graph $G(V, E)$, and two sets of vertices: a set of sources $S = \{s_1, s_2, ..., s_k\}$ and a set of sinks $T = \{t_1, t_2, ..., t_k\}$. A legal solution is a set of paths $P = \{p_1, p_2, ..., p_k\}$ where p_i is a path from s_i to t_i in G, such that the paths in P are disjoint. The NP-complete decision problem is whether or not a set P of disjoint paths exists, given G, S, and T; corresponding optimization problems may try to minimize the total lengths of the paths in P, the length of the longest path in P. In the routing problem for FPGAs, the graph G is the RRG, and the set of sources and corresponding sinks is derived from the placement solution. One important difference is that each path in the FPGA represents a net in a digital circuit, where a source may fan-out to drive multiple sinks. Each net has the form $N_i = (s_i, T_i)$, where s_i is the source and $T_i = \{t_i^1, t_i^2, ..., t_i^n\}$ is the set of n sinks driven by source s_i; thus, p_i is actually a *hyper-path* (tree) that connects s_i to the sinks in T_i.

A second important difference involves the equivalence of sinks. Because LUTs are programmable logic functions, their inputs are equivalent. Without loss of generality, if a 2-input LUT is configured to perform a logic function $f(s_1, s_2)$, then there is an equivalent logic function $f'(s_2, s_1) = f(s_1, s_2)$, yielding a symmetric source/sink assignment, shown in **Figure 2-15**(a). Explicitly listing either pair as the one possible legal solution, as shown in **Figure 2-15**(b), is overly restrictive. Thus, it is necessary to introduce a single vertex t

to represent a common sink, as shown in **Figure 2-15**(c). Therefore, any legal routing solution must be node disjoint, *except at the common sink*.

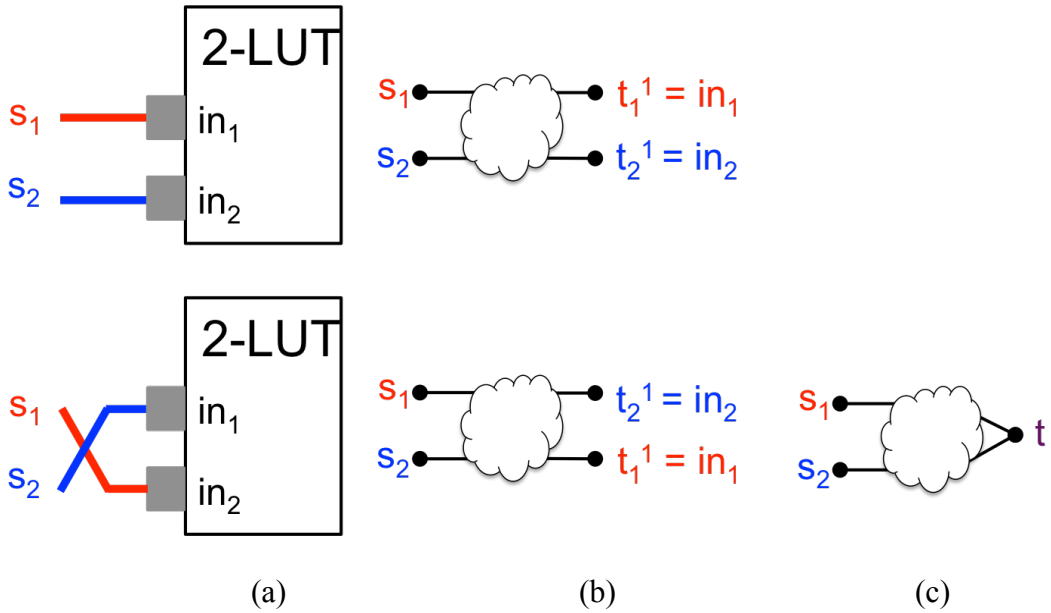


Figure 2-15: Logic equivalency of LUT inputs. Due to the equivalence of LUT inputs, different source-sink pairs may be legal solutions (a); however, enforcing specific source-sink pairs may be overly restrictive (b); the solution is to create a common sink (t) that represents all equivalent LUT inputs (c).

The objective of an FPGA router is twofold: (1) find a legal route, supposing the one exists; and (2) minimize the delay of the critical path in the circuit, which may involve the concatenation of several disjoint paths in the RRG. Many aspects of this delay will be technology-specific, including the logic delay through the BLEs on the path, delays relating to fanout, delays through routing multiplexers, wire delays in the routing network, etc.

Even though there have been several attempts at solving the routing problem in FPGAs [9] [40], mainly inspired by global routing for standard cells[63], by far the most successful approach is an algorithm called PathFinder [50], which is based on the

principle of negotiated congestion; both of VPR's routers are based on PathFinder, with some non-trivial modifications [46]. The next section presents the key elements of VPR's implementation of PathFinder, and highlights the limitations of the routing model in VPR, which motivated our work in modeling the intra-CLB routing resources as well as studying the routability issue in FPGAs that employ sparse intra-cluster routing crossbars.

2.2.3.3. PATHFINDER

This section summarizes the PathFinder FPGA routing algorithm [50]. PathFinder is based on the paradigm of *Negotiated Congestion (NC)*, which computes illegal routing solutions in which several nets may share a single wire (RRG vertex). The negotiation process dynamically adjusts a cost function, which, over time, pushes nets away from congested wires, and yields a globally legal routing solution.

Figure 2-16 presents pseudocode for PathFinder. The outer loop, called the *Global Router*, iterates until a legal routing solution is found (i.e., all nets are routed on unique RRG nodes). The pseudocode assumes that a legal routing solution *can* be found. In practice, the global router is often replaced with a fixed number of iterations; if a legal global route is not found, then routing is assumed to fail.

The *Signal Router* (lines 2-27) is oblivious to the notion of congestion (i.e., several nets sharing the same RRG vertex); a cost function (described below) is computed for each RRG vertex and is dynamically updated to dissuade the usage of congested vertices during routing. The objective of the signal router is to find a route that minimizes the aggregate cost of the RRG vertices that comprise the route.

```
// Global Router
1.  While at least two nets share a common routing resource
    // Signal Router
2.    For each net N_i
3.      Rip up routing tree RT_i for net N_i = (s_i, T_i)
4.      Reinitialize RT_i to contain only the source s_i
5.      For each sink t_i^j  T_i
6.        Initialize priority queue PQ to RT_i with cost 0
7.        While t_i^j  RT_i
8.          Remove min. cost vertex u from PQ
9.          Insert u into RT_i
10.           If u ≠ t_i^j
11.             For each RRG edge (u, v)
12.               If v RT_i and v PQ
13.                 Insert v into PQ with cost f_v = g_u,v + d_v^j
                    and predecessor edge (u, v)
14.               Else If v RT_i and v PQ and f_v > g_u,v + d_v^j
15.                 Change the cost of v in PQ to
                    f_v = g_u,v + d_v^j
16.                 Change the pred. edge of v in PQ to (u, v)
17.               EndIf
              EndFor
18.           EndIf
19.       EndWhile
20.     EndFor
21.     For each sink t_i^j  T_i
22.       For each node v in reverse path from t_i^j to s_i
23.         Update cost c_v
24.         Add v to RT_i
25.       EndFor
26.     EndFor
27.   EndFor
28. EndWhile
```

Figure 2-16: Pseudocode for the PathFinder FPGA routing algorithm.

The Signal Router routes one net a time; the routing tree RT_i for net N_i is expanded in search of each sink $t_i^j T_i$, one sink at a time, and in-order. The routing tree RT_i for net N_i, computed during the previous iteration, is discarded, and a new route is computed.

The new route may be computed using a priority-driven breadth-first search [50], similar to the *maze expansion* step of Lee's Maze Router [38]; more efficient routes can

be computed using an A* cost function, which includes an additional term that directs the search toward the target sink t_i^j [7][50][69].

The first search starts from source s_i of the current net N_i to the first sink, t_i^1, resulting in a routing path. Subsequent searches expand the routing path into a routing tree, RT_i. Inductively, suppose that RT_i connects s_i to the first j-1 sinks, $\{t_i^1, t_i^2, ..., t_i^{j-1}\}$. The search will find a path that connects one vertex in RT_i to the j^{th} sink, t_i^j.

Each search initializes a priority queue PQ to contain the vertices in RT_i at zero cost. After processing these vertices, PQ will contain each vertex that (1) has at least one neighbor in RT_i, and (2) does not belong to RT_i itself. The search works as follows: the lowest cost vertex v is removed from PQ and added to RT_i. The vertices adjacent to v are then examined and inserted into PQ accordingly. This process repeats until the current sink t_i^j is found. PQ includes an adjacent neighbor u of v that belongs to RT_i; thus, v and adjacent edge (u, v) are added together to RT_i.

Cost Function: An important implementation detail is the cost computed for each vertex when it is inserted into PQ. Different PathFinder implementations use different cost functions [7][50] [69], with different objectives and strategies. Let v be the vertex, and u be a vertex adjacent to v that has already been added to RT_i; in other words, if the search selects v for inclusion in RT_i, it will include edge (u, v) as well. Let f_u denote the cost of the path from source i to node u, and c_v denote the cost of adding node v to the route. Then the cost of routing from the source to v is:

$$g_{u,v} = f_u + c_v \quad \text{.....(2.4)}$$

To accommodate an A* cost function, let d_v^j be an estimate of the cost of completing

the route from node v to sink t_i^j. Then the cost of the path from source i to sink t_i^j along RRG edge *(u, v)* is

$$f_v = g_{u,v} + d_v^j. \quad \text{.....(2.5)}$$

A breath-first search, i.e., a Lee-style maze expansion [38], then corresponds to the case where $d_v^j = 0$. Several modifications have been proposed to assign relative weights to the the breadth-first and A* components of the cost function

$$f_v = g_{u,v} + \alpha d_v^j, \quad \alpha \geq 0; \text{ and} \qquad [13] \quad \text{.....(2.6)}$$

$$f_v = (1 - \beta) g_{u,v} + \beta d_v^j, \qquad 0 \leq \beta \leq 1 \qquad [14] \quad \text{.....(2.7)}$$

When adding a new vertex u into RT_i, each neighbor v of u is processed and added to PQ, unless v already belongs to RT_i. If is possible that a different neighbor w of v is also part of RT_i, so v may already be in the priority queue with some cost function

$$f_v = g_{w,v} + c_v.$$

In principle, it is now possible to add v to RT_i either via edge *(u, v)* or *(w, v)*. The best choice is the one that minimizes f_v. Therefore, the cost and predecessor of v in PQ are changed from w to u if $g_{u,v} < g_{w,v}$, or, equivalently, if $g_{u,v} + c_v$ is less than the current value of f_v.

Several different variants of the node cost function c_v have also been proposed:

$$c_v = (b_v + h_v) p_v, \text{ and} \qquad [1] \quad \text{.....(2.8)}$$

$$c_v = b_v h_v p_v, \qquad [7, \text{Eq. } (4.3)^1] \quad \text{.....(2.9)}$$

where b_v is the *base cost* of v (typically its intrinsic delay), h_v is the *history cost* of v, which depends on the number of nets that are routed through v during previous iterations,

[1] We ignore the *BendCost(...)* term from Eq. (4.3) in Ref. [7] because we are performing combined global-detailed routing.

[2] In combined global-detailed routing, which is the approach taken by VPR, the capacity of each routing resource is

and p_v is a *penalty function* associated with the number of nets routed through v in the current solution. PathFinder dynamically updates h_v and p_v accordingly as routing proceeds. According to Ref. [7], the advantage of Eq. (2.9) over Eq. (2.8) is that multiplying the b_v and h_v terms, rather than adding them, eliminates the need to normalize them; one possible drawback, not mentioned by [7], is that $b_v h_v > b_v + h_v$ for $b_v, h_v > 2$, so there is a greater chance of arithmetic overflow if both terms grow significantly as the algorithm iterates.

The difference between h_v and p_v is that h_v *permanently* increases the cost of using v to ensure that routes through other vertices are attempted, while p_v is based primarily on the current routing solution. Recall that PathFinder routes nets one-at-a-time. Suppose that nets N_1 and N_2 are being routed in subscript order. The history cost could potentially dissuade PathFinder from routing both N_1 and N_2 through v during the current iteration, especially if v has a history of congestion. Now, supposing that PathFinder routes N_1 through v despite the value h_v, then increasing p_v in response would dissuade PathFinder from routing N_2 through v, to increase the likelihood of converging to a legal solution.

A generalized form of the c_v terms that favors delay-minimization for source-sink pairs whose delay is expected to near-critical is

$$c_v = Crit_{i,j} delay_v + (1 - Crit_{i,j})(b_v + h_v)p_v, \text{ or} \quad \text{.....(2.10)}$$

$$c_v = Crit_{i,j} delay_v + (1 - Crit_{i,j}) b_v h_v p_v, \text{ such that} \quad \text{.....(2.11)}$$

$$Crit_{i,j} = 1 - Slack_{i,j}/D_{max}, \quad \text{.....(2.12)}$$

where $delay_v$ is the intrinsic delay of RRG node v, $Slack_{i,j}$ is the estimated amount of delay that could be added to the source-sink path from i to j before it becomes critical,

and D_{max} is the estimated critical path delay of the placed-and-routed circuits. In VPR's timing-driven router [[7], Section 4.4], the $delay_v$ term is based on the Elmore delay model, which is derived from the existing routing tree RT_i, including the prospective path from i to v; additionally, the $Crit_{i,j}$ term is more complex; details are omitted to conserve space.

The original PathFinder paper did not describe precisely which functions are used for h_v and p_v [50]. In VPR, p_v is reset and recomputed every time a routing tree is ripped up and rerouted, while h_v is defined as a recurrence relation which varies from iteration to iteration of the global router.

Let h_v^k denote the history cost of vertex v during the k^{th} iteration of the global router; for the first iteration, $h_v^1 = 1$. The p_v and h_v^k terms are then defined as follows:

$p_v = 1 + occupancy_v p_{fac}$, and [7, Eq. (4.4)[2]] ……(2.13)

$h_v^k = h_v^{k-1} + occupancy_v h_{fac}, k > 1$, [7, Eq. (4.5)[2]] ……(2.14)

where $occupancy_v$ is the number of nets that are current routed through RRG node v, and p_{fac} and h_{fac} are scaling factors. Ref. [[7], Section 4.3.1] suggests that p_{fac} should be at most 0.5 for the first iteration, and then increased by a factor of 1.5x to 2x for subsequent iterations, and that h_{fac} should remain constant and that any value between 0.2 and 1 should suffice.

[2] In combined global-detailed routing, which is the approach taken by VPR, the capacity of each routing resource is 1, which allows us to eliminate the *capacity(...)* term from Eqs. (4.4) and (4.5) in Ref. [7].

The enhancements to PathFinder introduced in this thesis, PPR and SERRGE, are compatible with any cost function (breadth-first or A* search) described in previous literature. We have implemented PPR and SERRGE in VPR 5.0, and all of our experimental results reported in Section 3.6 use VPR's timing-driven router [[7], Section 4.4].

2.3 ROUTING IN SPARSE CROSSBARS

A full intra-cluster routing crossbar is a configuration of the routing fabric inside the CLB in which a programmable routing connection exists between *every* CLB input and *every* BLE input within the CLB. This means that the router only needs to algorithmically compute routes from sources to CLB inputs, not BLE inputs; with a full crossbar connecting CLB inputs to BLE inputs, it is trivial to complete the route. A sparse intra-cluster routing crossbar on the other hand is a crossbar configuration in which each CLB pin is only connected to a subset of the BLE inputs. In this configuration, the router must find a full path from the sources to the BLE inputs; as CLB inputs are no longer connected to every BLE input. This means that the RRG has to include the routing resources inside the CLB.

In full crossbar architecture, the intra-cluster routing can be omitted from the RRG; this has been standard in VPR since its inception, although the assumption has since been lifted since the release of VPR 6.0. This also makes the CLB inputs logically equivalent; as each CLB input can connect to any BLE input. For sparse crossbars the CLB inputs

are no longer logically equivalent but for a given BLE the inputs are logically equivalent. As shown in **Figure 2-17**, due to the equivalence of LUT inputs, different source-sink pairs may be legal solutions; however, enforcing specific source-sink pairs may be overly restrictive; the solution is to create a common sink (t) that represents all equivalent LUT inputs.

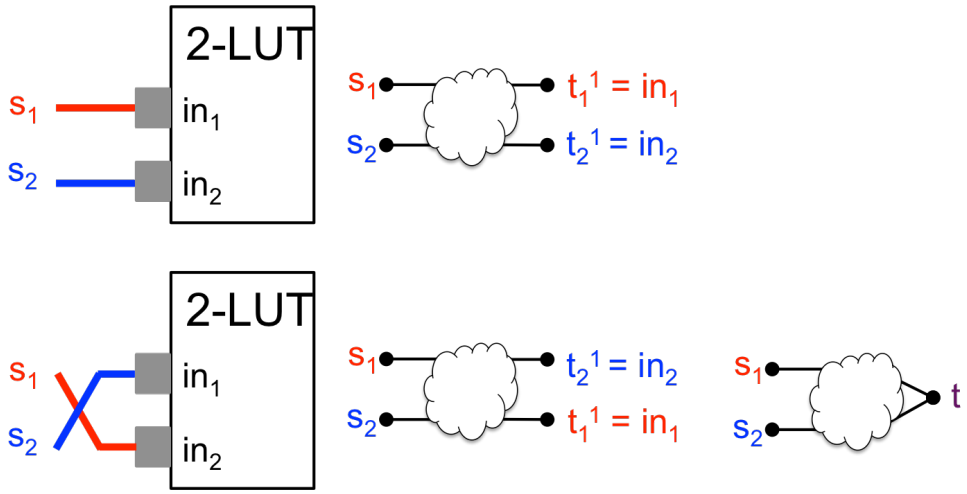
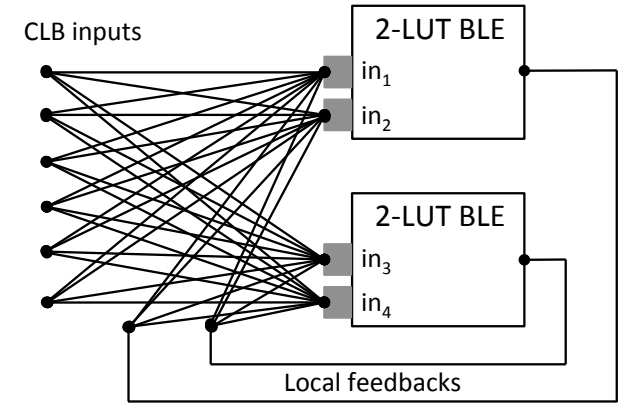
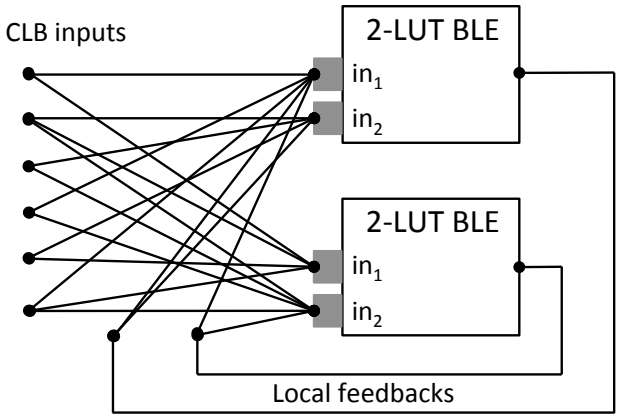


Figure 2-17: Logic equivalency for LUT pins.

Now that the crossbar is sparsely connected, in order for the router to complete a legal disjoint path routing solution, it is necessary to explicitly represent the intra-cluster routing crossbar in the RRG. This enlarges the size of the RRG: the set of vertices must include each CLB input and each BLE input (before, the CLB inputs could be represented as a single sink, akin to **Figure 2-18**, while BLE inputs were omitted altogether); and the number of edges that are added to the RRG depends on the population density of the crossbar. Taken in aggregation across the entire FPGA, the RRG size can increase significantly.



(a)



(b)

Figure 2-18: CLBs with fully connected (a) and sparsely connected (b) intra cluster routing crossbars.

This potential growth in the RRG size translates into significant increase in PathFinder's runtime and memory footprint. We present two heuristics for routing in FPGA with intra-cluster routing crossbars that reduces the runtime and memory footprint of the PathFinder FPGA routing algorithm for FPGAs with sparse intra-cluster routing crossbar. The two approaches are introduced with different characteristics in terms of runtime, memory usage, and quality of solution. *SElective RRG Expansion (SERRGE)* employs a memory manager that compresses the RRG and decompresses relevant

portions of it as the router executes, thereby eliminating the need to fully expand it prior to routing. A second, heuristic, *Partial Pre-Routing (PPR)* computes routes for each intra-cluster routing crossbar a-priori, and then routes the rest of the circuit using the global routing resources of the FPGA. Between the two, PPR achieves shorter runtimes and consumes less memory, while SERGGE tends to find legal routing solutions with lower critical path delays, equating to higher clock frequencies. Our results demonstrate that SERRGE and PPR address the routing challenge imposed by FPGAs with sparse intra-cluster routing crossbars, as they offer a clear and unequivocal improvement over the state-of-the-art in FPGA routing algorithms.

Prior work in this area has mostly investigated the routability of sparse crossbars. VPR's routability-driven and timing-driven routers, both based on PathFinder [50], have introduced sparse intra-cluster routing crossbars since the release of VPR6.0 [47]; using an approach similar to PPR to perform routing. The main difference with our work is that VPR integrates the partial pre-routing phase into the packer [47], as a legality check. In other words, any packing solution that cannot be routed locally within a CLB is disallowed. The router (which follows packing and placement) is similar to PPR's global router, described in Chapter 3 of this thesis.

In its original description, PathFinder computes a route from each source to each sink. For a multi-terminal net, the wavefront expansion obtained when routing to the i^{th} sink is discarded before routing to the $(i+1)^{st}$. VPR's implementation does not discard the wavefront, and continues expansion until all sinks are discovered.

If a global iteration of PathFinder fails to find a legal solution, all nets are ripped up and rerouted, which partially eliminates the dependence on net ordering when routing. Mulpuri and Hauck [54] modified PathFinder to exclusively rip up nets routed through oversubscribed resources. Although this modification speeds up PathFinder's convergence time significantly, a non-negligible increase in critical path delay was observed for all benchmarks; for this reason, we do not consider this implementation choice in our experiments.

Gort and Anderson [22] reduce contention for CLB output pins by forcing a multi-sink net to use the same output pin for all sinks during wavefront expansion while exclusively ripping up and re-routing nets that route through oversubscribed resources, similar to Mulpuri and Hauck; they reported a 3x speedup in router runtime coupled with a 2% increase in critical path delay and wirelength. Subsequently, Gort and Anderson [24] observed that PathFinder spends up to 40% of its runtime resolving congestion among nets that have been routed legally. They allow PathFinder to converge when the routing solution is almost legal on a coarsened RRG, and then legalize the result using a SAT solver. In principle, this approach is also compatible with either PPR or SERRGE.

Chin and Wilton [15] developed an RRG compression scheme that takes advantage of the regular tiled nature of an FPGA. An RRG is instantiated for each tile, and inter-tile connections are represented using "wrap-around" edges. Different tile types are instantiated for programmable logic, embedded blocks, and I/Os. Extensions are presented to handle long wires and sparse intra-cluster routing crossbars, including the heterogeneous depopulation schemes that vary from tile-to-tile. Separate storage is

maintained for the costs associated with each edge in the fully expanded RRG; this information is not compressed. The additional steps added to the router to enable the compressed RRG representation increase the router's runtime by a factor of 2.16x, on average. In contrast, PPR and SERRGE reduce the runtime of the router, although the reductions in RRG size reported here are far more modest in comparison to Chin and Wilton's scheme.

So [67] introduced a delay budgeting scheme to reduce the critical path delay of a circuit synthesized on an FPGA; since this is a post-processing step, it could improve the quality of results; however, it significantly increases the router runtime by a factor of at least 7x and also increases the memory footprint.

Rubin and DeHon [66] observed that small perturbations in initial conditions (e.g., the order in which nets are routed; variations in intrinsic delays associated with routing resources) yield significant variations in the critical path delays reported by VPR's implementation of PathFinder. They introduced noise mitigation strategies that repeatedly re-route each circuit with variations in parameters that express a timing constraint. In principle, this approach could be used in conjunction with either PPR or SERRGE, as long as the runtime overhead of repeatedly routing the circuit is tolerable.

