CAD Tools and Architectures for Improved FPGA Interconnect

by

Oleg Petelin

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

CAD Tools and Architectures for Improved FPGA Interconnect

Oleg Petelin
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2016

The FPGA routing architecture consists of routing wires and programmable switches which together account for a significant fraction of the fabric delay and area, making interconnect optimization a key topic for FPGA architects. Moreover, the rapid growth of wire RC delay with process scaling has increased the importance of efficiently using the different layers available in the metal stack.

We enhance architecture models and tools to allow exploration of interconnect topologies that use a mix of wire types connected in complex ways. We use our enhancements to investigate interconnect hierarchies that take advantage of routing on the upper layers of the metal stack, and show 5-13% delay improvements compared to previously-published routing topologies. To promote insight and fast routing architecture design iteration, we develop Wotan, an analytic tool to estimate interconnect routability without benchmarks; we show that Wotan is in good agreement with the full VPR CAD flow while requiring much less compute effort.

Acknowledgements

First and foremost, I would like to offer my sincerest thanks to Professor Vaughn Betz. Always willing to make time for discussions, Vaughn has provided valuable guidance in all aspects of this thesis. I could not have asked for a more patient, hard-working or technically savvy supervisor. But more than that, Vaughn has also been a mentor, and I'm grateful for the many lessons that he has shared beyond the formal and the academic.

I would like to thank Lattice Semiconductor who have provided funding for this research, as well as Jun Zhao, Duan-Ping Chen, Brad Sharpe-Geisler and David Rutledge for their helpful discussions and suggestions. I would also like to acknowledge the computing resources provided by SciNet, whose gpc supercomputer at the SciNet HPC consortium was invaluable for the timely generation of many of the results presented in this thesis.

I would like to thank Jeff, Kevin and Mohamed, whose insightful advice has consistently raised the bar for my research and increased the quality of its presentation. A special thanks also goes to Zeng Hanqing, whose ideas and hard work as a summer student have made valuable contributions to the completion of this thesis. I am also grateful to my lab mates for the many interesting experiences outside the lab – the lunch runs, interesting discussions, bar outings and hard-learned lessons in proper canoeing have been a highlight of my graduate experience.

I am grateful to the many family and friends who have supported me during my studies. I am thankful to George and Jessica for their seemingly boundless support. Svetlana, Alex and Tom, though many kilometres separate us, they do not attenuate your love and wisdom. Galina and Vladislav, you bring out my curiosity and love for the sciences, and will always be a source of inspiration. I would also like to extend a very special thanks to my friends Sean, Michael, Larry, Mario, Rocky and Ken. I feel fortunate to call your friendship and support a constant in my life, and our many interesting adventures have provided much-needed breaks from acting like responsible adults.

Last but not least, I would like to thank Elizabeth. This acknowledgement is the final note in this thesis; a long and interesting journey that would not have been possible without your love and support!

Contents

List of Tables

List of Figures

List of Acronyms

ASIC Application-specific integrated circuit.

BEOL Back end-of-line. Integrated circuit manufacture phase dealing with interconnect.

BLE Basic logic element.

BRAM Block RAM.

CAD Computer-aided design.

CB Connection block.

COFFE Circuit Optimization for FPGA Exploration. An open source transistor sizing tool for FPGAs.

CPU Central processing unit.

DSP Digital signal processor.

FF Flip-flop.

FPGA Field-programmable gate array.

GPIO General-purpose input/output.

HDL Hardware description language.

HLS High level synthesis.

I/O Input/output.

ITRS International Technology Roadmap for Semiconductors.

LB Logic block.

LUT Look up table.

MST Minimum spanning tree.

MWTA Minimum-width transistor areas.

QoR Quality of results.

RAM Random access memory.

RC Resistance-capacitance.

ROM Read-only memory.

RR Graph Routing resource graph.

SB Switch block.

SRAM Static RAM.

VPR Versatile Place and Route. An open source tool to pack, place and route circuits.

VTR Verilog-to-Routing. A collection of open source tools to elaborate, map, pack, place and route Verilog circuits.

Chapter 1

Introduction

1.1 Motivation

Field-Programmable Gate Arrays (FPGAs) consist of an array of logic resources and a reconfigurable interconnct fabric that, with suitable programming, allows the logic to be connected to meet the needs of any application circuit. FPGAs play an important role in the integrated circuit ecosystem, filling a gap between fast, efficient, but hard-wired application-specific integrated circuits (ASICs), and slower, flexible microprocessors that execute a user-specified stream of instructions on general-purpose compute units. The reconfigurable hardware of an FPGA device lends it flexibility, at the expense of performance, compared to a hard-wired ASIC [1], while the ability to target the reconfigurable logic fabric for specific applications gives FPGAs significant speed and power advantages over microprocessors [2].

FPGAs are commonly used in applications where lower sales volumes do not justify the high costs of ASIC manufacture, but which require parallel computing and power performance not accessible to microprocessors. Such applications include medical imaging, networking and aerospace, to name a few. Recently, FPGAs have also emerged as a possible architectural solution to the challenges faced by Moore's Law [3] [4] [5]. While improvements to microprocessor power and compute throughput due to semiconductor scaling have stopped or slowed [4], the reconfigurable fabric of FPGAs promises to improve power and performance by accelerating compute-intensive tasks in multiprocessor systems.

As the integrated circuit fabrication process improves, FPGAs stand to benefit from the increasing transistor count, allowing them to implement ever larger and more complex designs. However, increasing design complexity and the challenges faced by fabrication process scaling demand continued FPGA architecture innovations to maintain the performance improvements expected with each device generation.

For a typical FPGA design, most of the delay and 50% or more of the area is due to the reprogrammable interconnect [6] [7], so its optimization is a priority for FPGA architects. The subject of FPGA interconnect design is a complex topic involving many interacting decisions, including the length of prefabricated wires, wire electrical characteristics, and the pattern of programmable switches used to stitch together the interconnect fabric and the logic array. Moreover, interconnect RC delay has increased rapidly with process scaling, particularly for connections implemented on the lower metal layers. While the lower layers of the metal stack feature a large number of thin wires, higher layers have fewer, thicker wires that are difficult to reach but have significantly smaller RC delay, motivating investigations of interconnect topologies that take advantage of the relative RC characteristics and wire abundance of

the different metal layers.

Prior published explorations of FPGA interconnect have focused on simple switch topologies and have investigated questions of interconnect architecture largely in isolation, studying the effect of prefabricated wire segment length [8], wire electrical characteristics [9], and switch patterns [10], with limited investigation of the interdependence between these topics, or the effect of different metal layer characteristics on efficient interconnect topologies. This thesis makes three main contributions to the topic of FPGA interconnect design:

1. To allow explorations of complex interconnect topologies, we enhance the open-source FPGA tool VPR [7] to enable flexible, but concise, switch pattern descriptions and to allow VPR to properly optimize for these topologies without hard-coding for specific interconnect patterns. We also create VPR architecture files for the 22nm process node to enable interconnect explorations that accurately reflect current design challenges.

2. We investigate routing topologies that mix wire types from two different metal stack layers by simultaneously exploring the interconnect hierarchy and the choice of wire segment length, and suggest rules of thumb to take advantage of fast, but scarce, wires on the upper layers of the metal stack. We also investigate the effect of unidirectional (direct-drive) wiring on the interconnect switch patterns.

3. We investigate metrics to promote insight and faster iteration for early-stage interconnect explorations. The standard exploration methodology uses the full FPGA tool flow which provides accurate estimates of the overall design performance, but does not necessarily promote insight into the performance of specific interconnect architecture decisions. We develop Wotan, a tool which evaluates the routability of FPGA interconnect using a combination of analytic and heuristic methods. We show that Wotan routability estimates are in good agreement with the full VPR tool flow across a wide range of routing topologies, while requiring a significantly smaller compute effort.

1.2 Organization

The rest of this thesis is organized as follows. Chapter 2 discusses basic FPGA concepts, prior work in the area of FPGA interconnect design, and motivation for further investigation of complex interconnect. Chapter 3 discusses our VPR enhancements which enable complex interconnect topology explorations, as well as our new 22nm VPR architecture files. Chapter 4 investigates the effect of unidirectional (direct-drive) wiring on interconnect switch patterns and explores optimized hierarchies to take advantage of wire segments on the fast, but scarce, upper layers of the metal stack. Chapter 5 discusses Wotan, a tool for estimating interconnect routability without the use of benchmark circuits. Chapter 6 concludes and discusses directions of future work.

Chapter 2

Background

2.1 The Semiconductor Metal Stack and Interconnect Scaling

The metal stack of a semiconductor chip refers to layers of wires separated by dielectric material with successive layers connecting to each other through vias. While earlier generations of semiconductor processes relied on a combination of aluminium and tungsten to form wires and vias, the pressures of decreasing interconnect dimensions have motivated the use of less-resistive copper interconnect. Copper interconnect is added in layers on top of the chip substrate during the back end-of-line (BEOL) fabrication steps using a Damascene process. A cross section of the metal stack used in Intel's 14nm process is shown in Figure 2.1 [11].