We present a detailed description of our algorithms (PPR and SERRGE) in Chapter 3. The next section presents our approach to placement and routing of hybrid logic blocks capable of implementing floating point shifters.

2.4 ROUTING FOR HYBRID CLBS

In section 2.1.2 we have described dynamic configuration of multiplexers, and showed how this technique can be used to configure a regular Configurable Logic Block to implement shifting operations required by floating point mantissa alignment and normalization. We have also shown that by combining 8 hybrid blocks we can efficiently implement a 27-bit shifter. This section briefly describes how these hybrid CLBs can be placed and routed as part of a modern FPGA CAD flow.

2.4.1. FUNDAMENTAL CHALLENGES

The benefit of the SD-MUX is evident for circuits that require a significant amount of multiplexing; however, the introduction of dynamic multiplexers into an FPGA fabric creates new challenges for physical design tools. The following issues must be addressed in order to justify the inclusion of dynamic multiplexers in an FPGA fabric:

(1) Given that FPGA routing networks consume as much as 90% of on-chip area [20], is the area overhead of replacing static multiplexers with SD-MUXes justifiable?

(2) When the SD-MUX is configured for dynamic control, how can the router overcome the lack of flexibility arising from the fact that 8 input signals must be routed to 8 multiplexer inputs in a pre-specified order, as shown in Figure 2(b)? How is routability achieved in the general case?

(3) How are the dynamic control bits generated, and how are they routed into SD-MUX, as noted in Figure 2(b)?

The next section briefly presents our proposed solution to these challenges.

2.4.2. CAD SUPPORT FOR HYBRID CLBS

To address the area overhead of SD-MUXes, only a small number of these SD-MUXes is introduced. Realistic user circuits may contain a significant amount of multiplexer-based logic that benefits from the presence of dynamic multiplexers; however, they also contain other logic that maps better onto existing FPGA logic and arithmetic resources, such as LUTs, carry chains, and DSP blocks. As an example, floating-point operators require a large number of LUTs for shifters, but also include components that would not benefit from dynamic multiplexers, such as fixed-point adders and multipliers and leading zero counters. Thus, there is no need to replace more than a handful of static multiplexers with SD-muxes.

The remaining challenges are solved through the CAD algorithms. As previously noted, SD-MUXes configured as dynamic multiplexers impose significantly more constraints on the router than static multiplexers. To handle these constraints, the CAD tools extract *macro-cells*, which are sub-circuits comprised of the user logic that will use the dynamic multiplexers, plus the immediately preceding logic layer as well. The macro-cells are placed-and-routed separately from the remainder of the circuit.

This ensures that the router can satisfy all of the constraints imposed by the dynamic multiplexers without having the macro-cells compete with the remainder of the circuit for limited routing resources in congested areas. Placement then proceeds as normal, with some additional provisions to handle the macro-cells: the placer can move the entire macro-cell around within the FPGA, but cannot change the placement within the macro-cell. Once placement completes, routing resources within each macro-cell are reserved;

unused routing resources within the perimeter of the macro-cell are not reserved, as their usage does not affect the macro-cell's functionality. The remainder of the circuit is then routed as normal, with the restriction that the reserved routing resources within each macro-cell are not perturbed, thereby ensuring its correct functionality.

Chapter 3 of this thesis provides more details on the architecture and CAD support for hybrid logic blocks. Next section presents our model for parallelizing FPGA routing on multi-core, shared memory CPU architectures.

2.5 PARALLEL ROUTING FOR FPGAS

This section presents the background and related work to our parallel implementation of an FPGA router. We first give an overview of the proposed method, then, in section 2.5.2 we briefly present our implementation of the router on multi-core shared memory CPUs using the Galois [61] framework.

2.5.1. OVERVIEW

As FPGAs are constantly increasing in size and complexity, the need for parallel CAD tools is becoming particularly critical. Routing is possibly the most time consuming and resource demanding step of the CAD process. Parallelizing an FPGA router could yield tremendous productivity benefits, as the process of synthesizing an industrial-scale circuit on a high-capacity commercial FPGA can easily take hours, days or even weeks, depending on the size of the circuit and the target device. Meanwhile, the evolution of computer architectures (*multi-core* and *many-core*) towards a higher number of cores can only confirm that parallelism is the most popular method for speeding up CAD

algorithms.

Commercial FPGA CAD tools use a variant of the PathFinder negotiation congestion algorithm for routing [50]. As stated in section 2.2.3.3, the routing procedure of PathFinder has two level of operation; at the highest level the algorithm invokes a signal router to control the negotiation procedure between signals. The lowest level of operation consists of a *maze expansion* step to explore the nodes of the routing resource graph (RRG) to route individual signals. Prior work in this area[22] has stated that this maze expansion consumes over 66% of run time of the router, making it a potential target of our parallelization scheme.

Parallelizing PathFinder can be done in one of two ways; parallelizing the signal router (coarse-grained), which involves routing the nets in parallel and using a shared congestion map to control the negotiation process. The biggest challenge of this approach is to find an efficient mechanism to handle contention and communication among concurrent threads. Parallelizing the maze expansion (fine-grained level) on the other hand involves routing nets serially, while parallelizing the maze expansion of individual nets. As the maze expansion is typically a directed breadth-first or A* search on the RRG, the main concern is finding a mechanism for implementing the priority queue such that it is optimized for multithreading operations.

In this work we investigated parallelizing both operations individually, on a multi-core, shared memory CPU system. The next section illustrates the proposed parallel routing model.

2.5.2. PARALLEL ROUTING ON MULTICORE SHARED MEMORY SYSTEMS

This section describes a parallel PathFinder implementation using the open source Galois framework [51][61]. We first present an overview of the parallel model used in Galois, next we describe our implementation of PathFinder in Galois, and illustrates previous work related to parallel routing in FPGAs.

2.5.2.1. THE GALOIS FRAMEWORK

Galois' programming model, compiler, and runtime synergistically accelerate irregular algorithms that dynamically modify linked-based data structures.

Irregular algorithms generally operate on a sparse graph, typically implemented using a linked data structure. They do exhibit significant parallelism, but are difficult to parallelize statically [61] because the amount of parallelism depends on the content of the data structure (e.g., graph topology) as well as the operations performed on the elements of the data structure at runtime.

a) Philosophy and Implementation: Galois employs a data-centric approach to irregular algorithm development called the *operator formulation*. In a graph, *active elements* are the vertices and/or edges where computation could be performed through the application of an *operator*. The *neighborhood* of an activity is the set of vertices and edges that the activity reads or writes.

In Figure 2-19, the active vertices are those in *PQ*; each neighborhood is the set of adjacent vertices to each active vertex, and the operator applied is the neighborhood expansion in which newly discovered adjacent vertices are inserted into *PQ* (Figure 2-19(b)) and sinks may be discovered (Figure 2-19(c)). When a sink is discovered, the backtrace process involves a different set of active elements, neighborhood definition, and operator to update the routing tree $RT(N_i)$.

The operator formulation naturally lends itself to *amorphous* data parallelism, which permits parallel processing of active vertices, limited by algorithm-dependent neighborhood and ordering constraints. Executing one activity may create others dynamically; e.g., applying the neighborhood operation to a vertex may create new active vertices to insert into *PQ*. Activities are allowed to modify the graph; the itself RRG is not modified, but the routing tree $RT(N_i)$ is constructed incrementally, one path at a time.

Conflicting activities cannot execute concurrently. For example, consider two vertices *u* and *v* that share a common neighbor, *w*. If both *u* and *v* expand their neighborhoods concurrently, then each expansion will discover and add *w* to PQ with different path costs. This type of conflict must be avoided. Activities that do not modify their neighborhoods can always execute in parallel.

Galois uses locks to ensure that only activities with disjoint neighborhoods execute in parallel. Each graph element has an exclusive lock that must be acquired by a thread before it can access that element. Locks are held until the activity terminates.

If a lock cannot be acquired because it is owned by another thread, the Galois runtime detects the conflict and rolls back one of the conflicting activities. Lock

manipulation is performed entirely by the methods in the graph class. To enable rollback, each graph API method that modifies the graph makes a copy of the data before modification, similar to transactional memory systems. This copy, called an *undo log*, supports rollback in the case of misspeculation, and is discarded whenever an activity successfully commits. When an activity aborts, all computation performed up to that point is lost; the Galois runtime system takes corrective action to roll back the activity and re-execute it after all other conflicting activities complete.

a) Scheduling in Galois : Galois schedules parallel tasks either deterministically[58] or non-deterministically.

For programming models with non-deterministic scheduling, conflict detection and correction can be done using much lighter weight mechanisms than for models that employ deterministic scheduling. Abstract locations can be acquired by an owner and the execution of a task can be divided into two phases: in the first phase, a task reads locations but does not write to any of them, acquiring ownership of these locations, and in the second phase, the task writes to some locations, but it does not write to any location that it did not read in the first phase. The point between the first and second phase is called the failsafe point. For cautious tasks, conflicts are detected in the first phase, and rollback is implemented simply by releasing ownership of all locations. Once the failsafe point has been crossed, global data structures can be updated in place without the need for backup copies of modified data.

The deterministic scheduler[58] on the other involves the creation of in interferance graph for the set of tasks and find the independent sets in that graph. The interference

graph can is defined as an undirected graph Gp = (Vp, Ep) in which there is a distinct node in Vp representing each task in P, and there is an undirected edge (v_1,v_2) in Ep if the tasks represented by v_1 and v_2 have a conflict.

The interference graph for a set of tasks can be built by executing each task up to its failsafe point while tracking its neighborhood and putting a conflict edge between two tasks if their neighborhoods overlap. This graph can be used for scheduling as follow:

The tasks in the task pool P are executed in rounds. In each round, the scheduler performs the following activities:

1 - Build an interference graph Gp for the tasks in P,

2 - Find an independent set I in Gp and remove corresponding tasks from P,

3 - Execute the tasks in I in parallel, adding any newly created tasks to P.

Scheduling is completed when all tasks have been executed.

This procedure guarantees deterministic scheduling of the tasks in P.

In the next sections we present how we leveraged the Galois optimization techniques to parallelize both the maze expansion and the signal router.

2.5.2.2. PARALLELIZING THE MAZE ROUTER

Maze expansion computes a path from the source to each sink in the RRG for each net. All of the RRG vertices that have been uncovered are stored in a priority queue (*PQ*) based on their cost. Maze expansion extracts the minimum cost vertex v_{min} from *PQ*. If v_{min} is a sink, then a backtrace procedure is invoked to construct a path from u to $RT(N_i)$, which is the routing tree for N_i; otherwise, each neighbor v of v_{min}, which has not

previously been discovered, is inserted into *PQ* and the maze expansion continues. Figure 2-19 shows an example of the maze expansion of a given net:

(a) PQ contains vertices that have been discovered, but have not yet had their neighborhoods expanded; $RT(N_i)$ contains the portions of N_i's routing tree that have been found thus far.

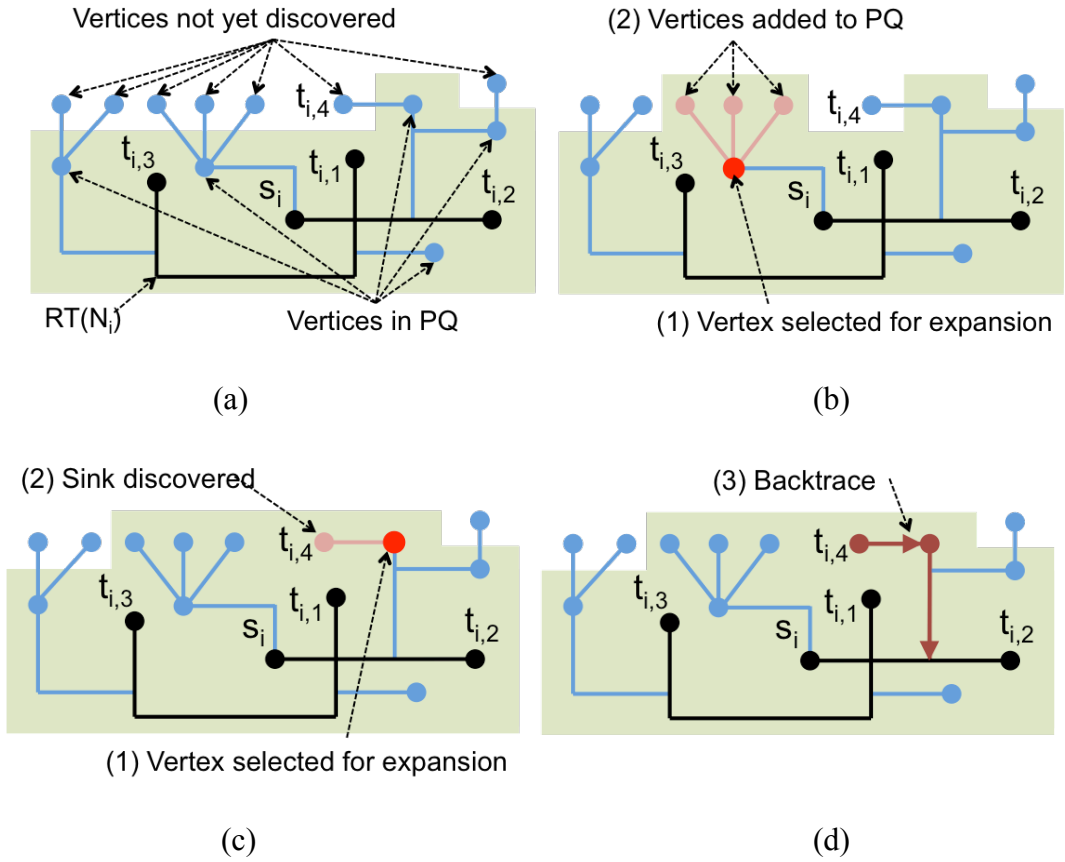


Figure 2-19: the maze expansion of a net in the routing resource graph (RRG

(b) A vertex is selected for neighborhood expansion; its neighbors that have not yet been discovered are inserted into PQ, expanding the "wavefront" of the search.

(c) Expanding the neighborhood of the next vertex discovers a sink.

(d) A backtrace adds the vertices and edges along the backtrace to $RT(N_i)$, creating a new path to the sink.

The *path cost* of vertex v is the sum of the vertex costs on the path from source s_i to v as uncovered by maze expansion:

$$PathCost(v) = c(v) + PathCost(v_{min}). \quad (2.15)$$

When v is inserted into PQ, $PathCost(v)$ is used as its priority.

Maze expansion is therefore a perfect example of a highly parallel irregular algorithm; it can explore many RRG vertices independently, albeit, with restrictions: namely, parallel operations cannot be applied to two adjacent vertices at the same time. Thus, the amount of parallelism that maze expansion can exploit depends on the RRG's sparseness; fortunately, RRGs are sparse in practice.

One of the most challenging aspects of parallelizing the maze expansion in Galois is to introduce a mechanism for threads to share a priority queue. We have extended the Galois framework with this capability by implementing a non-blocking priority queue based on the software transactional memory (STM) models.

The details of this implementation are provided in Chapter 5. Next section gives an overview of the signal router implementation.

2.5.2.3. PARALLELIZING THE SIGNAL ROUTER

Parallelizing the signal router involves partitioning the signals (netlist) into sets, with each set routed by a separate instance of the signal router running as a separate thread on

a separate processor. Each signal router instance maintains a local priority queue while the RRG and the associated congestion maps are shared among all threads. The Galois model maps each thread to a different processor. Threads update intermediate routing results through lock based data structures in shared memory, and synchronize their respective views of the overall routing state. Threads are synchronized at the end of each iteration of PathFinder to ensure that all instances are working simultaneously on the same iteration.

2.5.2.4. RELATED WORK

In 1997, Chan and Schlag [12] parallelized PathFinder's signal router on a distributed network of workstations. Using three processors, they achieved a 2.5x speedup. The drawback of their approach is that the results are highly sensitive to the order in which signals are routed. Consider two nets N_1 and N_2, and assume that N_1 would route before N_2 in a serial implementation. If so, N_2 would read congestion costs of any routing resource used by N_1, and may choose a different routing resource as a result; in the parallel implementation, N_2 may not read those congestion costs and could therefore make an ill-advised routing decision.

A deterministic scheduler [58] could rectify the issue by imposing a deterministic order on the nets; however, determining the best ordering a-priori appears to be an open problem [66]. Moreover, the overhead of speculatively parallelizing this particular scheme would be quite high. Galois would need to acquire locks for each routing tree $RT(N_i)$ before any signal route could commit; thus, threads would acquire and hold on to

locks for a long time, inhibiting parallel execution. This would create large undo lists, and the cost of conflict resolution would be exorbitant, as very large partially-computed routing trees would need to be discarded. Zhu et al. [74] addressed this concern, but in a limited way. They partition high-fanout nets into sets of low-fanout nets, which are routed individually; low-fanout nets with non-overlapping bounding boxes are routed in parallel because they are unlikely to conflict. Zhu et al. achieved a speedup of 1.9x on a quad-core machine with 2.3% degradation in critical path delay. Our approach using Galois yielded comparable speedups for two cores and higher speedups for four and eight cores, without requiring specialized handling of high-fanout nets; however, our approach parallelized the maze router, rather than the signal router.

Gort and Anderson [24] parallelized the signal router by partitioning the netlist into groups of disjoint subnets, and routed each subnet using independent instances of VPR running on different processors, communicating via MPI; blocking receive calls ensure deterministic results. Each VPR instance routes one signal at a time and then synchronizes with the other instances to update the relevant costs functions before routing the next net. To limit the synchronization overhead, nets to be routed are load-balanced among VPR instances based on fanout and bounding box size. They achieve a speedup of 1.5x on two cores 2.1x on four cores. The synchronization overhead suggests that parallel signal routing may not scale as well as maze expansion.

Our signal router is very similar to their approach except that we relied on Galois for load balancing and resource sharing. Our deterministic signal router achieved up to 1.84x speedup for 8 threads, we attribute this relatively low speedup to the amount of overhead

associated with the deterministic scheduling, as well as the lack of any non-blocking mechanism for the shared routing resources. Further optimizations, of the deterministic scheduler may yield better results.

Gort and Anderson [24] also parallelized the maze router using pthreads. Each pthread has an LPQ, similar to Galois' iteration coalescing, along with a GPQ in shared memory. The PQs are lock-based, similar to the PQ provided by Galois. Greater speedups can be obtained by using a non-blocking PQ. Unlike our work, their threads update the GPQ after each expansion; ours only access the GPQ when a sink is found or if a thread's LPQ is empty. Gort and Anderson achieved a speedup of 1.2x with two threads on a quad-core PC; increasing the number of threads yielded slowdowns. In contrast, our results achieve far greater speedups for up to 8 threads using Galois.

2.6 SUMMARY

In this chapter we first reviewed the basics of FPGA architecture. We then presented the logic blocks architecture investigated in this work, and introduced the hybrid logic block architecture used to reduce the cost of floating point shifters. Next, we presented a model for intra-cluster routing crossbars, and presented the CAD flow used with FPGAs. We focused specifically on algorithms for routing, and described two heuristics for routing in FPGAs that employ sparse crossbars. Finally, we described our parallel model to parallelize the router both on GPUs and multi-core CPUs

The next chapter describes routing in FPGAs with sparse crossbars, and details the routing approaches we have developed. Since we have described most of the concepts

related to routing in sparse crossbars in this chapter, we will focus on the enhancements we have made. Chapter 4 elaborate on the hybrid logic block model, and show the feasibility of this framework for reducing the cost of floating point shifters. In Chapter 5, we present our speculation-based model for parallelizing the router on a shared memory multi-core CPU system. Chapter 6 concludes the thesis.

Chapter 3. ROUTING WITH SPARSE CROSSBARS

This chapter extends the FPGA routing model implemented in VPR to encompass sparse intra-cluster routing crossbars, and introduces two scalable heuristics that reduce the runtime and memory footprint of FPGA routing: (1) SElective RRG Expansion (SERRGE), which employs an application-specific memory manager that stores the RRG in a compressed form, and dynamically decompresses it as the router proceeds; and (2) Partial Pre-Routing (PPR) locally routes all nets within each logic cluster, followed by a global routing stage to complete the routes. PPR and SERRGE converge faster than a traditional router using a fully expanded RRG. PPR runs faster and uses less memory than SERRGE, while SERRGE yields the highest clock frequencies among the three.

3.1 OVERVIEW

The long running times of commercial CAD software is one impediment to the widespread adoption of FPGA technology. Practically, all commercial FPGA routers have their origins in the PathFinder algorithm[50]. PathFinder employs an algorithmic approach called negotiated congestion, in which individual nets in the user circuit are allowed to share FPGA routing resources; as the algorithm proceeds, the negotiation process ensures that at most one net is routed along each resource. This process is often lengthy and memory-intensive. In particular, the Routing Resource Graph (RRG) of a commercial-grade FPGA can be very large, due to the inordinate quantity of uniquely programmable routing resources that are present in the architecture.

One of the significant contributors to overall RRG size is the presence of sparse intra-cluster routing crossbars within the FPGA routing network [20][42] [43]. In early FPGA generations, intra-cluster routing crossbars were fully connected, which allowed the RRG to implicitly represent them. When the crossbars become sparse, the implicit representation is no longer accurate, so the need to explicitly enumerate their connectivity significantly enlarges the overall RRG size.

In this work we reduce the runtime and memory footprint of the PathFinder FPGA routing algorithm for FPGAs with sparse intra-cluster routing crossbar. Two heuristics are introduced with different characteristics in terms of runtime, memory usage, and quality of solution. *SElective RRG Expansion (SERRGE)* employs a memory manager that compresses the RRG and decompresses relevant portions of it as the router executes, thereby eliminating the need to fully expand it prior to routing. A second, heuristic, *Partial Pre-Routing (PPR)* computes routes for each intra-cluster routing crossbar a-priori, and then routes the rest of the circuit using the global routing resources of the FPGA. Between the two, PPR achieves shorter runtimes and consumes less memory, while SERGGE tends to find legal routing solutions with lower critical path delays, equating to higher clock frequencies. Our results demonstrate that SERRGE and PPR address the routing challenge imposed by FPGAs with sparse intra-cluster routing crossbars, as they offer a clear and unequivocal improvement over the state-of-the-art in FPGA routing algorithms.

The user describes an FPGA using VPR's architecture configuration file. VPR reads in the architecture configuration file and algorithmically generates the logic and routing

architecture of the FPGA [46]. This alleviates the need for the user to specify every connection within the device.

VPR (versions 5.0 and before) model FPGAs with full intra-cluster routing crossbars, as described in the last chapter. Specifically, a full intra-cluster routing crossbar means that a programming routing connection exists between *every* CLB input and *every* BLE input within the CLB. This means that the router only needs to algorithmically compute routes from sources to CLB inputs, not BLE inputs; with a full crossbar connecting CLB inputs to BLE inputs, it is trivial to complete the route. Thus, the intra-cluster routing crossbar can be omitted from the RRG; this has been standard in VPR since its inception, although the assumption has since been lifted since the release of VPR 6.0[47]. Now, the intra-cluster routing crossbar topology is part of the architecture configuration file.

When the intra-cluster routing crossbar becomes sparse, as, CLB inputs are no longer equivalent (in the general case). In order for the route to complete a legal disjoint path routing solution, it is necessary to explicitly represent the intra-cluster routing crossbar in the RRG. This enlarges the size of the RRG: the set of vertices must include each CLB input and each BLE input (before, the CLB inputs could be represented as a single sink, while BLE inputs were omitted altogether); and the number of edges that are added to the RRG depends on the population density of the crossbar. Taken in aggregation across the entire FPGA, the RRG size can increase significantly.

3.2 BASELINE ROUTER

The *Baseline* router, described in this section, contains a minimalist set of algorithmic modifications to extend the PathFinder algorithm to support FPGAs with sparse intra-cluster routing crossbars. The Baseline router suffers from an enlarged memory footprint, which both SERRGE and PPR, described in the subsequent sections, overcome.

3.2.1. RRG TERMINOLOGY

We use the term *global RRG* to refer to the representation of the FPGA's global (inter-cluster) routing resources. The VPR 5.0 (and earlier) PathFinder implementation performs routing on a global RRG, which does not explicitly represent any local (intra-cluster crossbar) routing resources.

We use the term *local RRG* to refer to the representation of the intra-cluster routing resources for *one* CLB; if the intra-cluster routing crossbar contains just one layer of internal multiplexers, the local RRG is bipartite: each vertex is either a CLB input pin (including local feedback arcs) or a BLE input pin; each edge connects a CLB input pin to a BLE input pin.