The metal layer stack consists of a large number of wires with a small cross-section close to the silicon and a smaller number of wider, taller and less resistive wires on the upper metal layers. The thin wires close to the silicon are naturally useful for the multitude of short-distance connections on the chip, while the larger less resistive wires further up in the metal stack lend themselves to long-distance connections, clocking, and power distribution. Figure 2.2 shows the basic features of the metal stack in a typical copper Damascene process [12]. Copper is used to fill vias and wire trenches, which are

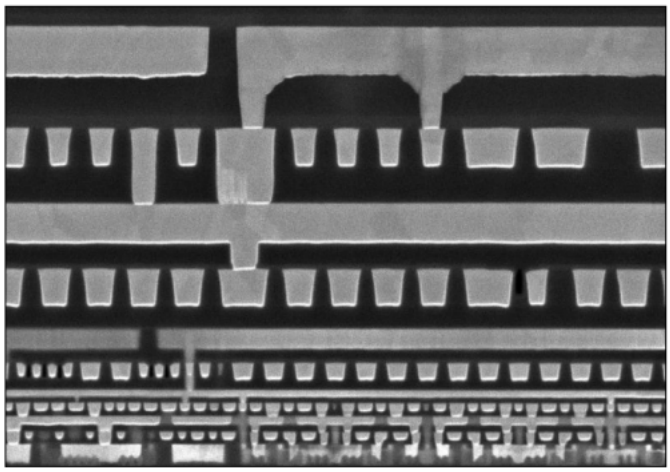

Figure 2.1: The metal stack in Intel's 14nm process [11].

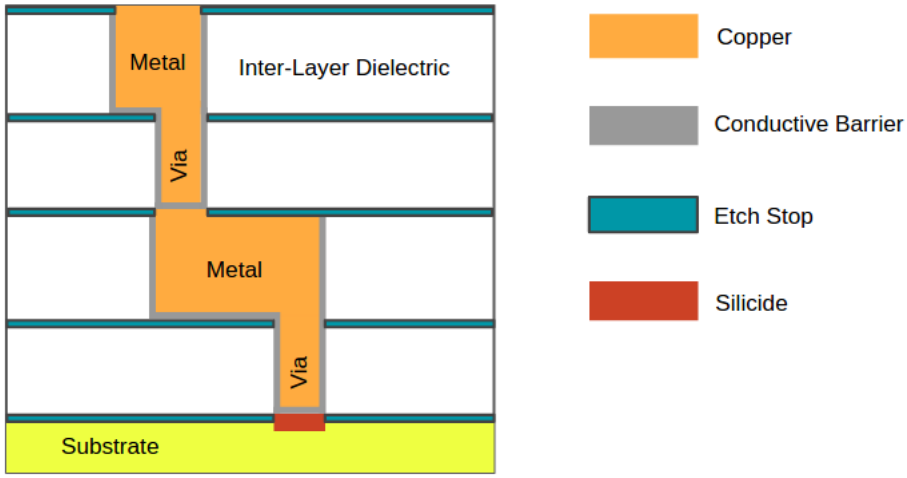

Figure 2.2: Copper interconnect in a Damscene process.

lined with a conductive barrier metal (such as tantalum) to prevent the diffusion of copper atoms into the surrounding dielectric. A silicide – a mixture of silicon and a conductive metal – facilitates the connection between the metal layers and the transistor terminals in the silicon substrate below.

2.1.1 Interconnect Scaling

The International Technology Roadmap for Semiconductors (ITRS), a set of semiconductor outlook documents published by industry experts, generally classifies the types of wires in the metal stack into the M1, intermediate, semi-global and global layers, with each consecutive layer located further from the substrate [13]. Wires higher in the metal stack increase in pitch and cross-section, as can be seen in Figure 2.1. Semi-global wire dimension and pitch are 2x of intermediate wires, and global wires are 4x+ of intermediate wires (M1 wires have the same dimensions and pitch as the intermediate layer, but have slightly different electrical characteristics by virtue of their proximity to the substrate).

Wire dimensions and pitch are generally scaled by the same factor. Traditional scaling saw a decrease of all physical feature sizes by a factor of S. Thus, while both wire width and thickness were reduced by S, wire length on the lower metal layers was reduced by S as well, keeping the overall intrinsic (self-loaded) wire delay roughly constant for short connections[1] (the delay increase of long wires spanning the same physical distance could be kept linear, as opposed to quadratic, by breaking up a single long wire into multiple buffered segments) [12]. In addition, the resistance of short wires has traditionally been negligible compared to driver resistance, having a delay impact for only the longest connections.

Unfortunately, the optimistic outlook of traditional interconnect delay scaling has not held true with recent process generations, and wire resistance has increased rapidly for two main reasons [13]:

- Electron scattering on grain boundaries, interfaces and side-walls significantly increases the resistance of nano-scale copper interconnect.
- The thickness of the conductive diffusion barrier layer (see Figure 2.2) has not scaled in proportion to the copper fill. Since barrier layer metals have significantly higher resistance compared to copper, the rising proportion of barrier metal in the scaling interconnect has further increased resistance.

[1] The delay through a wire is proportional to the square of the wire length.

The impact of increasing interconnect resistance has already been felt in commercial devices [14], and the challenges of resistance scaling have led ITRS to project that at the current rate, interconnect will become the major delay bottleneck for semiconductors in the near future [13].

The challenge of interconnect scaling is a difficult one, but emerging manufacturing and device architecture solutions can mitigate the impact. Copper replacement metals such as silver or silicides can reduce wire resistance, but still suffer from electron scattering effects in addition to reliability issues [13]. Entirely new interconnect materials, such as carbon nanotubes, have low resistance and do not face electron scattering issues, but pose significant integration challenges [15] [13]. On the architecture front, 3D chip stacking can reduce the required wire length to connect device modules and thereby reduce delay, but 3D stacking still faces significant thermal and reliability issues [16] [13].

Working with a given manufacturing and interconnect process, high-level interconnect topology changes and improvements to computer-aided design (CAD) tools can also improve the overall efficiency with which the available interconnect is used [17] [18]. In this thesis we explore CAD tools and interconnect architectures to take advantage of routing with a heterogeneous mix of wires types across different layers of the metal stack, as well as efficient and accurate interconnect metrics that promote faster iteration during early-stage routing architecture explorations.

2.2 FPGA Architecture

Field-Programmable Gate Arrays (FPGAs) are devices with reconfigurable logic and interconnect elements that, with suitable programming, can implement virtually any digital circuit. While many applications have sales volumes that justify the manufacture of application-specific integrated circuits (ASICs), the high cost of semiconductor manufacturing prohibits ASIC use in other areas such as high-end networking, aerospace and medicine, to name a few. Though the programmable hardware of FPGAs carries a power and performance overhead when compared to ASICs [1], FPGAs have an order of magnitude power efficiency and performance advantage over processors for data-parallel computation [2], making them ideal for applications where ASICs are too expensive and processors too slow[2].

The main elements of an island-style FPGA architecture are shown in Figure 2.3 [7]. An array of logic blocks (LBs) provides the main computational fabric of the FPGA. Each LB can be programmed to behave as a combinational or sequential sub-component of a larger circuit, where the larger circuit is built by connecting multiple LBs together through the surrounding programmable routing fabric. The programmable routing fabric consists of both horizontal and vertical channels made up of wire segments that span one or more logic blocks. Wire segments can connect to each other through structures called switch blocks (SBs) which have programmable switches that, when selectively enabled, allow multiple wire segments to form an uninterrupted path through the FPGA. Logic blocks connect to wire segments in the horizontal and vertical routing channels through structures called connection blocks (CBs). Each output or input pin of a logic block may connect to one or more wires using a pattern prescribed by the CB. Input/output (I/O) blocks are typically positioned on the perimeter of the FPGA and provide an interface with external devices. Besides LBs and I/O, other FPGA blocks are also common. In addition to various serial and parallel interfaces to complement general-purpose I/O (GPIO), almost all FPGAs also include random access memory (RAM) and digital signal processing (DSP) blocks to

[2]Compared to GPUs, FPGAs are inferior in floating-point performance and memory bandwidth, though recent architectural changes may overcome the floating-point gap [19].

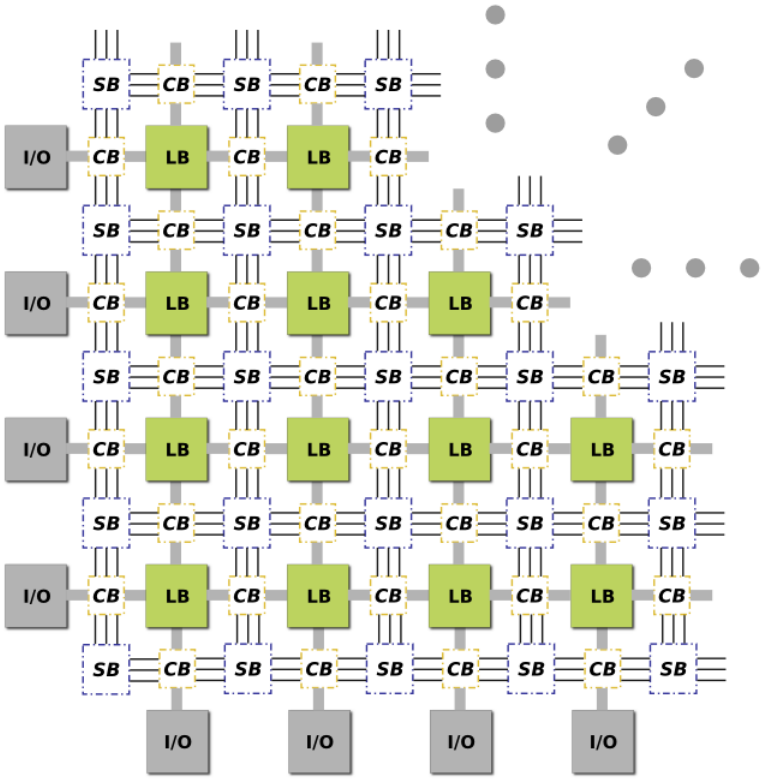

Figure 2.3: The basic elements of an island-style FPGA architecture.

perform memory management and computation tasks which can not be done efficiently with LBs. In many architectures block RAMs (BRAMs) and DSPs have similar width as LBs, but span multiple tiles vertically.