We use the term *complete RRG* to refer to the representation of all FPGA routing resources (inter- and intra-cluster) in a single graph: a complete RRG combines the global RRG with a local RRG for *each* CLB in the FPGA.

3.2.2. EXPANDED RRG AND CLB INPUT PIN EQUIVALENCE

The *Baseline* router performs routing on a complete RRG, which explicitly represents both inter- and intra-cluster routing resources. PathFinder now routes nets to BLE input pins, rather than CLB input pins, as shown in Figure 3-1, and delineates which BLE is the sink of each net. The input pins of each BLE are logically equivalent.

If the intra-cluster routing crossbar is fully populated, then all CLB input pins are logically equivalent and do not require explicit representation in the CLB. A legal route is obtained by routing all nets from their respective sources to any input pin of a CLB that contains the sink; a full crossbar guarantees a direct connection from each CLB input pin to the sink.

When the intra-cluster routing becomes sparse, CLB input pins are not logically equivalent; whatever equivalency exists depends on the crossbar topology. Let S_j contain the CLB input pins that can be routed to at least one input of BLE j. In general, each CLB input pin may belong to several such sets.

For example, consider Figure 3-1: all CLB inputs, except for the feedback emanating from the top BLE, connect to at least one input of both LUTs; all of them belong to subsets S_1 and S_2; the feedback output belongs to subset S_1, but not S_2.

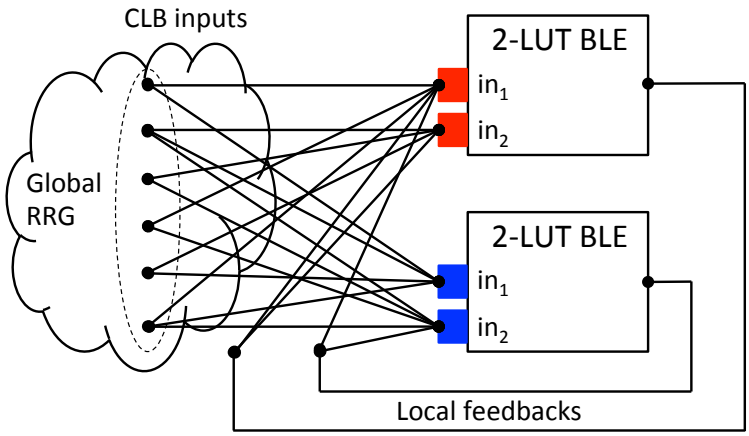


Figure 3-1: RRG expansion for the Baseline router.

As shown in Figure 3-1 the Baseline router, the RRG is extended to include LUT inputs (each of which forms a unique equivalence class) and the sparse intra-cluster routing crossbar; this expansion is performed for every CLB in the FPGA.

3.2.3. WIRE-TO-PIN LOOKUP MAP

An important data structure that complements the RRG in VPR is a wire-to-pin lookup map that identifies connections between the channel wires and CLB input pins. In VPR 5.0 and earlier, the lookup map is a 4-dimensional array, as shown in Figure 3-2(a). Since the intra-cluster routing crossbar is fully populated, there is no need to extend the map into the CLB. When the intra-cluster routing crossbar becomes sparse, the router needs to know whether a wire in the routing channel has a connection to each LUT input, which goes through both the C-Block (as before) as well the intra-cluster routing crossbar. To accommodate this information, two extra dimensions must be added to the wire-to-pin lookup map, as shown in Figure 3-2(b).

Although needed for correctness, the memory overhead of these two extra dimensions is significant. For example, consider an FPGA with parameters *K = 10, N=6, I = 33, W=100,* Fc_{in} *= 15%,* and Fc_{out} *= 10%.* In VPR 5.0, the intra-cluster routing crossbar implicitly has population density *p = 100%*, and a matrix of dimensions 33x4x15 = 1980 is allocated (assuming height=1). For a sparse crossbar with population density of *p = 50%*, the matrix size expands to 1980x50x60 = 5,940,000. The increase in cost is significant, since the wire-to-pin map is relatively sparse.

```
int **** tracks_connected_to_ipin;
tracks_connected_to_ipin = alloc(num_pins, height, 4, Fc);
/* tracks_connected_to_ipin[num_pins][height][4][Fc] */

for(int i = 0; i < num_pins; i++)
  for(int j = 0; j < height; j++)
    for(int k = 0; k < 4; k++)
      for(int l = 0; l < Fc; l++)
        tracks_connected_to_ipin[i][j][k][l] = OPEN;
(a)
int ****** tracks_to_LUT_ipin;
tracks_to_LUT_ipin = alloc(K*N, density, num_pins, height, 4, Fc);
/*  tracks_to_LUT_pin[K*N][density][num_pins][height][4][Fc] */
/*  K*N is the number of BLE inputs pins (N K-LUTs per CLB)
    density is the number of CLB input pins that connect to each BLE
    input. If p is the population density of the crossbar, then
    density = p*I. (e.g., if I=40, p=75%, then density = 30).
*/
for (int i = 0; i < K*N)
  for (int j = 0; j < density; j++)
    for(int k = 0; k < num_pins; k++)
      for(int l = 0; l < height; l++)
        for(int m = 0; m < 4; m++)
          for(int n = 0; n < Fc; n++)
            tracks_to_LUT_pin[i][j][k][l][m][n] = OPEN;
(b)
```

Figure 3-2: (a) Pseudocode to allocate and initialize the 4-dimensional wire-to-pin lookup map array in VPR 5.0 (and earlier).

3.2.4. MEMORY FOOTPRINT

Empirically, we observed that the Baseline router has an excessively large memory footprint. The two main causes are the expanded wire-to-pin lookup map, as described in the preceding subsection, and the Elmore delay trees computed by VPR's timing-driven router [7, Section 4.4], which are used to accurately estimate the delay term used in the cost function c_v shown in Eqs. (2.14) and (2.15); VPR's routability-driven router [[7] Section 4.3] does not use Elmore delay modeling and sidesteps this overhead. Other relevant data structures (e.g., the expanded RRG, priority queue, traceback list, etc.) become larger, but do not significantly impact the memory footprint.

VPR's timing-driven router builds an Elmore delay tree for each vertex as it is discovered during maze expansion. If the signal router is presently routing net N_i, a tree is computed for each vertex that is discovered during the search. The tree is then saved when the vertex is inserted into the priority queue; many of these vertices are never removed from the priority queue during the search, and even fewer are added to the routing tree. The contribution of Elmore delay trees to the memory footprint varies from iteration to iteration.

The lookup map in Figure 3-2(a) is allocated under the assumption that the intra-cluster routing crossbar was fully populated. Figure 3-2(b) presents the pseudocode to allocate and initialize a 6-dimensional wire-to-pin lookup map array, which has been extended to support sparse intra-cluster routing crossbars, which do not guarantee a connection between each CLB input pin and each LUT input. The map entries that are

set to USED (rather than OPEN), i.e., connections that actually exist, are derived from the FPGA routing architecture.

The impact of the memory footprint on performance depends on the target FPGA size, the placement solution, and the amount of memory available on the system that computes the route. The operating system's memory management policies and background applications and services also affect the amount of memory made available to the router. To manage this overhead, we set a limit on the memory size of all of the routing resources; in practice, the choice of limit depends on the system configuration and memory demands of the operating system and other persistent applications. When a PathFinder iteration exceeds the memory limit, the Baseline Router treats that iteration as a failure: it deallocates all data structures and propagates any history cost updates from the failed iteration to the next iteration. The Baseline Router does not otherwise modify PathFinder's core algorithmic behavior.

3.3 ROUTING WITH SERRGE

SElective RRG Expansion (SERRGE) refers to a collection of modifications to the Baseline router, which further reduce the memory footprint, yielding significantly faster runtimes. SERRGE features a custom memory manager and garbage collector that are specific to the RRG and other associated data structures used by VPR's implementation of PathFinder.

3.3.1. DYNAMIC RRG

SERRGE begins with a global RRG $G = (V, E)$ and one copy of a local RRG $G_L = (V_L, E_L)$ as a representative of each CLB. When routing each net N_i, PathFinder's maze expansion first finds a path in the global RRG from the source s_i to an input pin j of a CLB that contains one of N_i's sinks. SERRGE then refers to the local RRG and identifies the CLB input pin j' corresponding to j in G_L. Let $v_L(j')$ and $e_L(j')$ denote the sets of neighboring vertices and incident edges in the fanout of j' in G_L. SERRGE expands the global RRG according to the following two rules: (1) for each vertex $v' \in v_L(j')$, add a new vertex v to V; (2) for each edge $e' = (j', v') \in e_L(j')$, allocate a new edge $e = (j, v)$ to E.

With the newly expanded vertices and edges, PathFinder can now complete the route to the sink in the BLE, which will be one of the newly added vertices to V. The costs associated with each newly allocated vertex and edge are initialized and updated appropriately; nets routed during the current, and subsequent, PathFinder iterations, may negotiate to use these newly allocated routing resources.

This *dynamic RRG* is a super-graph of the global RRG and a sub-graph of the complete RRG, by construction. In the worst case, the dynamic RRG will grow until it becomes the complete RRG, but this is impractical. The intuition behind this approach is that the Baseline Router preemptively allocations portions of the RRG that are never expanded; SERRGE, in contrast, dynamically allocate the portions of the local RRGs that PathFinder explores, on-demand.

3.3.2. GARBAGE COLLECTION

To limit the memory footprint of the dynamic RRG (and other data structures that are proportional in size), SERRGE includes a dynamic garbage collector. If the dynamic RRG grows more than 30% larger than the global RRG, then the garbage collector deletes all presently unused vertices and edges that were dynamically allocated; this includes all auxiliary data structures associated with each vertex and edge, including Elmore delay trees (see Subsection 3.3.5), traceback information, etc. In other words, RRG growth is used as a proxy for the growth of a much larger set of data structures whose collective memory requirements greatly exceed that of the RRG (i.e., just the vertices and edges) in isolation.

The garbage collector never deallocates vertices and edges that belong to the global RRG. Within each CLB, the garbage collector identifies candidates for deletion by comparing the number of LUT input pins that have been reached thus far with the number of nets that have sinks in each LUT. If the number is equal, then all RRG resources that are incident on the LUT inputs are deallocated; this ensures that routes computed previously during this iteration can be recovered if PathFinder successfully converges. Otherwise, the routing resources are left in-place under the assumption that at least one future net may use them when searching for its sink.

The garbage collector does not consider history costs when deleting RRG vertices; all costs associated with a deleted vertex are lost, and are reset to zero if the vertex is later re-allocated. This may alter the way that PathFinder negotiates under SERRGE, and could yield a different routing results compared to the Baseline router (presuming that the

latter does not incur failed iterations due to exceeding the memory limit). The lost history costs are restricted to vertices and edges that represent the *final link* connecting a routed net to its sink. Even if the history cost is deleted, a net that routes through a re-allocated resource will increase the penalty cost, which would serve to dissuade subsequent nets from using those resources during the current PathFinder iteration.

3.3.3. EXAMPLE

Figure 3-3 illustrates the preceding discussion. PathFinder first searches the global RRG (not shown) from source *s* to a CLB that contains sink *t*. Upon reaching the CLB input pin, Figure 3-3(a) shows that the portion of the RRG corresponding to the CLB has not yet been allocated (gray). SERRGE consults the local RRG to expand the dynamic RRG to fan out from the CLB input pin, as shown in Figure 3-3(b). In Figure 3-3(c), the local route completes using a subset of the newly allocated routing resources. In Figure 3-3(d), the garbage collector claims unused routing resources that SERRGE expanded, but did not use. As shown in Figure 3-3, PathFinder maintains an RRG corresponding to global FPGA routing resources, while selectively expanding the RRG to include a subset of nets that may be used inside of an intra-cluster routing crossbar. When routing a net, the first step is to compute a route from the source *s* to an input of the CLB that contains the sink *t*. The portion of the RRG corresponding to the CLB's intra-cluster routing crossbar is initially not allocated Figure 3-3(a). The fanout of the CLB input found by the global router is allocated and added to the RRG Figure 3-3(b). In this example, the route is completed to the target LUT Figure 3-3(c). Later on, the garbage collector may reclaim CLB routing resources that have been allocated, but were not used Figure 3-3 (d).

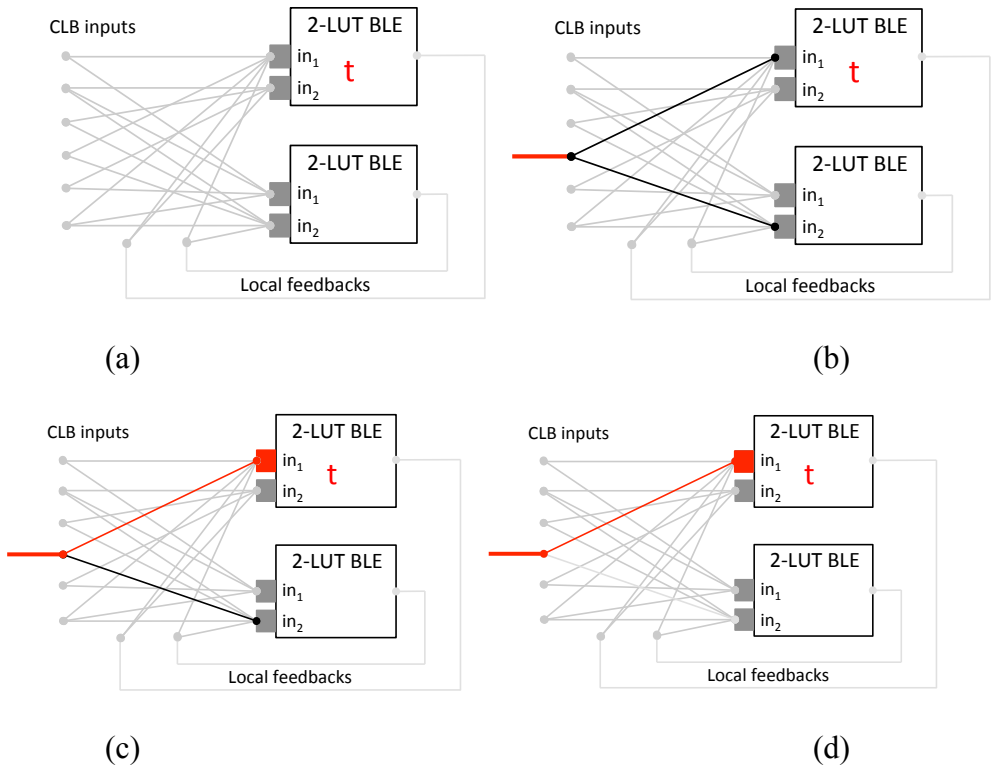


Figure 3-3: Illustration of the basic behavior of SERRGE.

3.3.4. COMPRESSED WIRE-TO-PIN LOOKUP MAP

To further reduce the memory footprint of SERRGE, the extended wire-to-pin lookup map (Section III.B) is converted to a one-dimensional array that exclusively represents routing resources that could possibly be used by the netlist being routed. For example, connections to BLEs within a CLB that are not used (as determined by the placer/packer) are omitted; likewise, CLB I/O pins that interface exclusively with unused BLEs, and CLB sides where all pins connect to unused BLEs, are omitted from the lookup map as well. Figure 3-4 provides pseudocode for the map initialization process.

```
#define USED 1
#define OPEN 0

/* Values vary from CLB to CLB, based on BLE utilization */
int used_CLB_pins = …;
int used_sides = …;
int used_MUXes = …;
int used_LUT_ipins = …;
int N = used_CLB_pins * used_sides * used_MUXes *used_LUT_ipins;

/* Initialize all map entries to OPEN */
int *tracks_to_LUT_pin = calloc(N, sizeof(int));

/* Mark the array entries that are used */
for (int i = 0; i < used_LUT_ipins; i++) {
    int I = (i * used_IIB_MUXes * used_sides * used_CLB_pins);
    for (int j = 0; j < used_IIB_MUXes; j++)  {
            int J = (j * used_sides * used_CLB_pins);
            for (int k = 0; k < used_sides; k++) {
                    int K = (k * used_CLB_pins);
                    for (int L = 0; L < used_CLB_pins; L++)
                            tracks_to_LUT_pin[I + J + K + L] = USED;
            }
    }
}
```

Figure 3-4: Pseudocode to initialize 1-dimensional wire-to-pin map.

3.3.5. ELMORE DELAY TREES

VPR's timing-driven router builds an Elmore delay tree for each vertex discovered during maze expansion, which is saved throughout the search. This increases the router's memory footprint and severely impacts performance.

Unlike the maps in Figure 3-2, the map entries that are marked as being USED (rather than OPEN) depend both on the FPGA architecture and the packing/placement result, and vary from CLB to CLB.

VPR's timing-driven router computes the Elmore delay tree for each vertex v when v is discovered during the search. It uses the tree to compute the term $delay_v$ that contributes to the vertex's cost term c_v, as per Eqs. (2.10) and (2.11), which, in turn, contributes to the cost function f_v, in Eqs. (2.4)-(2.7), i.e., the priority of v when it is inserted into the priority queue. The Baseline Router saves the Elmore delay tree for v, so that $delay_v$ can be updated quickly when necessary (Figure 2-16, Lines 14-17), and for quick access when and if the search removes v from the priority queue during the search (i.e., v has the highest priority among all enqueued vertices).

In contrast, SERRGE discards the Elmore delay tree after $delay_v$ is computed, and re-computes the tree on-demand, when necessary. The performance benefits accrued by the reduced memory footprint outweigh the overhead of re-computing the Elmore delay trees on-the-fly. When and if the router completes successfully, all of the Elmore delay trees are re-computed at the very end, in order to facilitate post-route timing analysis based on the Elmore delay model.

3.3.6. MEMORY LIMIT

Similar to the Baseline router, SERRGE sets a limit on the memory consumption per iteration, for all of the routing resources. If this memory limit is exceeded, then the iteration fails, and any adjusted history costs are propagated to the next iteration. Due to the compressed wire-to-pin lookup map and memory-efficient approach to computing the Elmore delays, SERRGE exceeds the memory less frequently than the Baseline router; despite these efficiencies, SERRGE cannot guarantee that it will always stay within the memory limit, especially when routing large netlists on large FPGAs.

3.4 ROUTING WITH PARTIAL PRE-ROUTING (PPR)

Partial Pre-Routing (PPR) starts by locally routing each CLB (having at least one used BLE) by executing PathFinder on the local RRG, as shown in Figure 3-5(a). Figure 3-5(b) illustrates one of many possible local routing solutions. PPR is then followed by a global routing step that completes each route (i.e., from each source to an appropriate CLB input) using the global RRG. By using one global and one local RRG, PPR avoids the large memory footprint of the Baseline router, and the complications associated with a dynamic RRG and garbage collection required by SERRGE.

The local RRG for each CLB is a bipartite graph. The local routing problem only involves a subset of the nets in the complete routing problem for the entire FPGA, and involves computation of a partial path for each net in the general case. To model the routing problem, a super-source $s*$ is allocated and connected to each CLB input.

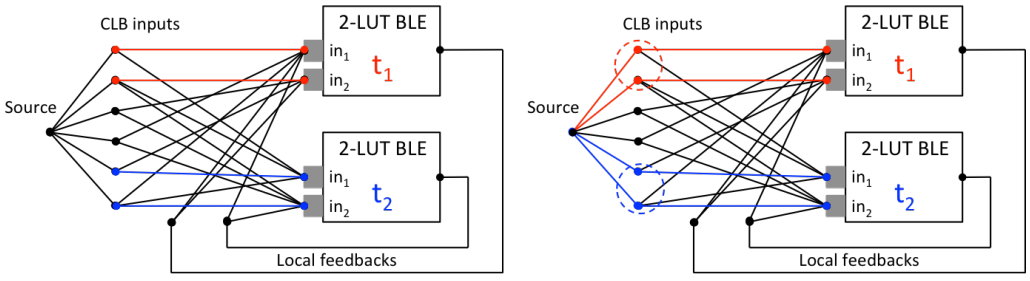


Figure 3-5: PPR Intra-CLB routing operation

Let $N*$ be the set of nets having at least one sink in the CLB. For net $N_i = (s_i, T_i)N*$, let $T_i' T_i$, be the subset of sinks in the CLB. If s_i is a source in the CLB, then net $N_i' = (s_i, T_i')$ is added to the local routing problem instance; otherwise, net $N_i' = (s*, T_i')$ is added

to the local routing problem instance. PathFinder then computes the local routes. If $|T_i'| = 1$ for every net N_i' in the local routing problem instance, then it can be simplified to bipartite matching, which can be solved optimally in polynomial-time using a network flow algorithm. We did not implement this option because PathFinder converged quickly enough in practice.

Solving the local routing problem for each CLB computes all of the intra-cluster routes required for each net. A global routing problem instance using the global RRG is required to compute the inter-cluster routes, subject to the constraints imposed by the local routing results computed by PPR. The constraints can be expressed as CLB input equivalence classes, as shown in Figure 3-5(b). Without loss of generality, the local routing solution computed two CLB inputs that route to BLE t_1. As a consequence, these two CLB inputs become logically equivalent sinks in the global routing problem. Specifically, they become the targets for the two respective nets for which BLE t_1 was the original target. In our experiments, PPR consumed far less memory than either the Baseline router or SERRGE, and did not suffer from memory-related performance issues. Consequently, we did not include SERRGE's memory optimizations for the wire-to-pin lookup maps or Elmore delay trees in PPR; however, they remain nonetheless fully compatible, in principle, with PPR.

PPR starts by routing each intra-cluster routing crossbar individually, finding a set of disjoint paths from the source to the inputs of all BLEs that are used Figure 3-5(a); CLB inputs that are connected to inputs of the same BLEs form equivalence classes that act as new sinks for the global router Figure 3-5(b).

3.5 EXPERIMENTAL SETUP AND METHODOLOGY

3.5.1. EXPERIMENTAL PLATFORM

We implemented the routing algorithms in VPR 5.0 [7], which was the most up-to-date version of VPR when we started this project. VPR 5.0 did not support sparse-intra-cluster routing crossbars. The Beta release of VPR 6.0 [47] featured sparse intra-cluster routing, but did not include a timing-driven router; that feature was added to the official release of VPR 6.0 early in 2012, when the implementation work outlined here was mostly complete. VPR 6.0's router shares many principle similarities with PPR (Section 3.3).

We used a tool described by Lemieux and Lewis [43] to generate routable sparse crossbars with a user-specified population density. We used ABC [6] for logic synthesis and technology mapping, T-VPack for packing, and VPR 5.0 for placement and (timing-driven) routing. All experiments reported here were performed on an Apple iMac featuring a 2.66 GHz Intel Core i5 with 4GB of DDR3 memory, running OS X 10.9.2.

3.5.2. EXPERIMENTAL PARAMETERS

We modeled an FPGA using an architecture configuration file from the iFAR repository [33][34] based on 65nm BPTM technology. Table I lists the architectural parameters that we used. We considered intra-cluster routing crossbars with population densities $p = 40\%$, 50%, and 75%.

K	N	I	Fc_{in}	Fc_{out}	p
6	10	33	0.15	0.10	40%, 50%, 75%

Table 3-1: FPGA architectural parameters

VPR repeatedly routes each benchmark using a binary search to identify the smallest channel width, *Wmin*, for which a legal route can be found. VPR also allows the user to specify a chosen channel width (*W*), and then tries to find a legal route, but may fail. Different routing algorithms may yield different W_{min} values for a given FPGA architecture, benchmark, and placement/packing result; for each, we ran Baseline, PPR, and SERRGE and computed their respective W_{min} values, the largest of which we denote as $Max(W_{min})$. We generate an FPGA with channel with $W = 1.4Max(W_{min})$. We then re-route each benchmark using all three algorithms on this FPGA and present those results. This prevents architectural differences due to varying W_{min} values from skewing the experiments.

PathFinder terminates after a user-specified number of iterations. We set the maximum number of iterations allowed to 300; if PathFinder cannot find a successful route after 300 iterations, then it fails. Since FPGA routing is NP-complete, PathFinder is not guaranteed to find a legal routing solution, even if one exists.

We report the size of the RRG, wire-to-pin lookup maps, and Elmore delay trees for each routing algorithm. The wire-to-pin lookup maps are allocated once remain static throughout routing; the Elmore delay trees grow and shrink dynamically. The RRG is

static under PPR and the Baseline Router, and dynamic under SERRGE. We measure the memory requirement of the static data structures once and profile the size of the dynamic data structures after each dynamic allocation that increases their size. We report the peak memory consumption of these data structures for each benchmark and architecture during the runtime of the routers.

3.5.3. TIMING AND AREA MODELS

Our timing model was similar to VPR 5.0. We added models to account for delays inside of the CLBs. The timing graph is generated such that every CLB or LUT input pin becomes a timing node. Timing edges represent connectivity between pins, and delays are marked on edges, not nodes.

The area model sums the aggregate areas of the number of minimum-width transistors required to place and route a circuit on in VPR; we did not modify VPR's counting method. We added extensions to account for the intra-cluster routing crossbar area, which depends on its population density.

We employed the basic techniques that were used in VPR to estimate the silicon area occupied by each multiplexer and wire in the CLB. We assume that a minimum width transistor takes 1 unit of area. A double-width transistor takes twice the diffusion width, but the same spacing, so we assume it takes 1.5x the area of a minimum-width transistor.

Buffer sizes are calculated based on the drive strength requirements and depend on the fan-out of the buffer. VPR uses 4x the minimum size, which we have adopted for general buffers. We sized the CLB input buffers using the approach used by Lemieux et al. [3], where the drive strength is at least 7x and at most 25x the minimum size.

We model an FPGA with single-driver wires; each wire segment begins with a multiplexer followed by a driver. We attempt to judiciously select the multiplexer size depending on the number of inputs. One-level multiplexers are used when there are 4 or fewer inputs, and more levels are used when the number of multiplexer inputs increases.

3.5.4. BENCHMARKS

We selected 10 of the largest IWLS benchmarks [17] for use in our experiments; their summary is given in Table 3-2.

Benchmark	Array size	Nets	CLBs
		5097	
ac_ctrl	48x48	5800	5008
aes_core	33x33	1569	2518
des_area	16x16	4464	695
mem_ctrl	27x27	8016	3158
pci_bridge32	74x74	923	7815
spi	13x13	2509	712
systemcaes	21x21	1068	2173
systemcdes	12x12	5154	706
usb_funct	40x40	1043	4429
wb_conmax	47x47	0	6297

Table 3-2: 10 of the largest IWLS Benchmarks

VPR generates a custom FPGA that is sized for each benchmark. The second column of Table II lists the dimensions of the FPGA generated for each benchmark (e.g., an *MxN* array of CLBs). The third and fourth columns list the number of nets and CLBs used in each benchmark for an FPGA architecture with parameters *N=8*, *K=6*, and *I=27*, i.e., each CLB contains eight 6-LUTs and has 27 input pins.

Some of the IWLS benchmarks are I/O bound, rather than logic bound. In these cases, the number of I/Os per physical pin on the perimeter of the FPGA dictates the dimensions. When this occurs, VPR generates an FPGA with far more LUTs/CLBs than are necessary to realize each benchmark, and LUT/CLB utilization is relatively low as a result.

For each benchmark and FPGA, we generate 10 placements by varying the random number seed used in VPR's simulated annealing-based placer. For each placement, we then route the benchmark using PPR, SERRGE, and the Baseline router. For each data point (benchmark/FPGA/router), the results reported are the averages over the ten placements.

3.6 EXPERIMENTAL RESULTS

3.6.1. W_{MIN} AND ROUTABILITY

Figure 3-6 reports the W_{min} values obtained by routing each benchmark/FPGA combination using PPR, SERRGE, and the Baseline Router.

Surprisingly, PPR yields the lowest overall W_{min} values across all benchmark/architectures, with the exception of *des_area* for the FPGA with intra-cluster routing crossbar population density $p = 75\%$.

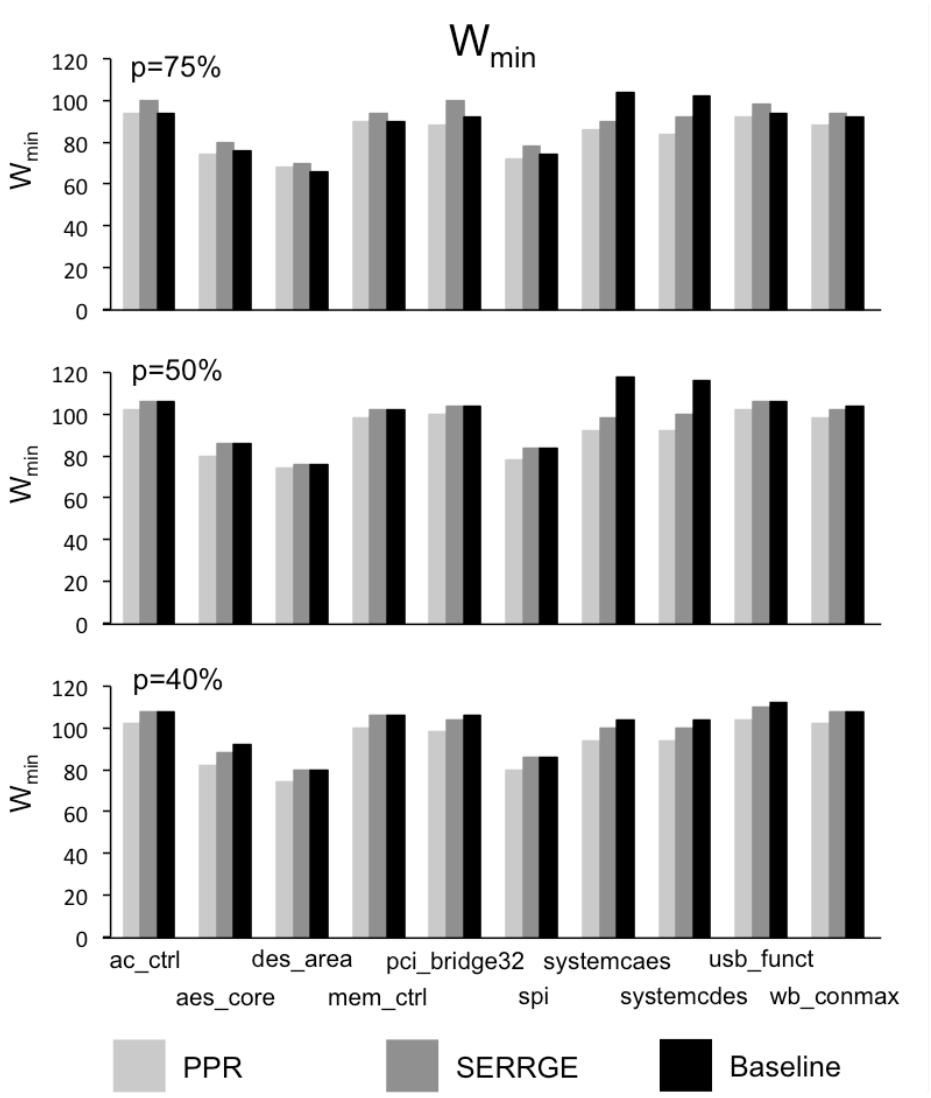


Figure 3-6: W_{min} for the ten largest IWLS benchmarks

These results indicate that PPR is more likely than SERRGE or the Baseline Router to find a legal routing result. When considering W_{min} as a proxy for routability, it is important to note that our experiments use VPR's timing-driven router; experiments have been published which demonstrate that VPR's routability-driven router, which does not employ the Elmore delay model, tends to yield lower W_{min} values than the timing-driven router [7, Table 4.8].

3.6.2. CRITICAL PATH DELAY

Figure 3-7 reports the critical path delays obtained by routing each benchmark/FPGA combination using PPR, SERRGE, and the Baseline Router. PPR is competitive with SERRGE and the Baseline Router in many cases; the biggest disparity in critical path delay is *3.43 MHz (143.88 MHz to 140.45 MHz)* for the *pci_bridge32* benchmark for the FPGA with intra-cluster routing crossbar population density *p = 40%*.

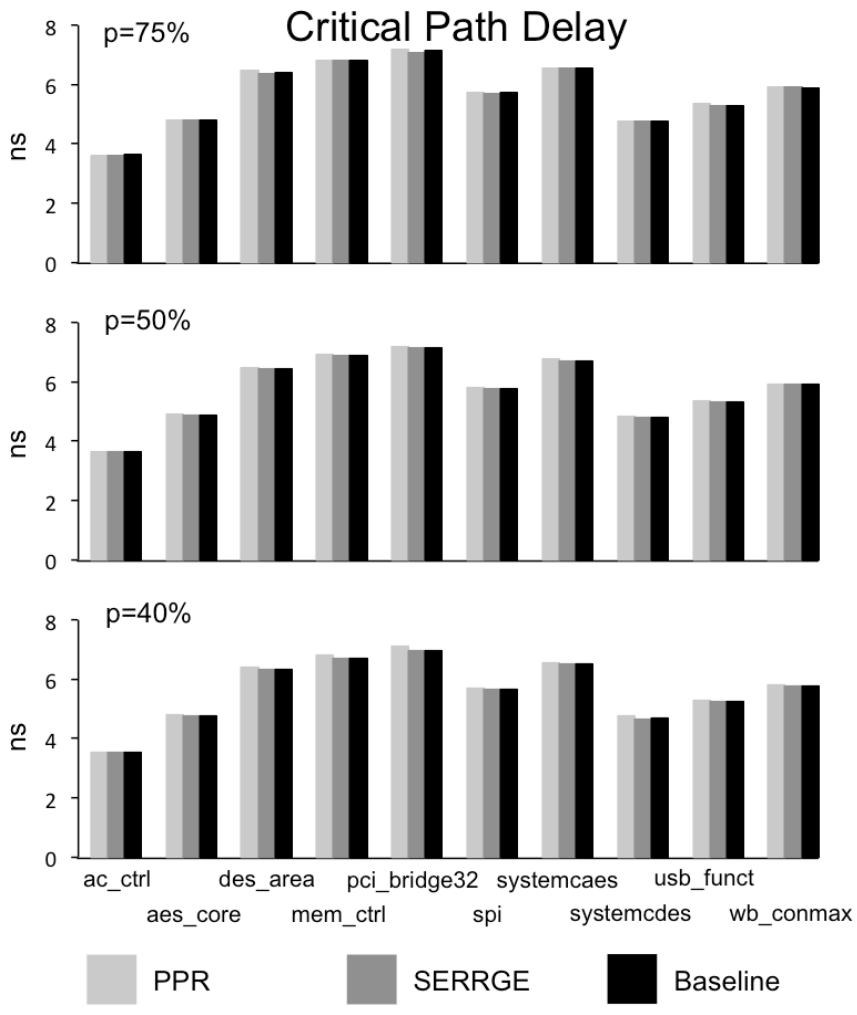


Figure 3-7: The critical path delay for the ten largest IWLS benchmarks

PPR's pre-routing phase does constrain the search space for negotiation, which accounts for the cases where SERRGE and the Baseline Router achieve lower critical path delays. SERRGE and the Baseline Router permit PathFinder to negotiate for routes at the CLB inputs and within the intra-cluster routing crossbar, facilitating discovery of faster routes.

3.6.3. RUNTIME AND NUMBER OF PATHFINDER ITERATIONS

Figure 3-8 reports the runtimes of PPR, SERRGE, and the Baseline Router for all benchmark/FPGA combinations. The runtime in measured on the ten largest IWLS benchmarks (Table 3-1) placed-and-routed on an FPGA with parameters specified in Table 3-2. Results are reported for devices with intra-cluster routing crossbar population densities of $p = 75\%$ (top) 50% (middle) 40% (bottom).

PPR is uniformly the fastest, followed by SERRGE, and Baseline. All three routing algorithms tend toward faster convergence at lower intra-cluster routing crossbar population densities.

Figure 3-9 reports the number of PathFinder iterations required for PPR, SERRGE, and the Baseline Router to converge for each benchmark/FPGA combination. With one exception (*wb_conmax* for an FPGA with intra-cluster routing crossbar population density $p = 50\%$), PPR requires the fewest iterations, followed by SERRGE, and then the Baseline Router. Reducing the intra-cluster routing crossbar population density marginally reduces the number of iterations.

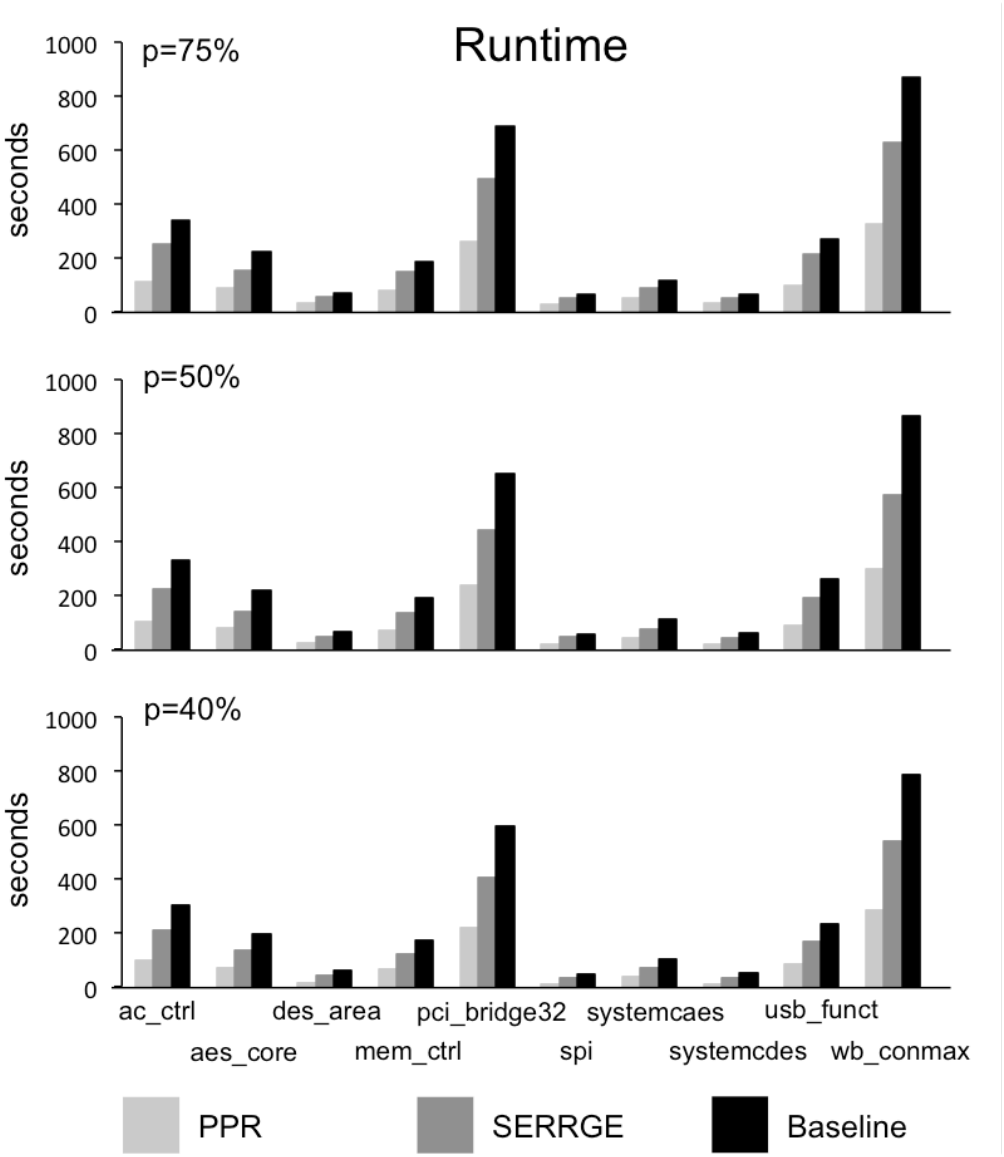


Figure 3-8: Runtime (seconds) for the ten largest IWLS benchmarks

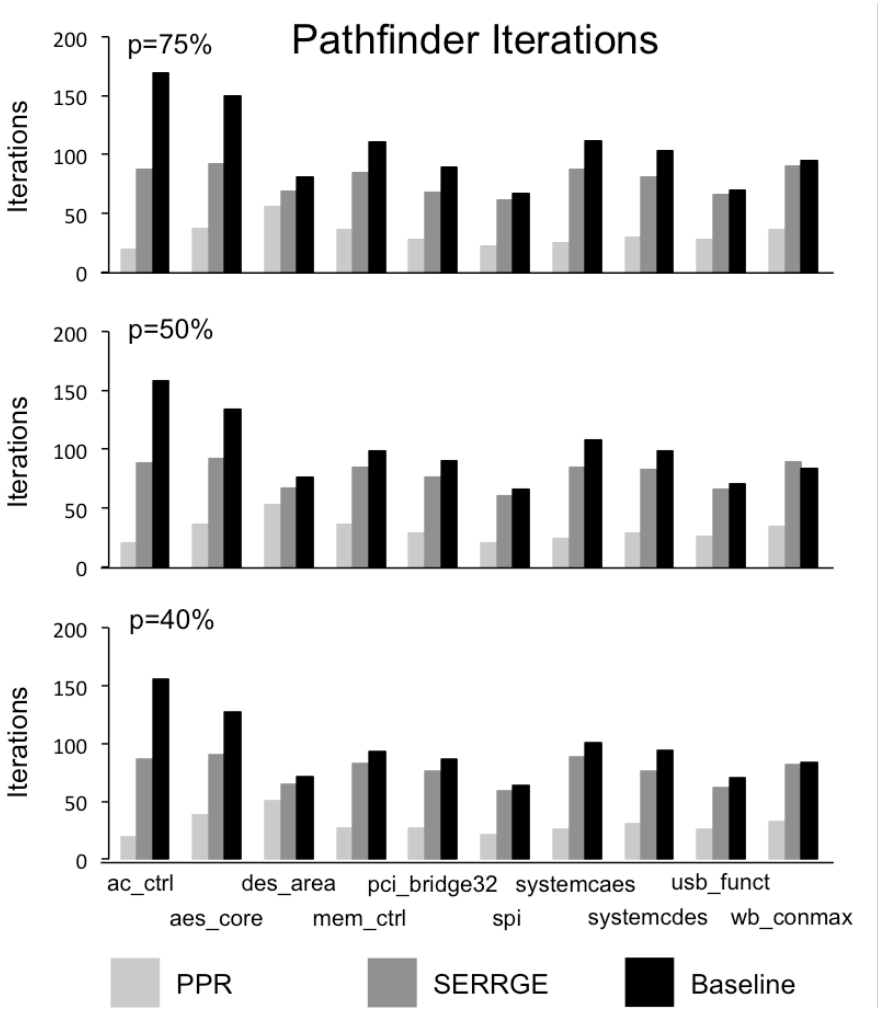


Figure 3-9: The number of PathFinder iterations for the ten largest IWLS benchmarks

PPR requires far fewer iterations to converge than SERRGE or the Baseline Router. This is due primarily to two factors: (1) PPR's restricted search space; and (2) PPR's more efficient usage of memory, which limits the number of iterations that fail due to exceeding the memory limit. These factors, correlate directly to the reduced runtimes reported in Figure 3-10.

3.6.4. MEMORY CONSUMPTION

Figure 3-10 reports the peak memory consumption of PPR, SERRGE, and the Baseline Router for each benchmark/FPGA combination. The Elmore delay trees, which are specific to VPR's timing-driven router [7, Section 4.4], consume more than twice as much memory than the wire-to-pin lookup maps and RRG combined, and the wire-to-pin lookup maps consume significantly more memory than the RRG.

SERRGE offers a marginal improvement in peak memory consumption compared to the Baseline Router, due primarily to its compressed wire-to-pin lookup map and re-computation, rather than storage, of the Elmore delay trees. PPR consumes far less memory than either SERRGE or the Baseline Router, because the global RRG, which is smaller than SERRGE's dynamic RRG at peak memory consumption and the Baseline Router's complete RRG, has fewer vertices and edges, and thus stores fewer Elmore delay trees. Also, PPR's wire-to-pin lookup maps stop at the CLB inputs, while SERRGE and the Baseline Router must route all the way to BLE input pins.

The peak memory consumption requires hundreds of MBs, as reported in Figure 3-10. Intel's i5 processors have 3-6 MB of L3 cache, and the i7 family has 4-8 MB. Although the routers may exhibit some temporal and/or spatial locality, the working set exceeds L3 cache capacity, degrading performance.

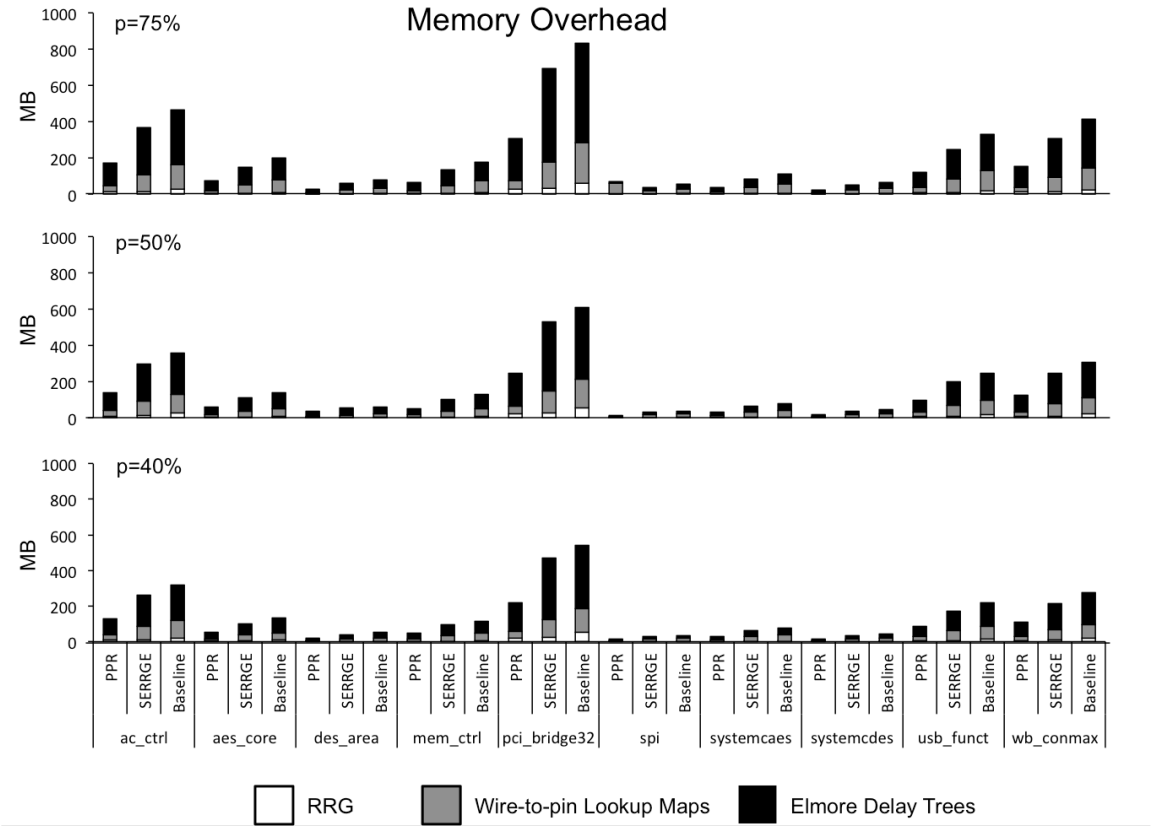


Figure 3-10: Memory overhead for PPR, SERRGE, and the Baseline routers

3.7 SUMMARY

This chapter introduced PPR and SERRGE to reduce the runtime and memory footprint of FPGA based on the PathFinder negotiated congestion algorithm [50], as implemented in VPR [46]. We first implemented a Baseline Router, which is an extension of VPR's timing-driven router with larger data structures that are required to represent intra-cluster routing crossbars and expose this new information to the router. We then implemented SERRGE as a more memory efficient variation of the Baseline Router, which employs compressed data structures and online garbage collection. Lastly, we implemented PPR, which divides routing into local and global phases, yielding a smaller memory and rapid

convergence, but at the cost of a drastically reduced search space. PPR offers the best overall routability (e.g., W_{min} values), fastest running times, and smallest memory footprint, while SERRGE tends to find routing solutions with the lowest overall critical path delays. If router runtime is a premium, then PPR should be used; if critical path delay is more important, then SERRGE is preferable; in the vast majority of our experiments, PPR and/or SERRGE outperformed the Baseline router for all metrics of interest, as reported in Figs. 3.8-3.11.

The next chapter investigates the use of dynamic multiplexing to configure the logic block to efficiently implement floating point shifters. It also describes the CAD tool support for realizing the hybrid logic blocks.