A logic block and the surrounding interconnect form a tile as shown in Figure 2.4; replicating this basic tile forms the programmable array of logic that allows an FPGA to act like any target digital circuit. Each vertical and horizontal routing channel contains W wires, referred to as the channel width. The channel width does not have to be constant throughout the FPGA, and certain commercial devices have a different W in the horizontal and vertical directions due to physical layout considerations [6].

The concept of routing element *flexibility* was introduced in [20] to quantify routing architecture design. Flexibility represents the number (or fraction) of possible resources that each wire or pin connects to and is defined for both switch blocks and connection blocks. Switch block flexibility, F_S, is the number of wires to which each wire segment connects inside the switch block. The destination wires are typically distributed evenly on the one opposite and two adjacent sides of the switch block and Figure 2.4 shows a switch block where wire endpoints connect to $F_S = 3$ wire start points (the most common SB flexibility). Connection block flexibility is defined separately for block input and output pins. The input connection block flexibility $F_{C,in}$ represents the fraction of wires in the channel to which each input pin can connect; Figure 2.4 shows an input connection block flexibility of $F_{C,in} = 0.5W$. The output connection block flexibility, $F_{C,out}$ is defined similarly as the fraction of channel wires to which each output pin can connect and is $F_{C,out} = 0.5W$ in Figure 2.4. Values of connection block flexibility are typically below $0.2W$ for cluster-based logic blocks [21] [22]. As with channel width, flexibility does not have to be constant throughout the FPGA and may be varied based on the wire type or block pin.

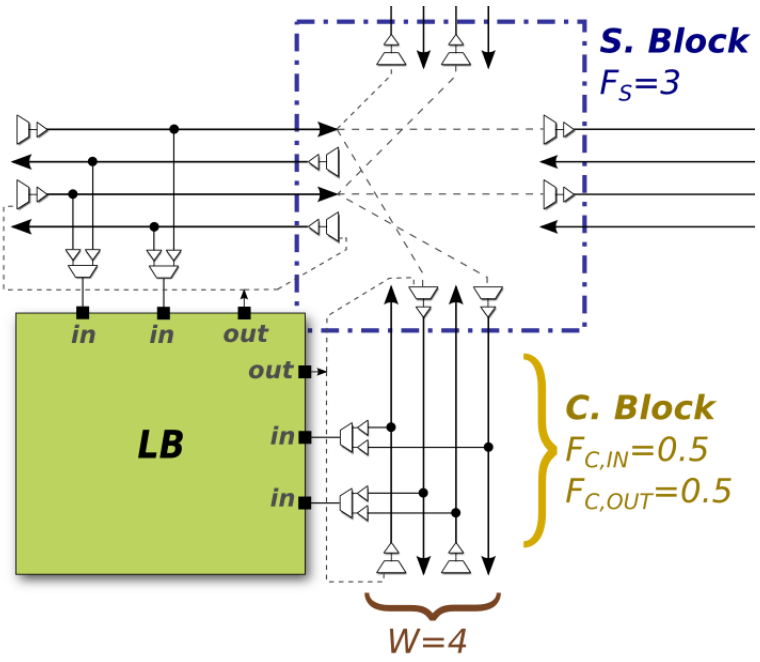

Figure 2.4: Routing elements of an FPGA tile. Not all switch block connections are shown.

Each wire segment in a routing channel has a start point and end point coinciding with FPGA switch blocks, and the number of tiles that a wire segment spans defines the wire length of that segment. Figure 2.4 shows length-1 wires that start/end in adjacent switch blocks. Other lengths of wires are possible and wire length considerations will be discussed later in Section 2.2.2.

A common choice of logic block [21] [23] is a cluster of N basic logic elements (BLEs) shown in Figure 2.5. The most common parts of a BLE include a K-input look-up table (LUT) together with a flip-flop (FF) and a multiplexer (or "mux" for short). The LUT can implement any logic function of the K inputs, and the FF/mux allow the BLE to act as a combinational or sequential element. The I inputs and O outputs of the logic block may interface with input/output crossbars which can help steer signals to their destinations. With an input crossbar LUTs may be used with a high degree of utilization without needing an LB input pin for every LUT input (i.e. $I < KN$) [24]. While crossbars are often included [6], they can be of varying sparsity[3] [25] or may be absent entirely [26]. An optional feedback path through the input crossbar can also be included between BLE outputs and inputs, and can ease demand for the scarce routing resources outside the logic block.

Almost all logic blocks used in commercial devices have the basic clustered architecture shown in Figure 2.5, but many variations and enhancements are possible.