Chapter 4. A NEW HYBRID LOGIC BLOCKS

This chapter introduces a new type of logic blocks that can be configured to efficiently implement shifting operations for floating points mantissa alignment and normalization; or as a regular logic block, if shifting operations are not required. We first describe the architectural modifications required in the intra-cluster routing fabric to realize this type of logic blocks. We then show the efficiency of this logic block architecture for implementing shifters for floating-point mantissa alignment and normalization. Finally, we investigate the CAD support for realizing this logic block architecture and illustrate the necessary modifications to the physical design tools (placement and routing) in order to support this architecture.

4.1 INTEGRATING SD-MUXES INTO FPGAS

In Chapter 2 we have described the operation of SD-MUXes and the benefits of using them. This section describes the necessary architectural enhancements to FPGAs to integrate SD-MUXes into the routing fabric. Starting with an overview a typical FPGA architecture (Section 2.1), we consider two locations in the FPGA to introduce the SD-MUXes (Sections 2.2 and 2.3). Lastly, we introduce the macro-cell and describe how a standard FPGA CAD flow can be modified to achieve routability (Section 4.4).

This work targets an FPGA architecture based on the *Versatile Place and Route*

(VPR) tool, which is publicly available from the University of Toronto [46]. The user specifies several architectural parameters in a configuration file. VPR generates an FPGA architecture based on these parameters.

The intra-cluster routing is a crossbar that connects *I* inputs and *N* local feedbacks to the *KxN* LUT inputs in the CLB. VPR 5.0 implements intra-cluster routing as a full crossbar, which provides a connection between every CLB input and LUT input. Full crossbars are costly in terms of area and power, but guarantee routability: i.e., any combination of signals routed to CLB inputs can be routed to any desired combination of LUT inputs. Highly routable sparse crossbar topologies for intra-cluster routing have also been investigated in recent years [20][43].

Ahmed and Rose determined that the ideal number of CLB inputs is $I = K\text{x}(N+1)/2$, which is less than the total number of LUT inputs, $K\text{x}N$. This suffices because many signals fan-out to multiple LUT inputs within a CLB after the FPGA has been configured. As each CLB input (other than LUT feedbacks) is driven by a $W\text{x}F_{cin}{:}1$ multiplexer, reducing the number of CLB inputs reduces the overall cost of the C Block, at the expense of some flexibility. In other words, *N* independent *K*-input logic functions cannot be packed into a CLB due to I/O limitations, despite the fact that the CLB has sufficient LUT capacity.

Recall that our goal is to replace static multiplexers in the routing network with SD-MUXes. There are two locations where this is possible: the C Block (input), and intra-cluster routing, as discussed in the next two subsections.

4.1.1. INTEGRATING SD-MUXES INTO THE C BLOCK

Example 1. To illustrate the integration of an SD-MUX into a C Block, let us consider a conditional swap, which has three inputs, I_0, I_1 and c, and two outputs, J_0 and J_1. The operation is:

$J_0 = c\ ?\ I_1 : I_0$, $J_1 = c\ ?\ I_0 : I_1$ (1) Figure 4 depicts a portion of the C Block that has been modified with two SD-MUXes to implement the conditional swap. static multiplexer provides the control bit, while two SD-MUXES compute J_0 and J_1. In this particular example, $W = 8$ segments per channel and $Fcin = 0.5$, i.e., each C Block multiplexer connects to 4 wires in the channel. Each of the three 4:1 multiplexers in the C Block are implemented using three 2:1 multiplexers; two of the 2:1 multiplexers have been replaced with SD-MUXes in **Figure 4-1**.

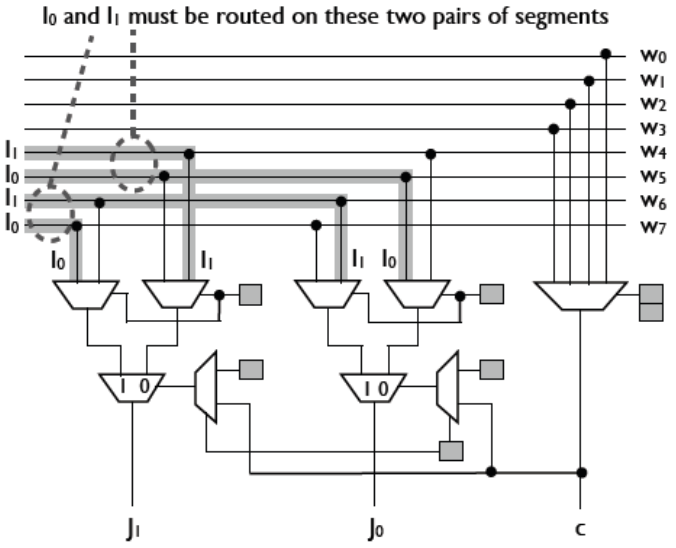


Figure 4-1: A C Block modified to implement a conditional swap by introducing two SD-MUXes.

The C Block in **Figure 4-1** imposes routing constraints that must be satisfied in order

to deliver input signals I_0 and I_1 in the correct order to the SD-MUX inputs. In particular, I_0 and I_1 must be routed on routing segments w_6 and w_7; the order is irrelevant, i.e., either I_0 can be routed on w_6 and I_1 on w_7, or vice-versa; similarly, I_0 and I_1 must be routed on w_4 and w_5 as well. The condition bit, c, has greater flexibility: it can be routed on w_0, w_1, w_2, or w_3.

The placer and router must satisfy these constraints. Let F_0 and F_1 be K-input logic functions that compute conditional swap inputs I_0 and I_1. F_0 and F_1 must be synthesized on LUTs whose outputs collectively drive a subset of the wires that satisfy the aforementioned constraints. Moreover, this assumes that such a combination of LUTs actually exists. Although it may be possible to satisfy this constraint for a 3-input conditional swap operation, it will be much more difficult to satisfy for a 24- or 27-bit shifter. *Example 2*. Consider a 4-bit left shift with rotation. The inputs are

$I_0 \ldots I_3$ and the outputs are $J_0 \ldots J_3$; two control bits c_0 and c_1 specify the shift amount (0-3 bit positions). Once again, we assume that $W = 8$ and $F_{cin} = 0.5$, and the C Block contains 4:1 multiplexers.

As the shifter has four data inputs rather than two, each of the four data inputs, $I_0 \ldots I_3$ must connect to *exactly one* input of each C Block multiplexer in a pre-determined pattern. **Figure 4-2**(a) depicts a portion of the C Block that produces the lower-order data outputs, J_0 and J_1; however, the interconnection topology does not allow the design to be satisfied due to conflicts on the routing segments. For example, the multiplexer that produces J_0 requires I_0 to be routed on segment w_4, while the multiplexer that produces J_1 requires I_1 to be routed on the same segment concurrently. Similar conflicts occur on

segments w_5, w_6, and w_7. In contrast, **Figure 4-2**(b) depicts an interconnect topology that eliminates the routing conflicts; of course, this topology *only* satisfies a 4-bit left shift with rotation, and would not necessarily be helpful for some other type of multiplexer-based circuit. As mentioned earlier, dynamic multiplexers impose strict ordering constraints on the signals that are connected to their inputs. The examples shown in Figures 4 and 5 demonstrate that these constraints are propagated into the global routing network when SD-MUXes are integrated into the C Block. The ability to implement very simple multiplexing circuits, as shown in Figures 4-1 and 4-2, is dependent on the interconnect topology between the CLBs and the routing network; this interconnect topology depends on parameters W, Fc_{in} and Fc_{out}, and the algorithm that generates the routing network from these parameters. Although it may be possible to modify the routing network generation algorithm to favor certain interconnect topologies, it is difficult to determine whether the basic idea will generalize to larger structures. For example, in a 27-bit shifter, input bit I_0 will fan out to 27 outputs, $J_0 \ldots J_{26}$. This means that a single routing segment or a subset of segments driven by the same BLE must individually or collectively fan-out to 27 pre-specified C Block SD-MUX inputs, all in close quarters. Similarly I_1 will need to fan-out to 26 pre-specified C Block SD-MUX inputs, etc. Moreover, this must be done with parameters that are representative of commercial FPGAs, e.g., $N = 10$, W $= 300$, $Fc_{in} = 0.15$, and $Fc_{out} = 1/N = 0.1$. The likelihood of success in this case is too low to be considered realistic; consequently, we conclude that the C Block is not a particularly promising location to integrate SD-MUXes into the routing fabric.

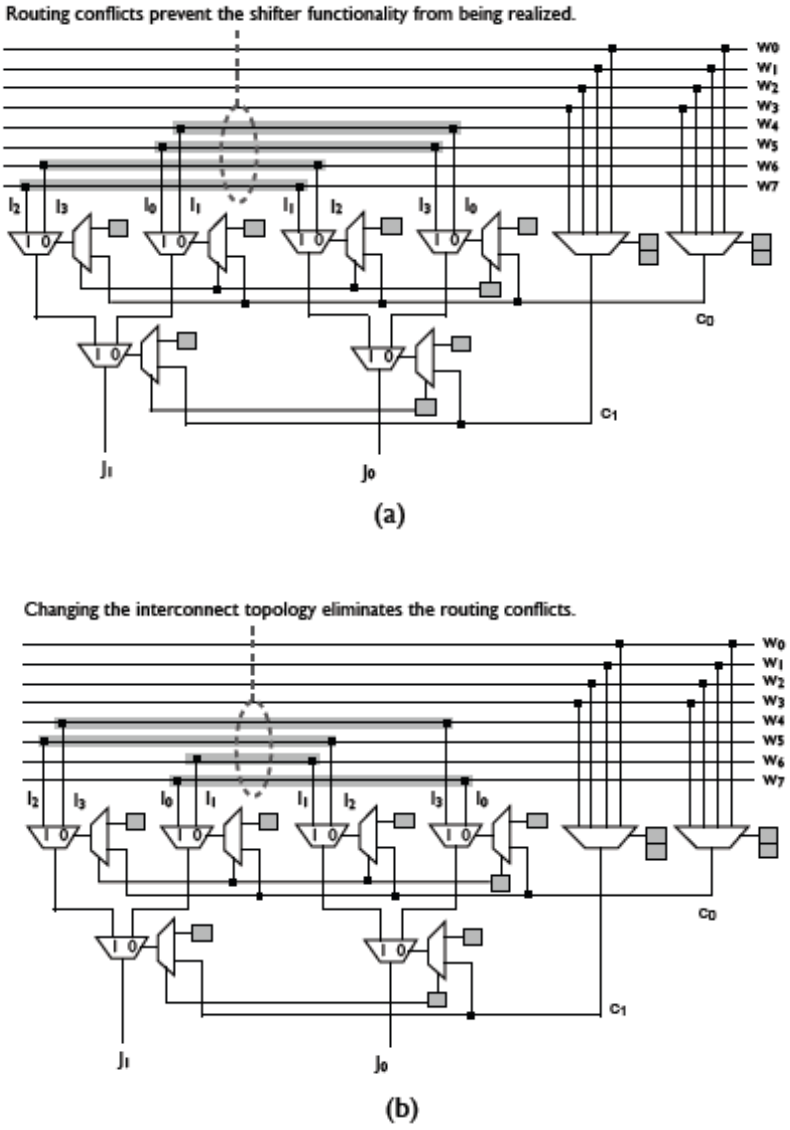


Figure 4-2: A conflict in the interconnect topology.

Figure 4-2 shows how a conflict in the interconnect topology makes it impossible to implement a 4-bit rotator using SD-MUXes in the C Block (a); changing the interconnect topology can eliminate the conflict (b).

4.1.2. INTEGRATING SD-MUXES INTO INTRA-CLUSTER ROUTING

Alternatively, we can introduce SD-MUXes into the intra-cluster routing instead of the C Block. The primary advantage of this approach is that it eliminates the routing constraints that arise due to the interconnection topology between the routing segments and the C Block. Instead, the interconnection topology constraints are internal within the intra-cluster routing. Signals that drive a specific SD-MUX input for dynamic multiplexing, as shown in **Figure 4-3**, are routed to pre-selected CLB inputs. Each pre-selected CLB input connects to one of the SD-MUX inputs: some to the data inputs, and others to the selection inputs. **Figure 4-3** shows an example of a 4:1 SD-MUX integrated into the intra-cluster routing; a significant portion of the intra-cluster routing is omitted from Figure 6 to conserve space. Two CLB inputs provide dynamic control (they may also drive other multiplexers, which are not depicted in the figure); control signals *c0* and *c1* must be routed to these two inputs. The other four CLB inputs drive the data inputs of the SD-MUX. The input signals are routed to these four CLB inputs in a specific order, e.g., the SDMUX selects input I_0 if $c1c0 = 00$. Thus, the connection topology between CLB inputs and SD-MUX inputs determines which signals must be routed to each pre-selected CLB input. In **Figure 4-3**, any 4-input multiplexer can be realized by permuting either the control or the data bits; however, additional restrictions are imposed when we consider multiple-output functions because each CLB input may connect to multiple SD-MUX inputs.

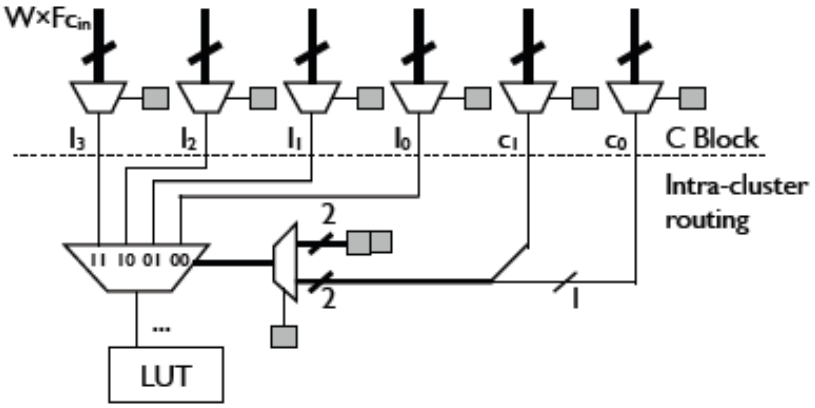


Figure 4-3: Integrating an SD-MUX into intra-cluster routing imposes a strict ordering on the signals that are routed to the CLB inputs.

Example 3. Let us reconsider the conditional swap operation from Example 1; this time, we want to implement it using SD-MUXes in the intra-cluster routing rather than the C Block. Figure **Figure 4-4**(a) shows an initial attempt. Due to the interconnection topology within the intra-cluster routing, both SD-MUXes conditionally select the same input bit, i.e., they both compute logic function J_0.

Figure 4-4 shows how SD-MUXes can be integrated into the intra-cluster routing; the interconnect topology may force both SDMUXes to implement the same logic function when configured to implement dynamic control (a); rearranging the topology enables the SD-MUXes to implement different functions (b). By swapping the order of I_0 and I_1 at the CLB inputs, then this intra-cluster routing topology would compute J_1, rather than J_0; however, by changing the topology, as shown in **Figure 4-4**(b), the two SD-MUXes compute logic functions J_0 and J_1, respectively. In this case, swapping the order of I_0 and I_1 at the inputs would likewise swap the order of J_0 and J_1 at the outputs.

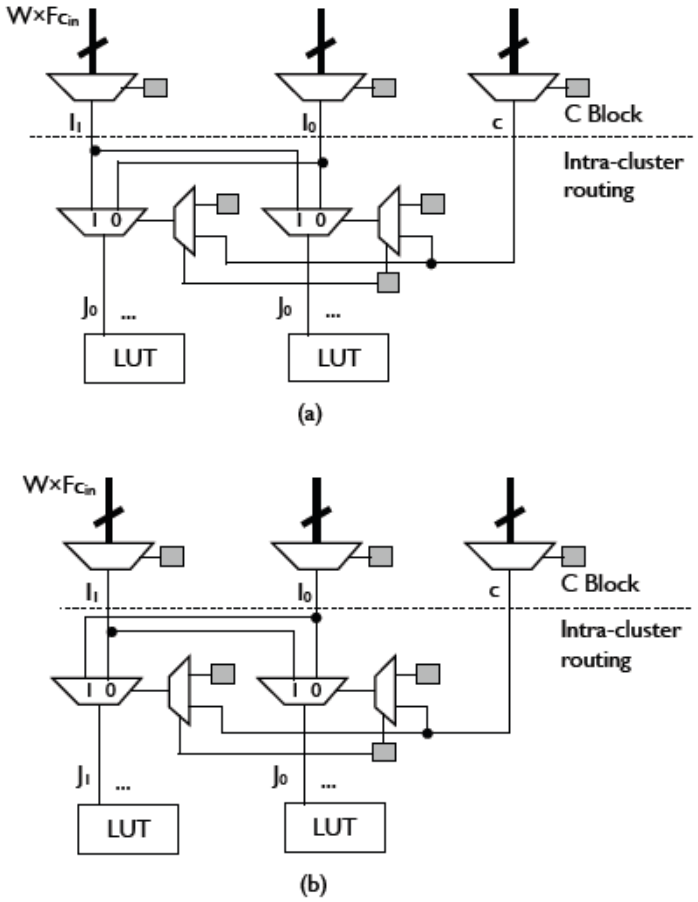


Figure 4-4: Integrating SD-MUXes into intra-cluster routing.

Example 4: Figure 4-5 illustrates intra-cluster routing with SDMUXes that can implement a 4-bit left shifter (bits shifted in are set to zero). The basic interconnection pattern shown here easily generalizes to a larger shifter sizes. In this case, the SD-MUXes implement all of the shifting functionality; the LUTs are configured to pass the SD-MUX outputs through unmodified.

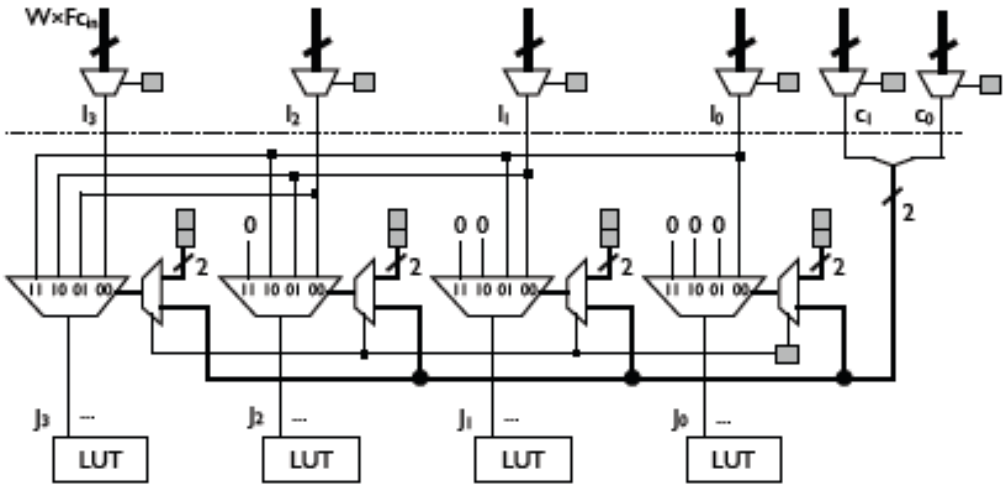


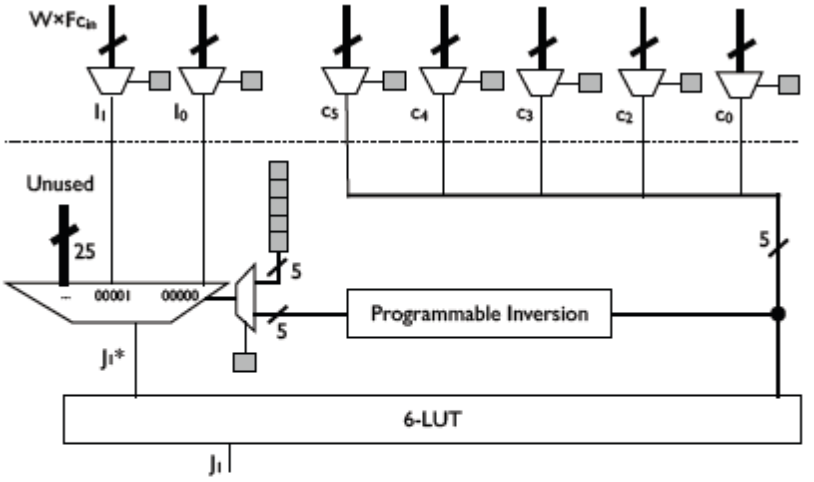
Figure 4-5: Intra-cluster routing with SD-MUXes modified to support a 4-bit left shift.

In Figure 4-5, many of the SD-MUX inputs in are '0' bits. It is not immediately clear how these bits should be handled. It would be unrealistic to extend some of the SD-MUXes to account for a large number of '0' bits, e.g., in a *K*-bit shifter, the SD-MUX that computes least significant output bit *J0* requires *K-1* '0' bits, the SD-MUX that computes *J1* requires *K-2* '0' bits, etc.; the area overhead required to support larger shifters would be prohibitive.

Another issue is that the routing network may invert signals en route. The LUT sink is usually reprogrammed to compensate if some of its inputs arrive with the wrong polarity. SD-MUXes, however, are not programmable in this respect. One possibility is to add programmable inversion at the shifter inputs; however, this incurs significant area overhead. Another option is to reprogram the previous layer of LUTs that generate the shifter inputs to compute the complement of its logic function; however, this logic layer may have a large fan-out, where some fan-out bits are inverted and others are not, rendering this approach ineffective.

We can solve both the ‘0’ SD-MUX input bit problem and the inversion problem by using LUTs in conjunction with the SDMUXes.

Each SD-MUX output drives a LUT input; we can then route the control bits to the remaining LUT inputs. The LUT is then programmed to invert the SD-MUX output if the selected input arrives in inverted form. The LUT is also programmed to output a ‘0’ for the appropriate control bit combinations (e.g., *c1c0 = {01, 10, 11}* for *J0* in Figure 4-5), which eliminates the need to route ‘0’ bits to the SD-MUX inputs.



I0 Not Inverted I1 Not Inverted							
c4	c3	c2	c1	c0	J1*	J1
0	0	0	0	0	0	0
0	0	0	0	0	1	1
0	0	0	0	1	0	0
0	0	0	0	1	1	1
--	--	--	--	--	--	0

I0 Inverted I1 Not Inverted						
c4	c3	c2	c1	c0	J1*	J1
0	0	0	0	0	0	1
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	0	1	1	1
--	--	--	--	--	--	0

I0 Not Inverted I1 Inverted						
c4	c3	c2	c1	c0	J1*	J1
0	0	0	0	0	0	0
0	0	0	0	0	1	1
0	0	0	0	1	0	1
0	0	0	0	1	1	0
--	--	--	--	--	--	0

I0 Inverted I1 Inverted						
c4	c3	c2	c1	c0	J1*	J1
0	0	0	0	0	0	1
0	0	0	0	0	1	0
0	0	0	0	1	0	1
0	0	0	0	1	1	0
--	--	--	--	--	--	0

Figure 4-6: LUT used in conjunction with an SD-MUX

The CLBs in modern high performance FPGAs contain 6-LUTs; this limits the number of control bits that can be supported using this approach to 5 or less, which, in turn, limits the maximize SDMUX size to 32:1. This suffices for the 24- and 27-bit

shifters used for single-precision floating-point mantissa alignment and normalization. **Figure 4-6** illustrates the preceding discussion for the second least significant bit, J_1 of a 27-bit shifter.

As can be seen from Figure 4-6, a LUT used in conjunction with an SD-MUX solves the problems of inverted input bits and generates '0' outputs when appropriate. This example is the second least significant output bit, J_1, of a 27-bit left shifter. Four truth tables are possible, depending on whether I_0 and I_1 are inverted. Programmable inversion is necessary for the five control bits.

CLB parameters also limit the size of the SD-MUXes that can be introduced. The intra cluster routing has a total of $I+N$ inputs and NK outputs. The inputs are the I CLB inputs provided by the C Block plus N LUT feedbacks from within the CLB. The cluster contains N K-LUTs; each LUT input is an output of the intra-cluster routing. A typical modern high-performance FPGA has $N = 8$, $K = 6$, and $I = K(N+1)/2 = 27$, using the formula provided by Ahmed and Rose [1]. To support a 27-bit shifter, we need to increase I to 32, to account for the control signals.

In VPR 5.0, the intra-cluster routing is a full crossbar. Given these parameter values, the intra-cluster routing would be composed of 48 40:1 multiplexers. Modern FPGAs, however, use sparsely populated crossbars [20] [43]. Depending on the population density of the sparse crossbar, the multiplexers may be smaller than 27:1. In this case, we would either need to limit the shift amount in accordance with the multiplexer size, or introduce SD-MUXes that are larger than the pre-existing static multiplexers; this latter option is unfavorable, because it introduces asymmetry in terms of delays: i.e., the delay

through a statically configured SD-MUX is greater than the delay through a standard static multiplexer, which could affect performance and complicate routing.