- **Fracturable LUTs**. Since two K-LUTs can implement one $(K+1)$-input LUT with the use of an extra 2-to-1 mux, many architectures have made use of *fracturable* LUTs [23] that, for example, allow one 6-LUT to be used as two 5-LUTs with some shared inputs. Fracturable LUTs allow bigger LUT sizes to be more area efficient: a bigger LUT can pack more logic (and thus decrease critical path delay) while fracturing it into smaller LUTs still allows for efficient packing of smaller blocks of logic that may naturally be more abundant [27].

[3]Crossbar sparsity refers to the internal flexibility of the crossbar; sparse crossbars are internally depopulated and cannot simultaneously connect all terminals.

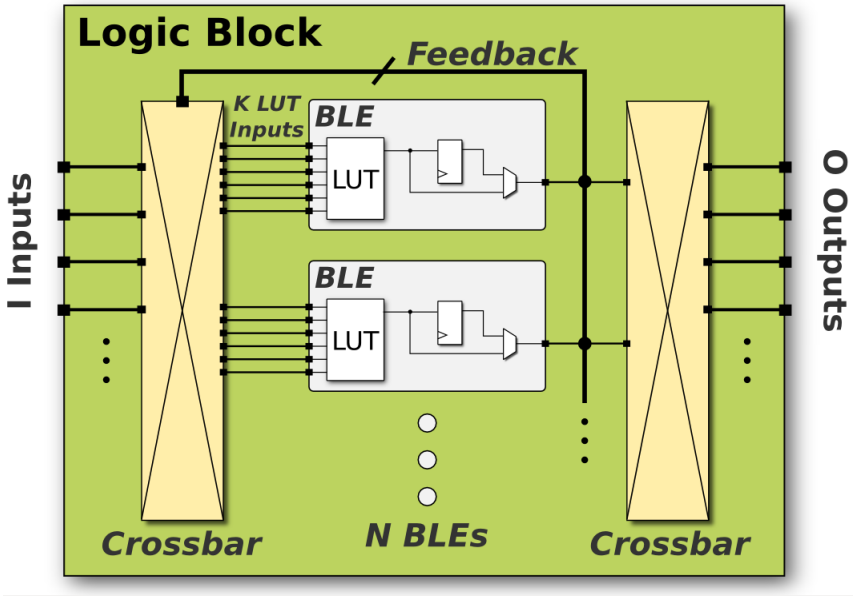

Figure 2.5: Elements of an FPGA logic block.

- **Adder hardware**. Since adders require some form of bit carry, implementing them using LUTs alone would be inefficient as it would require carry signals to propagate through slow general routing resources. Logic blocks frequently implement adder logic [23] using dedicated carry-in and carry-out lines to/from the logic block. A carry signal can then be propagated efficiently through all the logic block BLEs and on to the next logic block in the adder chain.

- **LB Memory**. A single lookup table can naturally act as a read-only memory (ROM) with 2^K addresses and a one bit output. With some modifications, logic blocks can also be used as a distributed memory resource with read and write functionality [28].

While FPGA logic blocks have typically been based on LUTs, other architectures that can implement arbitrary digital circuits have also been proposed [29].

2.2.1 Bidirectional and Unidirectional Wiring

Switch blocks implement programmable connections between adjacent wire segments; the two most common types of programmable switches are shown in Figure 2.6. The tristatable bidirectional switch allows wire segments to be driven by logic block pins and other wires at multiple points along their length[4]; and two adjacent wires are also able to drive each other, allowing a signal to travel in either direction along a wire. While earlier FPGA architectures have typically used bidirectional switches [7], unidirectional topologies are now more popular [30]. The FPGA tile in Figure 2.4 illustrated a unidirectional topology.

A unidirectional switch is composed of a mux followed by a buffer, a configuration that is also referred to as 'direct-drive'. Unidirectional/direct-drive topologies are motivated by the observation that a wire segment can be used by only one signal at a time, so if two adjacent wire segments are able to drive

[4]Simple pass gates have also been used to implement bidirectional switches [7]