The interconnection topology (i.e., which CLB inputs connect to exactly which SD-MUX inputs) has a significant impact on our ability to implement shifters in the intra-cluster routing; this was illustrated quite clearly by **Figure 4-4**. Figure 4-5 illustrates the general interconnect topology pattern required for a left shifter (which easily generalizes to more than 4 inputs), and a left shifter can implement a right shifter by reversing the order of the inputs.

Shifters that perform rotation (e.g., **Figure 4-2**) require a different topology as they do not shift-in zeroes. To summarize, the topology must account for ordering constraints on SD-MUX inputs in order to ensure correctness. Lastly, we do not advocate the introduction of SD-MUXes into every CLB, as the vast majority of CLBs in a given FPGA will not be configured to implement dynamic multiplexing circuits in most realistic designs. CLBs containing SD-MUXes are a new form of heterogeneity, similar in principle to the introduction of DSP blocks and block RAMs in past FPGAs. As a rough estimate, we suggest at most 10% of the CLBs in an FPGA should be enhanced with SD-MUXes, and that those that are enhanced should be laid out in columns within the FPGA; the column-based layout echoes the way that DSP blocks and block RAMs are currently laid out in FPGAs, and therefore makes intuitive sense.

4.2 ENSURING ROUTABILITY WITH MACRO-CELLS

Consider a 27-bit shifter implemented with SD-MUXes integrated into the intra-cluster routing. In accordance with prior notation, let I_0 ... I_{26} and J_0 ... J_{26} denote the shifter inputs and outputs, and let c_0 ... c_4 denote the control bits. This is a total of 32 inputs (including control bits) and 27 outputs. Eight CLBs, CLB_0 ... CLB_7 realize the shifter. LUT *Li* computes shifter input *Ii*, LUT *Si* computes shifter output *Ji*, and LUT *Ci* computes control bit *ci*.

Figure 4-7 depicts the interconnection pattern for the 27-bit shifter. The structure depicted in **Figure 4-7** is called a *macro-cell*, because the LUTs and CLBs are pre-placed and routed. Without loss of generality, if the placer (generally an iterative improvement algorithm) randomly moves L_5 to a new CLB, the likelihood is quite small that a legal route will be found that delivers shifter input I_5 to the pre-specified CLB inputs in CLB_5, CLB_6, and CLB_7. By fixing the locations of the LUTs relative to one another in the macro-cell, routability is achieved.

4.3 CAD SUPPORT FOR MACRO-CELLS

We used VPR 5.0 [46] for architectural simulation, placement, and routing, T-VPack for packing [22], and ABC for logic synthesis and technology mapping [6].

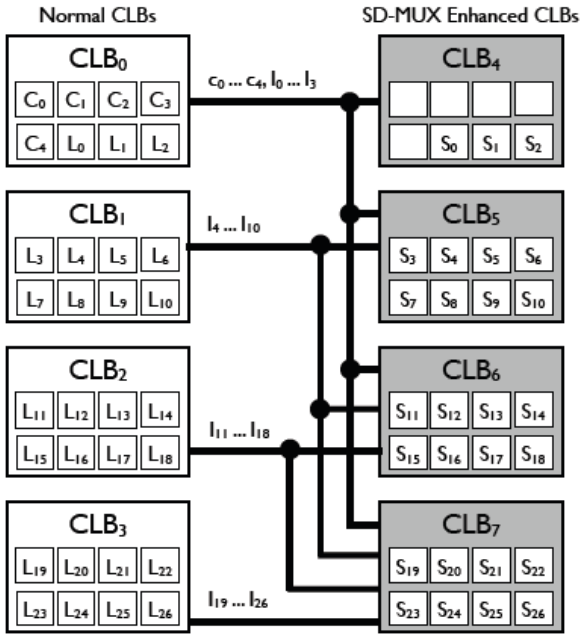


Figure 4-7: A macro-cell for a 27-bit shifter.

4.3.1. PROGRAMMING MODEL, ASSUMPTIONS, AND TECHNOLOGY MAPPING

We assume that the programmer will add annotations to the HDL code to specify when to configure the programmable macro-cell as a shifter, similar to how DSP blocks and carry chains are used. The technology mapper explicitly binds the annotated shifters to macro-cells rather than mapping them to LUTs. Large shifters are decomposed into smaller ones if macro-cell capacity is exceeded.

Next, we extract the layer of LUTs that precedes each shifter, e.g., LUTs L_0 ... L_{23} in **Figure 4-7**. The structure of the macro-cell effectively pre-packs, pre-places, and pre-routes these subcircuits.

4.3.2. MACRO-CELL PLACEMENT AND ROUTING

VPR's router, which is based on PathFinder [23], assumes that CLB intra-cluster routing is a full crossbar. Any path from the source to a CLB input can route a net: the crossbar connects all CLB inputs to all LUT inputs. We modified VPR to allow the user to specify specific CLB inputs pins as targets for certain sinks. VPR can find a legal route for a macro-cell, establishing a path from the LUT source that computes each net to all of its pre-specified inputs in the second macro-cell layer. VPR successfully routed 24- and 27-bit shifters in macro-cells using this approach.

Macro-cells are placed-and-routed offline, prior to the rest of the circuit. Placement of the shifter onto SD-MUXes within the macro-cell is deterministic. Placement of the LUT layer preceding the shifter is more flexible: any placement that successfully routes all nets within the macro-cell suffices. We try to pack the LUTs tightly into a small number of CLBs in the vicinity of the shifter.

4.3.3. GLOBAL PLACEMENT AND ROUTING

Extensive modifications were made to VPR's placer [16] in order to handle macro-cells. The input is a netlist, which may or may not contain macro-cells, and an architectural description of the FPGA in which certain columns have been annotated to indicate CLBs that have been enhanced with SD-MUXes. Each shifter in the netlist, along with the layer of preceding LUTs, is placed onto a macro-cell. Each macro-cell is routed up-front. SD-MUXes in the remaining (unused) macro-cells are configured as normal CLBs, similar to how shadow clusters are used [45].

The placer considers all other CLBs to be functionally equivalent. VPR's placer uses simulated annealing. We implemented two placement strategies. In the first, we place shifters onto macro-cells and fix their placement; the placer moves normal soft logic clusters around, but does not perturb the placement of the shifters onto macro-cells. The second option relaxes this constraint, and moves both soft logic clusters around the FPGA and may also move any shifter onto an unused macro-cell. Macro-cells, configured as shifters, are similar to DSP blocks from the perspective of the CAD tools. The difference is that unused logic and routing resources within each macro-cell, after it has been placed-and-routed, remain available to the global placer and router and can be used by the rest of the circuit.

4.4 EXPERIMENTAL RESULTS

Our experimental goals are twofold. Firstly, we wish to quantify reduction in LUT count that can be achieved by synthesizing shifters onto SD-MUX enabled macro-cells. Secondly, we wish to ensure that the inclusion of macro-cells does not adversely affect routability for industrial-scale benchmarks.

4.4.1. FLOATING-POINT OPERATORS

We consider a set of single-precision multi-operand floating-point adders that have already been optimized for area. These operators are similar to those produced by Altera's floating-point datapath compiler [36][37], which removes redundant normalizations. We used designs published by Verma et al. [71], which were slightly smaller than those produced by Altera's compiler. We used the smallest design approach,

which implemented the internal fixed-point multi-operand addition using a tree of 3-input adders.

For a *K*-input adder, we de-normalize *K-1* mantissas using shifters; the mantissa corresponding the largest exponent is not shifted.

Normalization is only applied once, at the output of the operator. For 2, 4, 8, and 16-input adders, Figure 11 reports that the area savings (in terms of Altera's ALMs) obtained by the macro-cell range from 25% to 32%. Assuming that the number of LUTs and CLBs in an FPGA are fixed, this means that 33-40% more operators can be packed into an area of fixed size when macrocells are used to implement shifters.

We did not measure the effect of the macro-cells on critical path delay or pipeline depth of the adders. The throughput of floating point data paths is driven mostly by spatial parallelism; reducing the area of an operator increases the number of operators that can be synthesized on a fixed area device. The area savings reported in Figure 11, thus, translate indirectly into increased throughput for parallel floating-point data paths that use these operators.

4.4.2. EXPERIMENTAL SETUP: VPR

We modeled an FPGA enhanced with macro-cells using VPR 5.0 [46]. We did not use VPR 6.0, which is now part of the Verilog-To-Routing (VTR) flow, for these studies because it did not have timing models in-place at the time this work was performed. As our baseline, we took one of the VPR architecture files from the iFAR repository [33][34]. Table 4-1 lists the baseline parameters for our architecture. CLB inputs and outputs are evenly distributed around all four sides of the CLB. VPR explicitly models a

C Block, but does not model the intra-cluster routing; as it is a full crossbar, only its delay is modeled.

We do not model SD-MUXes explicitly. Our experiments strive to show that macro-cells, which reserve a non-trivial quantity of routing resources in localized areas, do not adversely affect the ability to route large-scale circuits that contain shifters. Macro-cells are organized as vertical columns when they are introduced into the FPGA. The motivation is to mimic the layout of modern FPGAs. For example, logic clusters are generally laid out as columns; so are DSP blocks, block RAMs, etc. Only a small proportion of CLBs in the FPGA contain SD-MUXes.

For each experiment, we placed each benchmark once and routed it three times using different random number seeds. The delay for each benchmark is the average delay of the three runs. This reduces the noise in our delay results as different random number seeds can yield significantly different routing results.

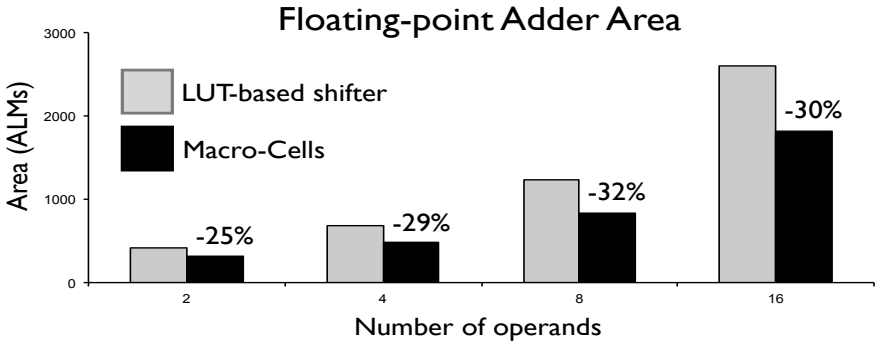


Figure 4-8: Area savings obtained by macro-cells for 24- and 27-bit shifters.

4.4.3. BENCHMARKS

We selected the ten largest IWLS 2005 benchmarks [28], which are described in

Table 4-2, to evaluate the impact of the macro-cell on large-scale applications. Using VPR, we synthetically added macro-cells (shifters) to these benchmarks; our goal is to ascertain whether these shifters, when pre-placed and routed onto macrocells, adversely affect area, delay, and routability.

Parameter	Value	Parameter	Value
LUT Size	**6**	**Fc input**	**0.15**
Cluster size	**8**	**Fc output**	**0.1**
Channel Width	**96**	**Technology***	**65nm CMOS**
Cluster Inputs	**36**	**Tile Area****	**18940**

***Berkeley predictive models** ** ** Min-width transistors**

Table 4-1: FPGA architectural parameters

Benchmark	Description
ac97_ctrl	**Interface to external AC 97 audio codec**
aes_core	**Advanced Encryption Standard (AES)**
des_perf	**16-cycle pipelined DES/3-DES Core**
Ethernet	**10/100 Mbps IEEE 802.3/802.3u MAC**
mem_ctrl	**Embedded memory controller**
pci_bridge32	**Bridge interface to PCI local bus**
Systemcaes	**Area-optimized AES implementation**
usb_func	**USB 2.0 compliant core**
vga_lcd	**Embedded VGA/LCD controller**
wb_conmax	**Wishbone Interconnect Matrix IP Core**

Table 4-2: Ten Largest IWLS Benchmarks

We modified each benchmark's netlist to include 20, 40, 60, 80, and 100 shifters, which are connected at arbitrarily chosen points to ensure that they are not completely

disjoint from the remaining logic. In VPR, we pre-allocated macro-cell columns and preplaced the shifters and preceding layers of logic onto them. We pre-routed the macro-cells, and marked the routing resources used as unavailable. Our primary concern was that locking down these resources up-front would adversely affect the quality of the routes obtained for the remainder of the circuit; fortunately, practically no degradation was observed.

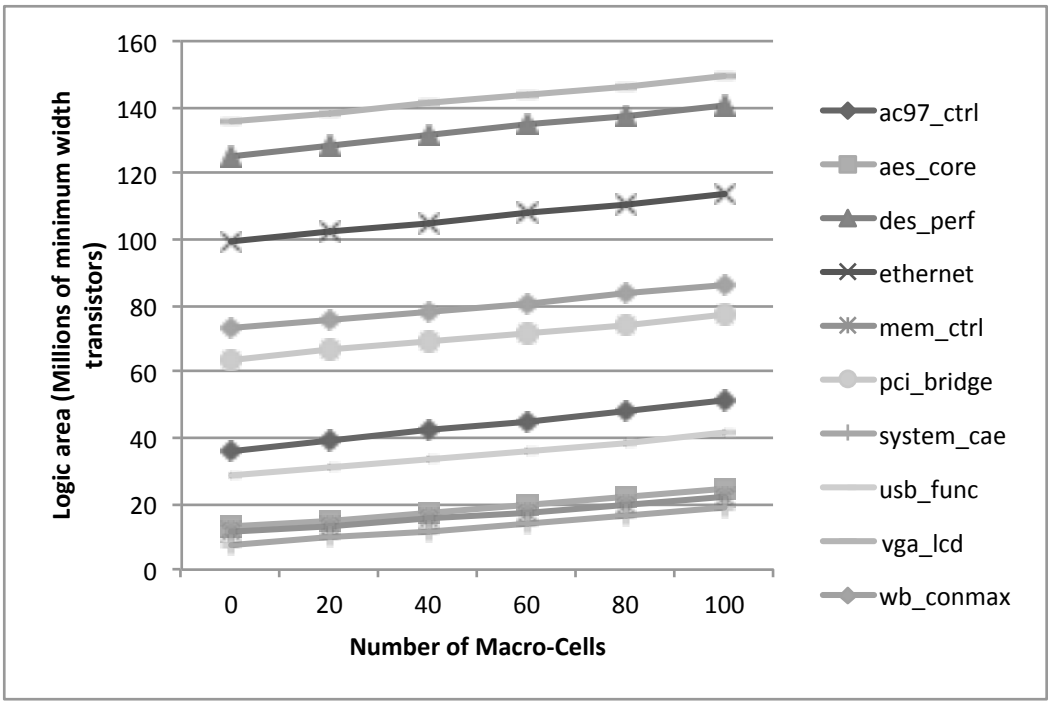


Figure 4-9: Area of the 10 circuits synthesized using VPR 5.0.

VPR generates a custom FPGA for each benchmark, based on its demand for logic and routing resources. Each benchmark is repeatedly placed and routed, varying the channel width each time; VPR converges onto the minimum channel width (*Wmin)* for which a legal route can be found. The average *Wmin* obtained by VPR across all benchmarks (with no macro-cells) here is 84.4. Figure 12 reports the area of each

benchmark with a varying number of shifters. The area is reported in terms of minimum-width transistors; this accounts for the fact that CLBs that have been augmented with SD-MUXes are larger than regular CLBs.

4.4.4. ROUTABILITY

Figure 4-10 reports the critical path delay of the IWLS benchmarks with a varying number of macro-cells; we observe practically no impact on critical path delay from the inclusion of as many as 100 shifters per benchmark. As noted in Section 4.4.2, we considered two different placement strategies: a *constrained* strategy in which the logic placed onto macro-cells is fixed a-priori, and an *unconstrained* strategy in which the placer can move the macrocell logic (the shifter, and logic layer preceding it) onto any macro-cell. The results reported in Figure 4-9 are for the constrained strategy; we observed that the unconstrained strategy produced essentially identical results, where the differences in delays for each data point are in the range of tens of pico-seconds.

Figure 4-10 shows that introducing macro-cells may adversely affect *Wmin*, as each macro-cell requires some routing resources. For many benchmarks, *Wmin* steadily increases when the number of macro-cells ranges from 20 to 80, but decreases rapidly from 80 to 100. The reason for this observation is that VPR automatically generates an FPGA that is sized to a specific application; based on the number of CLBs used and I/O pads required, VPR generates the smallest square FPGA that can provide sufficient resources. VPR then repeatedly places and routes the circuit to determine *Wmin*.

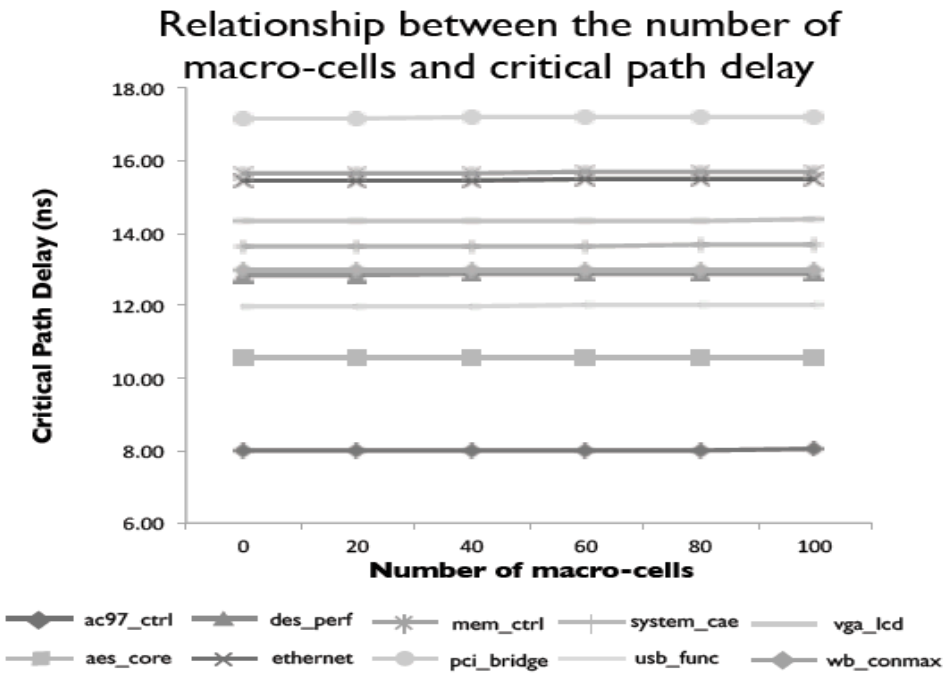


Figure 4-10: Effects of macro-cells on the critical path delay.

Many of the IWLS benchmarks are I/O bound, so CLB utilization is relatively sparse, and there is relatively little congestion in the routing network. Each macro-cell that is added increases CLB utilization, and introduces congestion, which increases *Wmin*. If we assume a fixed-size FPGA, eventually, the inclusion of more macro-cells will cause utilization to exceed 100%. VPR then generates a larger FPGA, with much lower utilization; consequently, the benchmark circuit routes much easier, and *Wmin* is reduced. This is precisely what occurred, for example, for benchmarks aes_core and des_perf (and a few others) between 80 and 100 macro-cells in Figure 4-11. It is important to recall that these benchmarks are synthetic. A floating-point operator, in contrast, would contain shifters and use the available macro-cells.

Moreover, *Wmin* as reported in Figure 4-11 is much smaller than the routing channel width of commercially available FPGAs. These experiments demonstrate that macro-cells are quite useful for benchmarks that contain shifters, while their presence will not adversely affect other benchmarks that do not contain shifters.

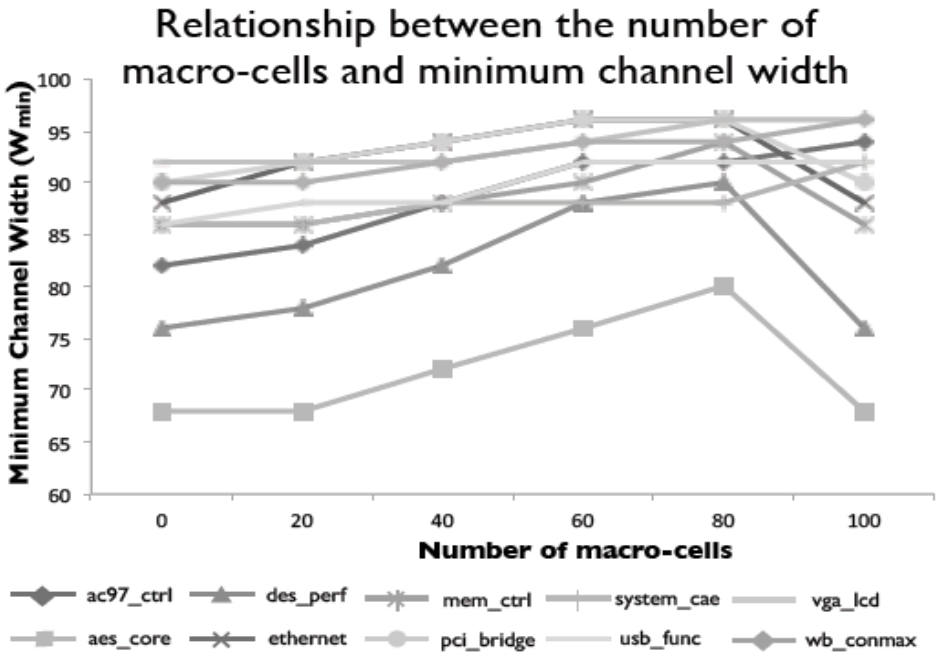


Figure 4-11: Effects of macro-cells on minimum channel width.

4.5 SUMMARY

This chapter has investigated the use of hybrid logic blocks to implement a macro-cell that can implement 27-bit shifters for single-precision floating-point mantissa alignment and normalization. The macro-cells reduce the area of floating-point addition clusters by up to 67%, which increases the number of operators that can be synthesized

into a fixed-area device. This aligns well with the strategy employed by Altera's floating-point datapath compiler [36][37]. We have extensively modified the VPR placement and routing modules to show the feasibility of macro-cells. In particular we investigated two approaches to placing the macro-cells; we first constrained the placement of the logic blocks of a given macro-cell so that they are as close to each other as possible, which in turn minimize the overall delay, we then relaxed this constraint and let the placer decides the location of the logic blocks of the macro-cells. Both approaches yield interesting results; with the constrained approach slightly better in terms of the achievable clock frequency, while the relaxed approach achieved better area savings.

Finally our experiments show that macro-cells do not adversely affect routability for benchmarks that do not contain shifters.

Next chapter investigates parallelizing FPGA routers using a speculation-based approach, on multi-core, shared memory CPU architectures.

Chapter 5. PARALLEL ROUTING FOR FPGAS

This chapter describes an implementation of an FPGA routing algorithm on a shared memory multi-processor using the Galois API, which offers speculative parallelism in software. The router is a parallel implementation of PathFinder, which is the basis for most commercial FPGA routers. We first present our parallel model for the maze router, which parallelize the maze expansion step for each net, while routing nets sequentially to limit the amount of rollback that would likely occur due to mis-speculation. We then present our parallel model for the signal router, which control the congestion negotiation among signals, and evaluate both approaches on large benchmarks.

5.1 PATHFINDER IN GALOIS

This section summarizes the aspects of VPR's routability-driven PathFinder implementation that are relevant to our parallelization scheme [69].

5.1.1. PATHFINDER COMPONENTS

Recall from Chapter 2 that PathFinder is a triple-nested loop [50]: the outer loop is called the global router; the middle loop is called the signal router; and the inner loop is maze expansion. Our parallelization effort focuses on maze expansion.

Global Router: The *global router* set the negotiation criteria, and repeatedly invokes the signal router to route all of the nets. It terminates when a legal routing solution is found, or after a fixed number of iterations fail.

Signal Router: Control the negotiation among signals and at each *signal router* iteration rips up each net and re-routes it by invoking maze expansion.

Maze Expansion: For net N_i, maze expansion computes a path from the source to each sink in the RRG. All of the RRG vertices that have been uncovered are stored in a priority queue (*PQ*) based on their cost. Maze expansion extracts the minimum cost vertex v_{min} from *PQ*. If v_{min} is a sink, then a backtrace procedure is invoked to construct a path from u to $RT(N_i)$, which is the routing tree for N_i; otherwise, each neighbor v of v_{min}, which has not previously been discovered, is inserted into *PQ* and the maze expansion continues.

5.1.2. THE GALOIS FRAMEWORK

We introduced the Galois programming model[51][61] in Chapter 2. Here, we just summarize the key features and issues relevant to parallelizing the router.

Galois implements *speculative parallelism* in a manner that hides the complex underlying details from the programmer. Galois introduces new syntactic constructs that enable the programmer to clearly express algorithms in terms of the operator formulation applied to elements in ordered or unordered sets. Galois provides an API of concurrent data structures, which are challenging to implement; this simplifies application development for the programmer. The Galois runtime automatically detects and rectifies conflicts that cannot be discovered statically. In principle, using Galois is much simpler than requiring the programmer to implement these mechanisms every time he or she parallelizes a new irregular application.

5.1.3. BOTTLENECKS

The major sources of runtime overhead in Galois are as follows:

Dynamic assignment of work: Threads obtain work from a centralized workset. This requires synchronization and leads to challenges in terms of load balancing. If each activity requires minimal computation, then the overhead of synchronization and contention becomes a bottleneck.

Neighborhood constraints: Acquiring and releasing abstract locks on neighborhood elements is a major source of overhead.

Undo log: When an activity modifies an element of a graph (or other data structure), a copy of the element is stored in an "undo" log, which enables rollbacks in the case of misspeculation. The time spent creating and maintaining these copies is non-trivial.

Aborted activities: When an activity aborts (e.g., due to misspeculation), the computational work performed up to that point is wasted; the Galois runtime system takes corrective action to roll back the activity.

To reduce overhead, Galois provides three optimization techniques that we leverage in our PathFinder implementation.

5.1.4. OPTIMIZATIONS USING GALOIS

Cautious operators: A cautious operator reads all elements of its neighborhood before modifying any of them; the reading phase acquires all of the locks. If lock acquisition is successful, then the operator is guaranteed to compete without conflicting with other transactions. In this case, the undo list can be safely discarded. Additionally,

Galois' internal conflict management for the cautious operator can be suppressed since it only accesses elements for which it has already obtained a lock.

One-shot operator implementations: It is often possible to predict the neighborhood of an activity without performing any computation, or to compute fairly tight over-approximations. In a one-shot implementation, the neighborhood elements are never read, so the locks can be released once successful completion is guaranteed; in contrast, a cautious operator must hold onto the locks while its activity commences, even after guaranteeing completion. Releasing locks early enables greater concurrency.

Iteration Coalescing: Iteration coalescing allows one thread to process multiple iterations at once, breaking the one-to-one correspondence between iterations and activities. Galois provides each thread with a local workset. Any activity that generates new active elements places them in the thread's local workset, as opposed to the global workset, which is accessed by all threads. When an activity completes, the iteration grabs work from its local workset if possible without releasing any abstract locks. This continues until the local workset is empty, a conflict is detected, or the maximum coalescing factor is reached. Each iteration releases all of its abstract locks when it finishes. If a conflict occurs, the currently executing activity aborts, but all prior completed activities that were coalesced into the same iteration commit; the activities in the local workset are then moved to the global workset, where other threads may execute them.

5.2 PARALLEL PATHFINDER IN GALOIS

In this section we describe our implementation of PathFinder in the Galois framework. We describe both parallel approaches: Maze expansion and signal router.

5.2.1. PARALLEL MAZE ROUTER

Figure 5-1 illustrates our parallel implementation of maze expansion in Galois. Prior work has shown that maze expansion accounts for approximately 68% of the total runtime [5]. Our expectation was that Galois could identify a large number of non-conflicting operations, enabling parallel execution of a large number of active vertices. Fig. 3 illustrates *iteration coalescing*: each thread has its own local priority queue (LPQ) while a global priority queue (GPQ) is stored in shared memory, along with the RRG, a set of routing trees for each net, and an array, *VCost*, which contains information relating to the cost *c(v)* of each RRG vertex; *c(v)* depends on several other cost terms that contribute to *PathCost(v)* per Eq.(1) [50][69]. Each *VCost* entry is a struct that holds these cost values, which are stored separately from the RRG in shared memory to reduce contention for locks. Each thread repeatedly accesses its LPQ to obtain another vertex to expand. If the LPQ is empty, then the access is forwarded to the GPQ. The maze expansion process stops when all sinks are found.

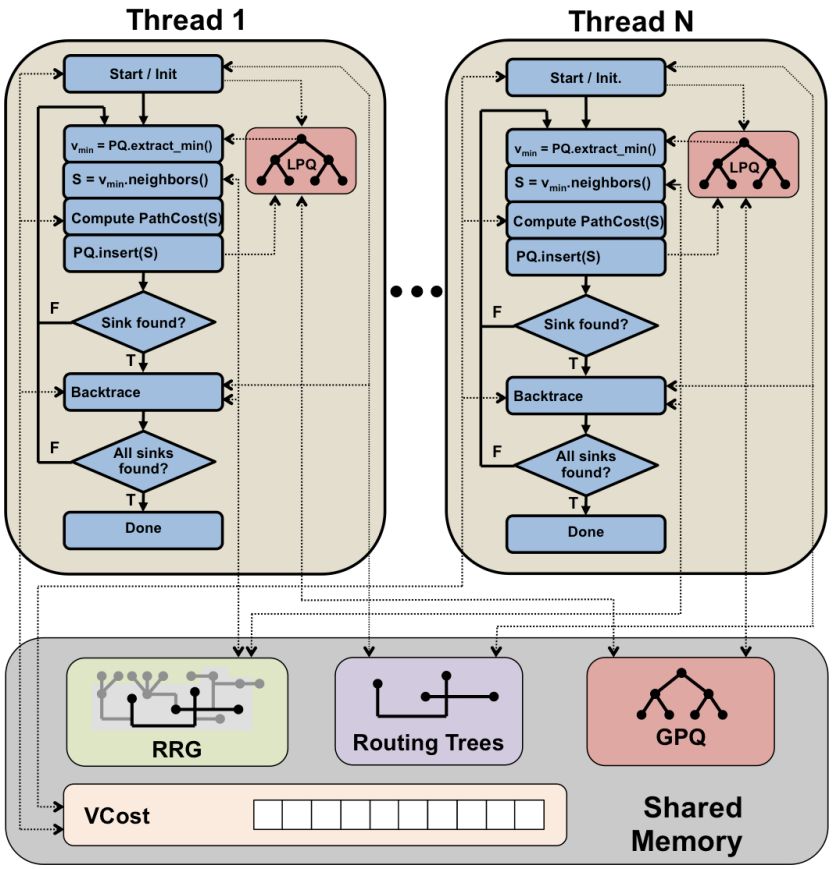


Figure 5-1: Multi-threaded parallelization strategy.

Figure 5-1 depicts our multi-threaded parallel strategy for the Galois implementation of PathFinder. Solid lines indicate control flow within each thread; dashed lines indicate the data structures accessed by each operation.

5.2.2. MAZE EXPANSION OPERATORS

The neighborhood expansion operator is *cautious*. Each thread picks an active vertex v_{min} from its LPQ. The operator identifies a set S of vertices adjacent to v_{min} that have not yet been discovered. Galois acquires locks for each vertex vS. The operator inserts v into LPQ with *PathCost(v)* as its priority. The *Backtrace* operator, which is called if v is a

sink, is *one-shot*, creates a path from *v* to the routing tree for the current net, and sets *PathCost(u)* to zero for each vertex *u* on the path. If *any* neighbor of *u* is locked, then the corresponding activity *may* access *u*. If so, then it is better to keep *u* locked while updating *PathCost(u)*. If none of *u*'s neighbors are locked, then it is safe to release the lock and update *PathCost(u)*.

5.2.2.1. PRIORITY QUEUE IMPLEMENTATION

Although Galois provides a concurrent PQ, we discovered that its functionality was limited and its performance was a bottleneck. Galois's PQ is integer-based and can only store the ID number of a vertex; any additional information needs to be mapped to another data structure. Additionally, each thread that tries to insert or remove a vertex from the priority queue must acquire a lock; with a large number of concurrently executing threads, the lock acquisition process becomes a performance bottleneck.

To improve performance, we implemented a non-blocking priority queue based on software transactional memory (STM) [19][62] which deviates from the Galois model. The STM-based PQ declares the access functions as atomic. The underlying implementation is a binary heap, similar to VPR's non-concurrent PQ. Insertion and extract operations are executed as transactions.

5.2.3. PARALLEL SIGNAL ROUTER

Parallelizing the signal router involves partitioning the signals (netlist) into sets, with each set routed by a separate instance of the signal router running as a separate thread on a separate processor. Each signal router instance maintains a local priority queue while the RRG and the associated congestion maps are shared among all threads. The Galois runtime maps each thread to a different processor. Threads update intermediate routing

results through lock-based data structures in shared memory, and synchronize their respective views of the overall routing state. Threads are synchronized at the end of each iteration of PathFinder to ensure that all instances are working simultaneously on the same iteration. The pseudo-code for our parallel PathFinder implementation is given in **Figure 5-2**

```
While shared resources exists
    partition signals into N sets ;
    ripup any old routing;
    assign nets to threads
    each_thread
        for (net = 1 .. N)
            PathFinder_route (net);
            update congestion costs;
            update routing trees;
        end for
    end each_thread
    update congestion costs;
    check if routing is legal (No shared resources exist);
end while
```

Figure 5-2: Pseudo-code for the the Signal router

The signal router can suffer from high misspeculation and rollback costs as it routes multiple signals in parallel. Likewise, tearing down partially routed paths can cause high overhead. The next section presents the experimental results from both the signal and maze expansion routers.

5.3 EXPERIMENTAL SETUP

5.3.1. VPR IMPLEMENTATION IN GALOIS

We ported VPR 5.0 into the Galois system. We modified all of the data structures used by the router to be thread-safe and compatible with Galois. We rewrote the router to be compliant with the operator formalism [61], which is a central requirement of Galois.

We implemented the RRG using Galois' graph model, and the STM-based non-blocking PQ as discussed in Section 4.2.

5.3.2. VPR ARCHITECTURAL PARAMETERS

VPR generates an FPGA architecture from a set of parameters [7], whose dimensions are approximately equal to the per-benchmark resource requirements. **Table 5-1** lists the parameters we used.

K	N	W	I	F_{cin}	F_{cout}	CLB Area
6	10	$1.4W_{min}$	33	0.15	0.1	8069.46

Table 5-1: FPGA architectural parameters, taken from the publicly available iFAR repository[33][34]; we assume 65nm CMOS (BPTM).

5.3.3. BENCHMARKS

We selected 10 of the largest IWLS benchmarks [28] for use in our experiments. **Table 4-2** summarizes them, including the size of the FPGA generated by VPR, the number of nets, and the number of configurable logic blocks (CLBs) used.

VPR repeatedly routes each benchmark using a binary search to identify the smallest channel width, W_{min}, for which a legal route can be found. VPR also allows the user to specify a chosen channel width (W), and VPR will try its best to find a legal route, but may fail. We took that latter approach in our experiments: first, we compute W_{min} (using VPR's serial implementation of PathFinder) and set $W = 1.4W_{min}$ for each benchmark.

Benchmark	Dimensions	Nets	CLBs
ac_ctrl	48 x 48	5097	5008
aes_core	33 x 33	5800	2518
des_area	16 x 16	1569	695
mem_ctrl	27 x 27	4464	3158
pci_bridge32	74 x 74	8016	7815
spi	13 x 13	923	712
systemcaes	21 x 21	2509	2173
systemcdes	12 x 12	1068	706
usb_funct	40 x 40	5154	4429
wb_conmax	47 x 47	10430	6297

Table 5-2: Benchmark summary

5.3.4. SYNTHESIS FLOW

The IWLS benchmarks are provided in .blif format. To target VPR, we used ABC [6] for logic synthesis and technology mapping, T-VPack[3] for placement, and our Galois-compatible VPR implementation for placement and routing; we did not parallelize VPR's placer, which is based on simulated annealing.

VPR's routability-driven router [69] is based on PathFinder [50], which terminates if it cannot find a legal route after a user-specified number of iterations. We set the maximum number of iterations allowed to 50. If routing is successful, VPR reports the estimated critical path delay of the circuit.

[3] T-VPack has been deprecated as part of the Verilog-to-Routing (VTR) flow, which expands VPR 6.0 [47]; packing is now integrated into VPR 6.0. The router has not changed significantly from VPR 5.0 to 6.0.

5.3.5. EXPERIMENTAL PLATFORM

Our primary experimental objective is to assess the performance and scalability of our parallel implementation of PathFinder on a modern multi-processor. Our experiments were performed on a server featuring 8 Intel Xeon E5540 processors running at 2.53 GHz, with 4 cores per processor and 40 GB shared memory. We ran our router using 1, 2, 4, and 8 threads. Our baseline is the single-threaded VPR 5.0 router, which was implemented in C and does not incur any overhead due to the Galois runtime.

Galois, as presently implemented, imposes several constraints. It supports a maximum of 8 concurrent threads; ideally, we would have liked to run at least 32 threads on our system. Additionally, the Galois runtime allocates threads to cores automatically; in all cases, it allocated one thread per processor, leaving three cores unused. We were unable to override this decision to experiment with alternative thread allocation policies. In particular, with one thread per processor, we cannot explore the implications of shared cache hierarchies on the router's performance.

5.4 EXPERIMENTAL RESULTS

This section illustrates the results for both the signal router and the maze expansion router using both deterministic and non-deterministic Galois schedulers.

5.4.1. ROUTABILITY

Our first concern is routability, i.e., what percentage of nets routed successfully in each experiment? Anything less than 100% would indicate a routing failure. Both routers successfully routed all of the nets for all benchmarks in all of our experiments.

5.4.2. SPEEDUP

Figure 5-3 reports the speedup for the Maze Router attained by increasing the number of threads, normalized to VPR 5.0's single-threaded execution. For non-deterministic scheduling, Galois achieved an average speedup of 1.47x with two threads, 2.99x with four threads, and 5.46x with eight threads.

Maze Router - Normalized Speedup as a function of the number of threads

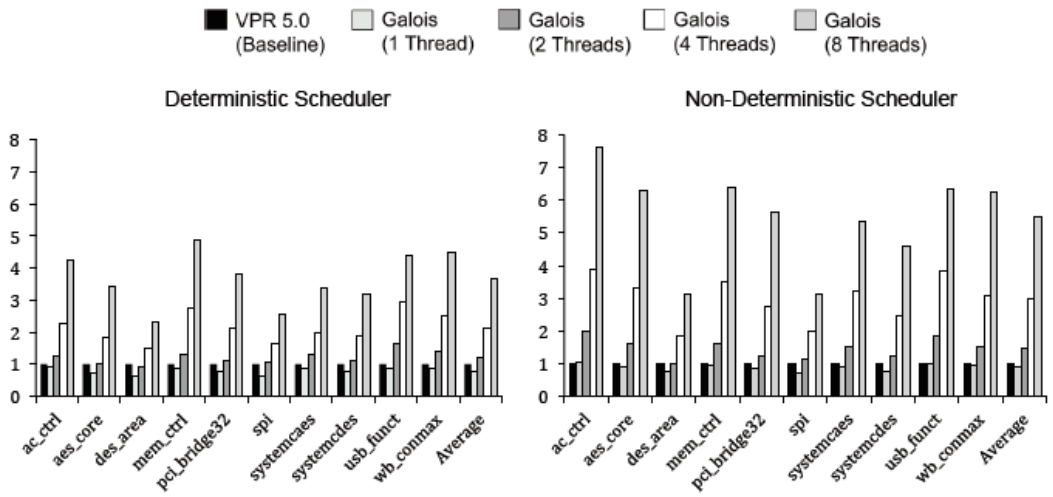


Figure 5-3: Speedup (normalized to VPR 5.0) of our Galois implementation of the Maze Route with 1, 2, 4, and 8 threads.

The deterministic scheduling achieved lower speeups of 1.21x for 2 threads, 2.14x for 4 threads, and 3.67x for 8 threads. This is expected due to the high overhead associated with the deterministic scheduler as the interference graph in general is rebuilt from scratch each round. In general, these results indicate favorable scalability up to at least eight threads.

Perfectly linear speedups are not expected, due to the overhead of lock acquisition, aborted activities, and the cost of accessing data structures in shared memory. That being said, these experiments show that maze expansion exhibits ample amorphous parallelism and that a runtime system like Galois can readily extract it.

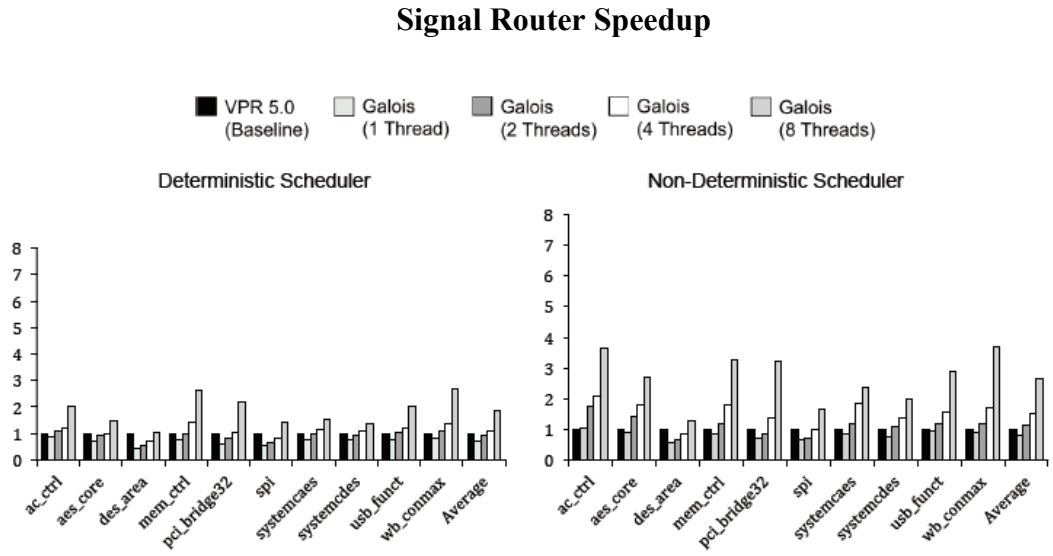


Figure 5-4: Speedup (normalized to VPR 5.0) of our Galois implementation of the Signal Router with 1, 2, 4, and 8 threads.

Figure 5-4 reports the speedup for the Signal Router by increasing the number of threads, normalized to VPR 5.0's single-threaded execution. For non-deterministic scheduling, the Signal Router achieved an average speedup of 1.13x with two threads,

1.54x for 4 Threads, and 2.67 for 8 threads. The deterministic Signal Router on the other hand achieved an average speedup of 0.91x for 2 threads, 1.11x for 4 threads, and 1.84x for 8 threads. This clearly indicates that the Signal router incurs a huge amount of overhead due to high misspeculation and rollback costs. As well, we observed the same differences between the deterministic and non-deterministic schedulers.

5.4.3. CRITICAL PATH DELAY VARIATION

We have used both deterministic and non-deterministic schedulers currently implemented in Galois to perform our experiments. We were concerned that varying the number of threads could cause alter the routing results, which could impact the critical path delay.

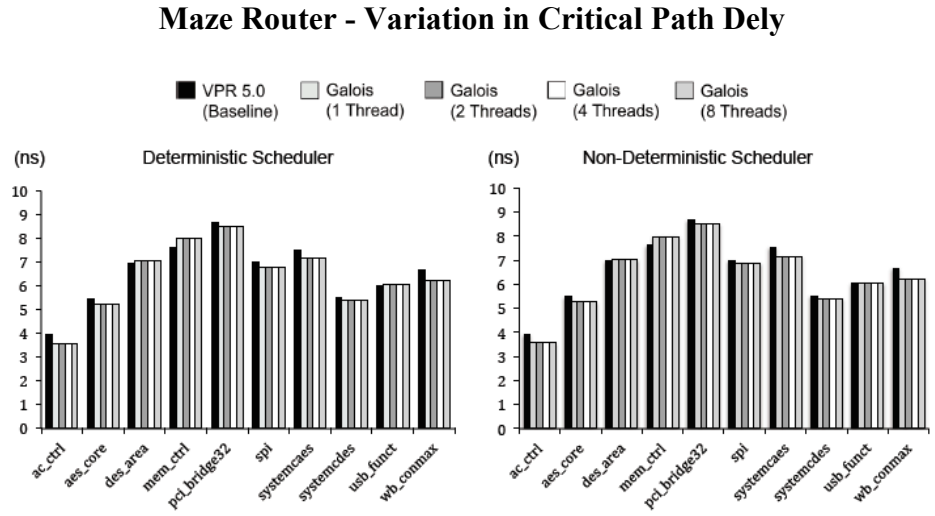


Figure 5-5: Effect of the number of threads on critical path delay for the Maze Router.

Figure 5-5 and **Figure 5-6** report the critical path delays obtained by the maze router and the Signal Router respectively for each benchmark routed by VPR 5.0 and Galois with 1, 2, 4, and 8 threads. A significant difference between the VPR 5.0 results

and Galois is observed for several benchmarks; however, differences between Galois using different numbers of threads, per-benchmark, are negligible. This does not imply that Galois obtained identical routing solutions for all benchmarks; it only suggests that all benchmarks achieved similar critical path delays.

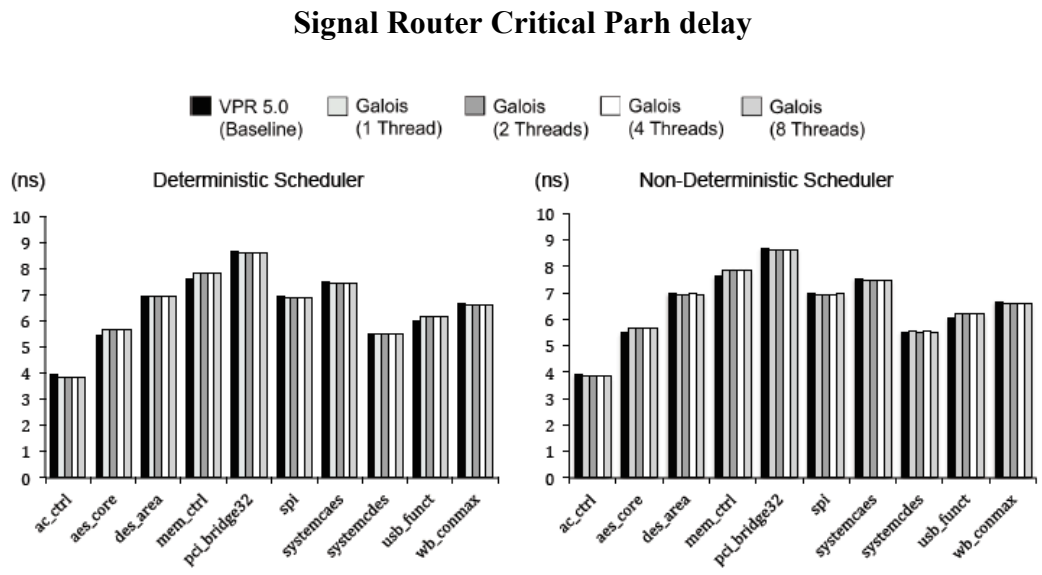


Figure 5-6: Effect of the number of threads on critical path delay for the Signal Router.

5.4.4. IMPLEMENTATION CHOICES

The two most important implementation decisions were iteration coalescing and replacing the Galois PQs with STM-based PQs, as described in Section 5.2.2.1. Using 8 threads, we ran our parallel implementation of PathFinder with four configurations: (1) no iteration coalescing with Galois' PQ; (2) no iteration coalescing and STM PQ; (3) iteration coalescing with Galois' PQ; and (4) iteration coalescing with the STM PQ. When iteration coalescing is enabled, we use the same PQ implementation (Galois or

STM) for both the GPQ and each thread's LPQ. Figure 5-7 reports the speedup of each implementation decision, normalized to Galois running one thread.