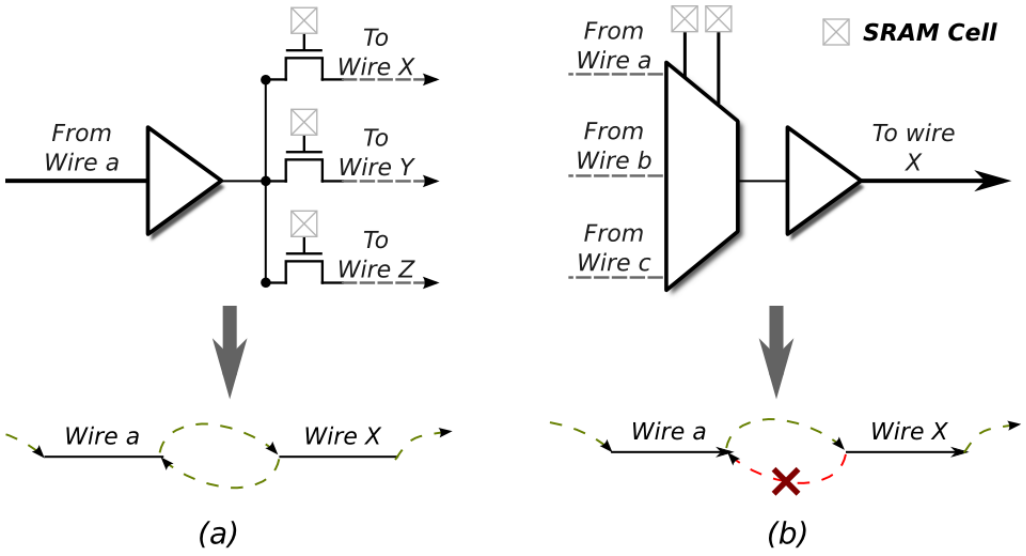

Figure 2.6: (a) A bidirectional switch is tristatable, allowing wire segments to be driven at multiple points along their length. (b) Unidirectional switches allow wires to be driven from one point only.

each other with bidirectional switches, one of those switches will always remain unused. By eliminating redundant switches, unidirectional architectures achieve an area-delay improvement of over 30% [31].

While unidirectional switches improve overall FPGA performance, they do require that wire segments be driven from one point only to avoid multiple switches from conflicting. This places restrictions on the wires that can be driven from switch blocks and connection blocks, and an FPGA architect should ensure that a sufficient number of wire drive points is available within each channel segment.

2.2.2 Wire Staggering and Segmentation

Figure 2.7 shows two possible layouts of length-2 wire segments in an FPGA channel. Staggering wire segments in a unidirectional architecture is necessary to ensure that every switch block and connection block has access to wire drive points. In a bidirectional architecture wire staggering is not necessary because each wire segment can have multiple drive points, but is still useful to allow logic block output pins convenient access to other logic blocks that are one wire length away [7].

Figure 2.7 shows a wire length of 2, but other lengths are possible. The delay through N buffered wire segments can be written as

$$T_{path} = N(T_{seg} + T_{buf}) \tag{2.1}$$

Where T_{seg} is the delay through each wire segment and T_{buf} is the delay through each buffer. Longer wire segments can help decrease delay between far-away connections by lowering N, the number of segments and buffers required (which also decreases area usage). However, wire RC delay increases quadratically with wire length, so overly long wires are slow for short connections, and at some point increasing wire length provides no further delay benefit. Another trade-off is that long wires have a decreased routing flexibility: a wire segment becomes unavailable once it is committed to route a signal, so shorter segments can be used with a higher granularity. Thus, while long wires can decrease delay and area usage due to buffers, shorter wires are more flexible and require a smaller channel width to

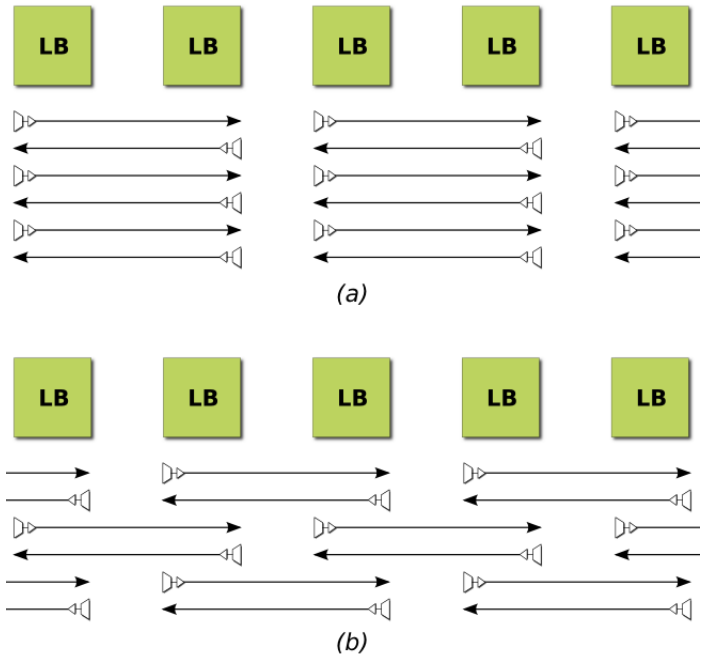

Figure 2.7: (a) No wire staggering. (b) Staggering wire segments improves the efficiency with which wires are used.

implement target circuits.

A mixture of wire lengths with different electrical characteristics is a common feature in commercial FPGAs [32] [23]. For example, Altera's Stratix series takes advantage of the relative strengths of different metal layers by implementing the majority of inter-LB routing on a slower, more accessible metal layer, while a low-resistance metal layer towards the top of the metal stack is used to implement a small number of long (length 16+) wires [23].