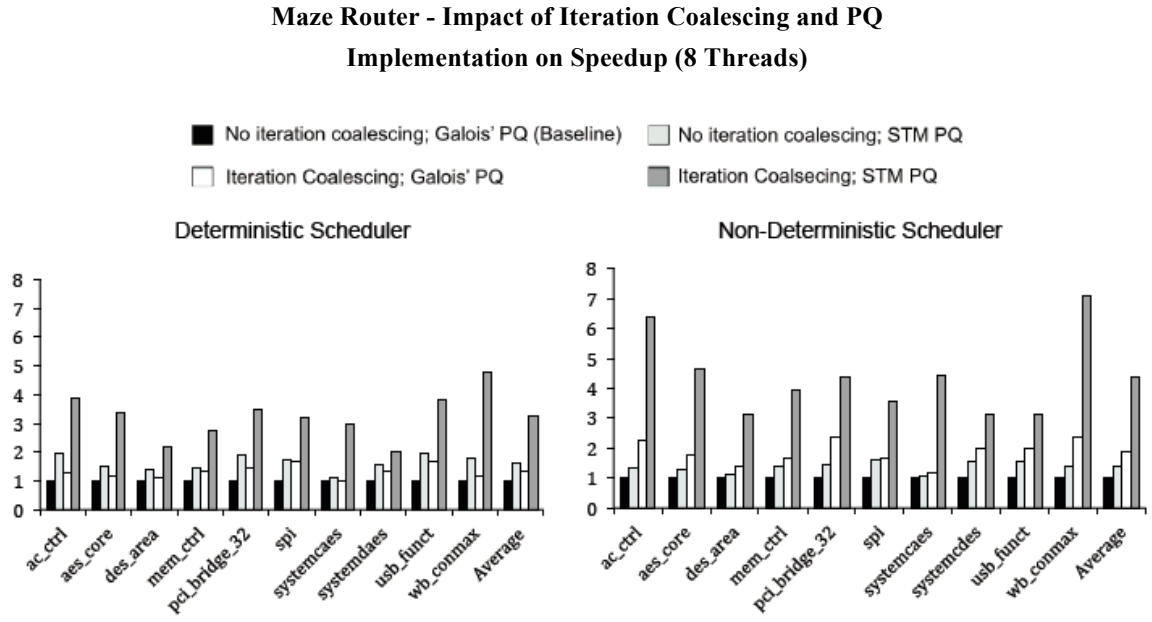


Figure 5-7: Normalized speedups obtained by introducing iteration coalescing and STM-based PQs into our Galois implementation of the Maze Router.

Figure 5-7 clearly shows that performing both optimizations (iteration coalescing using an STM-based PQ) offers significantly greater speedup compared to enabling one optimization, but not the other (4.38x, on average, compared to 1.38x for using STM-based PQs without iteration coalescing, and 1.87x for enabling iteration coalescing while using Galois' PQs). These results indicate that we made the correct implementation decisions; one possibility for future work is to consider concurrent PQs other than STM [62].

5.5 SUMMARY

This chapter has demonstrated that speculative parallelization and the operator formalism, central to Galois' programming model and philosophy, is an effective choice to implement an FPGA router; moreover, we suspect that it will be equally effective for other irregular CAD algorithms that operate on graph-based data structures. These algorithms exhibit significant parallelism, but require a runtime system like Galois to detect and exploit it. The speedups obtained depend on implementation choices, as we have shown that non-blocking priority queues and iteration coalescing aided us significantly. Our work has shown that this general approach scales well and that the amount of parallelism exploited is a function of the circuit size not the inter-dependence of the signals. Lastly, it is important to note that Galois permits the user to trade raw performance for deterministic results, if desired.

Chapter 6. CONCLUSIONS

6.1 CONTRIBUTION

This thesis addressed three major issues related to logic block architecture, the routing fabric, and CAD for FPGAs. On these issues this work first presented two modifications to the PathFinder negotiated congestion router to handle FPGAs that employ sparse intra-cluster routing crossbars, PPR and SERRGE, both of which reduce PathFinder's runtime and memory footprint. Between the two, SERRGE offers a small improvement in critical path delay compared to our Baseline router; in contrast, PPR offers comparable results, while being much simpler to implement than SERRGE, running much faster, and offering a modest reduction in memory footprint. SERRGE is particularly complex because it is essentially an application-specific dynamic memory management and garbage collection framework that has been specialized to the the PathFinder algorithm running on the RRGs generated by VPR.Although we did not discuss details, significant modifications were made to several of VPR 5.0's internal data structures in order to accommodate SERRGE's requirement. To summarize: if router runtime is a premium, then PPR should be used; if critical path delay is more important, then SERRGE is preferable; in the vast majority of our experiments, PPR and/or SERRGE outperformed the Baseline router for all metrics of interest.

The second major contribution of this work is the introduction of SD-MUXes, which enable CLBs to be configured as 27-bit shifters, which can reduce the cost of

implementing mantissa alignment and normalization for single-precision floating-point operators on FPGAs. When configured dynamically, a signal that routes to an SD-MUX must target one specific pin as a sink; this is a far more constrained routing problem than for traditional FPGAs, where all LUT inputs are treated as being logically equivalent. To ensure reasonable place-and-route solutions, we introduced the notion of pre-placed-and-routed macro-cells, which establish the feasibility of CAD support for CLBs enhanced with SD-MUXes. This leads to a 67% reduction in the number of floating-point addition clusters that can be synthesized onto a fixed-area device. This architectural innovation aligns well with the strategy employed by Altera's floating-point datapath compiler [36][37].

The final contribution of the thesis is the introduction of a speculative approach for parallelizing the FPGA router, using the Galois framework; our experiments using this approach demonstrated a near linear-time speedup (when deterministic results were not required), and showed that the amount of parallelism in an irregular routing CAD tool depends on the size of the circuit not the inter-dependence of the individual signals, and that the quality of results did not suffer as a result of the parallel implementation.

6.2 FUTURE WORK

We believe that the problem of routing for FPGAs with sparse intra-cluster routing crossbars has effectively been solved; we do believe that future work on this topic is warranted. That being said, the combined integration of FPGA logic clusters with sparse-

intra-cluster routing crossbars and SD-MUXes remains an open problem, and may entail additional work on FPGA routing algorithms for support.

We see envision several opportunities to advance the state-of-the-art in parallel routing for FPGAs using Galois. We believe that better results could be obtained for our by replacing the concurrent priority queue based on software transactional memory [62] with a non-blocking one based on skipped lists [68]. We expect to obtain further improvements by parallelizing an A* search, rather than a priority-driven breadth-first search, the latter of which characterizes our current implementation. Similarly, there has not yet been any attempt to parallelize VPR's timing-driven router using Galois; the timing-driven router employs a quadratic, rather than linear, delay model based on Elmore delay; computing these delays entails significant performance overhead, and the cost of storing the Elmore delay trees once they have been computed is also high. We also anticipate that future enhancements to Galois' internals may lead to higher performance, for both deterministic and non-deterministic parallel routers. We also hop

Lastly, we believe that Galois' approach to speculative parallelization readily extends to other problems in CAD, for both standard cell VLSI technologies and FPGA. Limiting the discussion here to FPGAs, there are clearly additional opportunities to parallelize other relevant CAD algorithms including placement, packing, technology mapping, and various FPGA-oriented logic optimizations and decompositions; future work can and should look into these issues.

REFERENCES

[1] Ahmed, E., and Rose, J. The effect of LUT and cluster size on deep submicron FPGA performance and density. *IEEE Trans. VLSI*, vol.12, no. 3, March, 2003, pp. 288-298. DOI=http://dx.doi.org/10.1109/TVLSI.2004.824300

[2] Altera, Website: http://www.altera.com/

[3] Altera Arria 10 Core Fabric and General Purpose I/O Handbook, http://www.altera.com/literature/hb/arria-10/a10_handbook.pdf

[4] Altera. Stratix 5 Device Handbook http://www.altera.com/literature/hb/stratix-v/stx5_core.pdf

[5] Beauchamp, M. J., Hauck, S., Underwood, K. D., and Hemmert, K. S. Architectural modifications to enhance the floating-point performance of FPGAs. *IEEE Trans. VLSI*, vol. 16, no. 2, Feb. 2008, pp. 177-187. DOI=http://dx.doi.org/10.1109/TVLSI.2007.912041

[6] Berkeley Logic Synthesis and Verification Group. "ABC: A system for sequential synthesis and verification.: December 2005 release. URL= http://www.eecs.berkeley.edu/~alanmi/abc

[7] Betz, V., Marquardt, S., and Rose, J., *Architecture and CAD for Deep Submicron FPGAs*. Norwell, MA, USA: Kluwer Academic Publishers (now Springer), 1999.

[8] Betz, V., and Rose, J., "Automatic generation of FPGA routing architectures from high-level descriptions," ACM/SIGDA Int. Symp. FPGAs (FPGA '00), pp. 175-184, Feb. 10-11, 2000, DOI= http://doi.acm.org/10.1145/329166.329203

[9] Brown, S., Rose, J., and Vranesic, Z. G., "A detailed router for field programmable gate arrays," *IEEE Trans. on Computer-Aided Design*, vol. 11, no. 5, pp. 620-628, May 1992. DOI=http://dx.doi.org/10.1109/43.127623

[10] Burtscher, M., Nasre, R., and Pingali, K., " A quantitative study of irregular programs on GPUs," IEEE International Symposium on Workload Characterization (IISWC), pages 141-151, 2012. DOI=http://dx.doi.org/10.1109/IISWC.2012.6402918

[11] Carter, W. S., Duong, K., Freeman, R. H., Hsieh, H.-C., Ja, J. Y., Mahoney, J. E., Ngo, L. T., and Sze, S. L., "A user programmable reconfigurable logic array," Proceedings of the Custom Integrated Circuits Conference, pp. 233-235, 1986.

[12] Chan, P. K., and Schlag, M. D. F. "Acceleration of an FPGA router," IEEE Symp. Field Programmable Custom Computing Machines (FFCM '97), pp. 175-181, Apr. 16-18, 1997, DOI= http://dx.doi.org/10.1109/FPGA.1997.624617

[13] Chang, Y. W., Zhu, K., and Wong, D. F., "Timing-driven routing for symmetrical array-based FPGAs," *ACM Trans. Design Automation of Electronic Systems*, vol. 5, no. 3, pp. 433-450, Jul. 2000, DOI=http:/doi.acm.org/0.1145/348019.348101

[14] Chen, D., Cong, J., and Pan, P., FPGA Design Automation: A Survey, Foundations and Trends® in Electronic Design Automation, Vol. 1, No. 3, pp 139-169, 2006.

[15] Chin, S. Y. L., and Wilton, S. J. E., "Static and dynamic memory footprint reduction for fpga routing algorithms," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 4, pp.1–20, Jan. 2009. DOI=http://doi.acm.org/10.1145/1462586.1462587

[16] Chin S. Y. L., and Wilton, S. J. E. , "Towards scalable FPGA CAD through architecture," in Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2011, pp. 143-152.

[17] Chong, Y. and Parameswaran, S., "Flexible multi-mode embedded floating-point unit for field programmable gate arrays," ACM/SIGDA Int. Symp. FPGAs (FPGA '09), pp. 171-180, Feb. 22-24, 2009, DOI=http://doi.acm.org/10.1145/1508128.1508155

[18] de Dinechin, F., Klein, C., and Pasca, B., "Generating high performance custom floating-point pipelines," Int. Conf. Field Programmable Logic and Applications (FPL '09), Aug. 31- Sept. 2, 2009. DOI=http://dx.doi.org/10.1109/FPL.2009.527255/

[19] Dragicevic, K., and Bauer, D. "Survey of concurrent priority queue algorithms", IEEE Int. Symp. Parallel and Distributed Processing (IPDPS '08), Apr. 14-18 2008, DOI=http://dx.doi.org/10.1109/IPDPS.2008.4536331

[20] Feng, W. and Kaptanoglu, S. Designing Efficient Input Interconnect Blocks for LUT Clusters Using Counting and Entropy. *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 1, Mar. 2008, pp. 1-28. DOI= http://doi.acm.org/10.1145/1331897.1331902

[21] Gigliotti, P., "Implementing barrel shifters using multipliers," XAPP– Application Note: Virtex II Family, pp. 1-4, Aug., 2004. URL=http://www.xilinx.com/support/documentation/application_notes/xap p195.pdf

[22] Gort M., and Anderson, J. H., "Accelerating FPGA routing through parallelization and engineering enhancements," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 31, no. 1, pp. 61-74, Jan. 2012.

[23] Gort, M., and Anderson, J. H., "Combined architecture/algorithm approach to fast FPGA routing," IEEE Transactions on Very Large Scale Integration Systems (TVLSI), vol. 21, no. 6, pp. 1067-1079, June, 2013.

[24] Gort, M., and Anderson, J. H., "Deterministic multi-core parallel routing for FPGAs," Int. Conf. Field Programmable Technology (ICFPT '10), pp. 61-69, Dec. 8-10, 2010, DOI=http://dx.doi.org/10.1109/FPT.2010.5681758

[25] Greene, J. W., Kaptanoglu, S., Feng, W., Hecht, V., Landry, J., Li, F., Krouglyanskiy, A., Morosan, M., and Pavzner, V., "A 65nm flash-based FPGA fabric optimized for low cost and power," in Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2011, pp. 87-96.

[26] Ho, C. H., et al., Floating-point FPGA: architecture and modeling. *IEEE Trans. VLSI*, vol. 17, no. 12, Dec. 2009, pp. 1709-1718. DOI=http://dx.doi.org/10.1109/TVLSI.2008.2006616

[27] Intel Corporation. IntelR XeonR Processor E5-2600 Product Family Uncore Performance Monitoring Guide, 2012.

[28] IWLS 2005 Benchmarks. URL=http://iwls.org/iwls2005/benchmarks.html

[29] Jamieson, P., and Rose, J., "Enhancing the area-efficiency of FPGAs with hard circuits using shadow clusters," *IEEE Trans. CAD*, vol. 18, no. 12, Dec. 2010, pp. 1696-1709. DOI = http://dx.doi.org/10.1109/TVLSI.2009.2026651

[30] Jamieson, P., and Rose, J., "Mapping multiplexers onto hard multipliers in FPGAs," 3rd Int. IEEE Northeast Workshop on Circuits & Systems (IEEE-NEWCAS '05), pp. 323-326, June 19-22, 2005. DOI=http://dx.doi.org/10.1109/NEWCAS.2005.1496692

[31] Karp, R. M., "Reducibility among combinatorial problems," in Proceedings of the Symposium on the Complexity of Computer Computations, Yorktown Heights, New York, March, 1972, pp. 85-103.

[32] Kaviani, A., FPGA with improved structure for implementing large multiplexors. U.S. patent, no. US 6,556,042 B1, Apr. 29, 2003.

[33] Kuon, I., and J. Rose, "Area and delay trade-offs in the circuit and architecture design of FPGAs," ACM/SIGDA Int. Symp. FPGAs (FPGA '08), pp. 149-158, Feb. 24-26, 2008, DOI= http://doi.acm.org/10.1145/1344671.1344695

[34] Kuon, I., and J. Rose, "Automated transistor sizing for FPGA architecture exploration," ACM/IEEE Design Automation Conference (DAC '08), pp. 792-795, June 8-13, 2008, DOI= http://doi.acm.org/10.1145/1391469.1391671

[35] Kuon I., Tessier, R., and Rose, J., FPGA Architecture: Survey and challenges, in "Foundation and trends in electronic design automation" Volume 2, issue 2 April 2008.

[36] Langhammer, M., "Floating point datapath synthesis for FPGAs," Int. Conf. Field Programmable Logic and Applications, (FPL '08), pp.355-360, Sept. 8-10, 2008. DOI=http://dx.doi.org/10.1109/FPL.2008.4629963

[37] Langhammer, M., and Vancourt, T., "FPGA floating point datapath compiler," IEEE Symp. 17th IEEE Symp. Field-programmable Custom Computing Machines (FCCM '09), April 5-7, 2009. DOI = http://dx.doi.org/10.1109/FCCM.2009.54

[38] Lee, C. Y., "An algorithm for path connections and its applications," IRE Transactions on Electronic Computers, vol. EC-10, no. 2, pp. 346-365, February, 1961.

[39] Lee, Y. S., and Wu, C. H., "A performance and routability driven router for FPGAs considering path delay." In *Proc. ACM/IEEE Design Automation Conference*, June 1995, pp.557–561. DOI=http://dx.doi.org/10.1109/DAC.1995.250009

[40] Lemieux, G. and Brown, S. D., A detailed routing algorithm for allocating wire segments in FPGAs. *ACM/SIGDA Physical Design Workshop*, 1993.

[41] Lemieux, G. Lee, E. Tom, M., and Yu, A. "Directional and single driver wires in FPGA interconnect," IEEE International Conference on Field-Programmable Technology (FPT '04), pp. 41-48, Dec. 6-8, 2004, DOI=http://dx.doi.org/10.1109/FPT.2004.1393249

[42] Lemieux, G., Leventis, P., and Lewis, D., "Generating highly routable sparse crossbars for PLDs," In *Proc. ACM/SIGDA Int. Symp. FPGAs*, Feb. 2000, pp. 155-164. DOI=http://doi.acm.org/10.1145/329166.329199

[43] Lemieux, G, and Lewis, D. "Using sparse crossbars within LUT clusters," ACM/SIGDA Int. Symp. FPGAs (FPGA '01), pp. 59-68, Feb. 11-13, 2001, DOI= http://doi.acm.org/10.1145/360276.360299

[44] Lewis, D., Betz, V., Jefferson, D., Lee, A., Lane, C., Leventis, P., Marquardt, S., McClintock, C., Pedersen, B., Powell, G., Reddy, S., Wysocki, C., Cliff, R., and Rose, J. "The Stratix™ routing and logic architecture," in Proceedings of the 11th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2003, pp. 12-20.

[45] Luu, J., Anderson, J., and Jonathan Rose, "Architecture Description and Packing for Logic Blocks with Hierarchy, Modes and Complex Interconnect," ACM/SIGDA Int.

Symp. FPGAs (FPGA '11), pp. 227-236, Feb. 27- Mar. 1st, 2011, DOI=http://doi.acm.org/10.1145/1950413.1950457

[46] Luu, J., Kuon, I., Jamieson, P., Campbell, T., Ye, A., Fang, W. M., and Rose, J. "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling," ACM/SIGDA Int. Symp. FPGAs (FPGA '09), pp. 133-142, Feb. 22-24, 2009, DOI= http://doi.acm.org/10.1145/1508128.1508150

[47] Luu, J., Rose, J., and Anderson, J. H., "Towards interconnect-adaptive packing for FPGAs," in Proceedings of the 22nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2014, pp. 21-30.

[48] Marquardt, A., Betz, V., and Rose, J. "Timing-driven placement for FPGAs," ACM/SIGDA Int. Symp. FPGAs (FPGA '00), pp. 203- 213, Feb. 10-11, 2000, DOI=http://doi.acm.org/10.1145/329166.329208

[49] Marquardt, A., Betz, V., and Rose, J. "Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density," ACM/SIGDA Int. Symp. FPGAs (FPGA '99), pp. 37-46, Feb. 21-23, 1999, DOI= http://doi.acm.org/10.1145/296399.296426

[50] McMurchie, L., and Ebeling, C. "PathFinder: a negotiation-based performance-driven router for FPGAs," ACM/SIGDA Int. Symp. FPGAs (FPGA '95), pp. 111-117, Feb. 12-14, 1995, DOI=http://doi.acm.org/10.1145/201310.201328

[51] Méndez-Lojo, M., et al., "Structure-driven optimizations for amorphous data-parallel programs," ACM/SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '10), pp. 3-14, Jan. 9-14, 2010, DOI=http://dx.doi.org/10.1145/1693453.1693457

[52] Metzgen, P., and Nancekievill, D. Multiplexer restructuring for FPGA implementation cost reduction. Design Automation Conf. (DAC '05) pp. 421-426, June 13-17, 2005, DOI=http://doi.acm.org/10.1145/1065579.1065692

[53] Moctar, Y. O. M., Lemieux, G. G. F., and Brisk, P. "Routing algorithms for FPGAs with sparse intra-cluster routing crossbars," in Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, Aug. 2012, pp. 91-98.

[54] Mulpuri, C., and Hauck, S., "Runtime and quality tradeoffs in FPGA placement and routing," in Proceedings of the 9th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2001, pp. 29-36.

[55] Nasre, R., Burtscher, M. and Pingali, K. "Atomic-free irregular computations on GPUs," *Proceedings of the 6th Workshop on General Purpose Processor Using*

Graphics Processing Units, GPGPU-6, pages 96-107, New York, NY, USA, 2013. DOI= http://dx.doi.org/10.1145/2458523.2458533

[56] Nasre, R., Burtscher, M., and Pingali, K. "Data-driven versus topology-driven irregular computations on GPUs," *IEEE 27th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 463-474, 2013. DOI= http://dx.doi.org/10.1109/IPDPS.2013.28

[57] Nasre, R., Burtscher, M. and Pingali, K. "Morph algorithms on GPUs," *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 147-156, New York, NY, USA, 2013. DOI=http://dx.doi.org/10.1145/2442516.2442531

[58] Nguyen, D., Lenharth, A., and Pingali, K. "Deterministic Galois: on-demand, portable, and parameterless," Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '14), pp. 499-512, Mar. 1-5, 2014, DOI= http://dx.doi.org/10.1145/2541940.2541964

[59] Nvidia Corporation. Kepler White paper for the GK110 architecture, 2012.

[60] Nvidia-Corporation. Nvidia CUDA C Programming Guide, 2012.

[61] Pingali, K., et al. "The tao of parallelism in algorithms." ACM/SIGPLAN Conf. Programming Language Design and Implementation (PLDI '11), pp. 12-25, June 4-8, 2011, DOI=http://dx.doi.org/10.1145/1993498.1993501

[62] Shavit, N., and Touitou, D., "Software transactional memory," *ACM/SIGACT-SIGOPS Int. Conf. Principles of Distributed Computing (PODC '95), pp. 204-203, Aug. 20-23, 1995. DOI=http://dx.doi.org/10.1145/224964.224987

[63] J. Rose. "Parallel global routing for standard cells," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 9(10):1085–1095, October 1990.

[64] Rose, J., Luu, Yu, C., Densmore, O., Geoders, J., Sommerville, A., Kent, K. B., Jamieson, P., and Anderson, J. "The VTR project: architecture and CAD for FPGAs from Verilog to routing," in *Proc. ACM/SIGDA Int. Symp. FPGAs*, Feb. 2012, pp. 77-86. DOI=http://doi.acm.org/10.1145/2145694.2145708

[65] Rubin, F., "The Lee Path Connection Algorithm," IEEE Trans. Cornputers, Sept. 1974, pp. 907 - 914.

[66] Rubin, R. and DeHon, A., "Timing-driven PathFinder pathology and remediation: quantifying and reducing delay noise in VPR-PathFinder," in Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2011, pp. 173-176.

[67] So, K., "Enforcing long-path timing closure for FPGA routing with path searchers on clamped lexicographic spirals," in Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, CA, Feb. 2008, pp. 24-34.

[68] Sundell, H., and Tsigas, P., "Fast and lock-free concurrent priority queues for multi-thread systems." IEEE Int. Symp. Parallel and Distributed Processing (IPDPS '03), Apr. 22-26 2003, http://dx.doi.org/10.1109/IPDPS.2003.1213189

[69] Swartz, J. S., Betz, V. and Rose, J. "A fast routability-driven router for FPGAs," ACM/SIGDA Int. Symp. FPGAs (FPGA '98), pp. 140–149, Feb. 22-24, 1998. DOI=http://dx.doi.org/10.1145/275107.275134

[70] Tessier, R., "Negotiated A* routing for FPGAs," in Proceedings of the 5th Canadian Workshop on Field Programmable Devices (FPD), Montreal, Quebec, Canada, June, 1998.

[71] Verma, A., et al. "Synthesis of floating-point addition clusters on FPGAs using carry-save arithmetic," Int. Conf. Field Programmable Logic and Applications (FPL '10), pp. 19-24, Aug. 31- Sep. 2, 2010.

[72] Xilinx Corporation. Virtex-6 FPGA DSP48E1 Slice User Guide UG369 (v1.2), September 16, 2009.

URL=http://www.xilinx.com/support/documentation/user_guides/ug369.pdf

[73] Ye, A., "Using the minimum set of input combinations to minimize the area of local routing networks in logic clusters containing logically equivalent I/Os in FPGAs," IEEE Transactions on Very Large Scale Integration Systems (TVLSI), vol. 18, no. 1, pp. 95-107, January 2010.

[74] Zhu, C., Wang, J., and Lai, J. "A novel net-partition-based multithreaded FPGA routing method," Int. Conf. Field Programmable Logic and Applications (FPL '13) Sept. 2-4, 2013, DOI=http://dx.doi.org/10.1109/FPL.2013.6645563

[75] Xilinx. Website, http://www.xilinx.com.

[76] Xilinx. Virtex-7 Data Sheet. http://www.xilinx.com.