In contrast with commercial devices, published explorations of wire length have focused on simple bidirectional architectures and have largely ignored the question of how best to exploit a heterogeneous mix of wire types. Studies of wire length and segmentation [8] found that moderate-length wires (length-4 in the study) offer the best area-delay trade-off; the same study also explored a heterogeneous mixture of wire lengths, but did not consider how different wire types should connect.

A major question that has not received attention in published literature is how best to exploit wire types across the different layers of the metal stack. This issue is becoming increasingly important as wire resistance increases with process scaling, particularly for the thin wires at the bottom of the stack, and we revisit it in Chapter 4.

2.2.3 Switch Block Patterns

The most popular published switch block patterns were explored in the context of bidirectional architectures and length-1 wires. Figure 2.8 illustrates the three classic switch block patterns; each wire segment is labelled with a number. With reference to Figure 2.8, the three patterns are:

(a) The disjoint switch block [20], also known as the subset or planar, was the earliest pattern used in FPGAs. It was easy to physically lay out in FPGA chips, and it significantly simplified analysis of

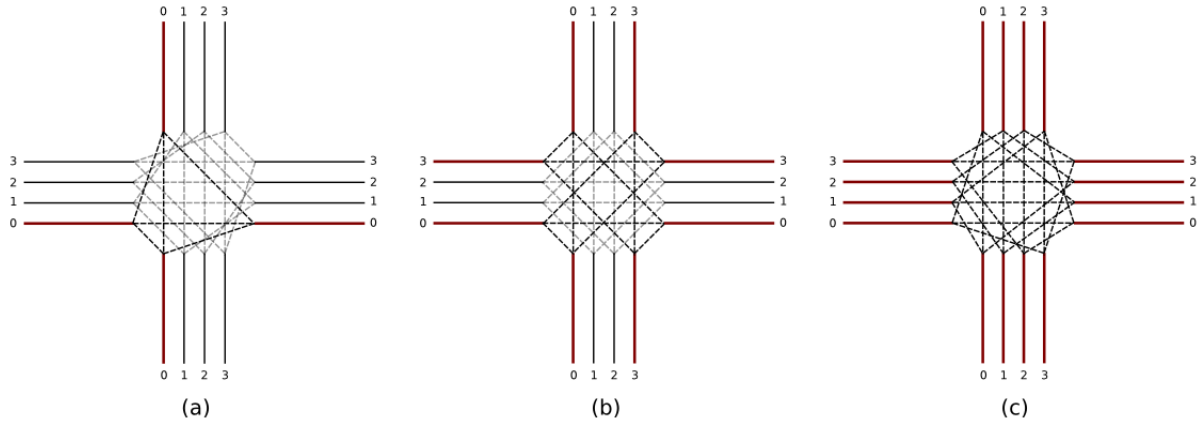

Figure 2.8: Classic switch block patterns. The wire segments highlighted in red represent a disjoint set of tracks that can connect to each other but have no possible connection to the other track indices. (a) Disjoint/subset/planar. (b) Universal. (c) Wilton.

the routing fabric. On the other hand, this switch block pattern restricted any given signal to a small subset of possible FPGA wires – as an example, in Figure 2.8 a signal on wire 0 would forever be confined to tracks with index 0.

(b) The universal switch block [33] guaranteed maximum routing capacity. Capacity is a measure of the number of signals that can be routed simultaneously, and the universal pattern could route, without conflict, as many signals as the wires allowed. However, while the universal pattern had the maximum theoretical capacity, it still restricted signals to a subset of possible track indices as shown in Figure 2.8 (though the number of tracks available to each signal was greater compared to the disjoint switch block).

(c) The Wilton switch block [10] applied a twist to the universal pattern which guaranteed that a signal can eventually find its way onto any track in the FPGA by taking the appropriate twists and turns through the routing. As highlighted in Figure 2.8, all wires are part of one domain, and this flexibility has made the Wilton switch block more popular compared to the disjoint and the universal patterns.

Switch Block Permutation Functions

Switch block patterns can be described in terms of mathematical permutation functions [34], which specify the index of the destination track as a function of the source track and the source/destination switch block sides. Figure 2.9 illustrates the possible permutation functions for a bidirectional switch block with length-1 wires.

Table 2.1 lists the permutations functions for the classic switch patterns discussed earlier. The mappings are specified as a function of t, the index of the source track and all evaluations are implicitly modulo W to ensure that the computed destination track index is within the bounds of the channel width. Functions of the form $(t + A)$ simply increment/decrement the track index, while functions of the form $(W - t + B)$ implement a twist. Note that the permutation functions of Table 2.1 apply to a bi-directional topology; for each of the six functions the reverse connection is implicit. Unidirectional architectures can use 12 permutation functions; the additional six implement connections in the reverse directions compared to those shown in Figure 2.9, and need not correspond to the forward permutations.

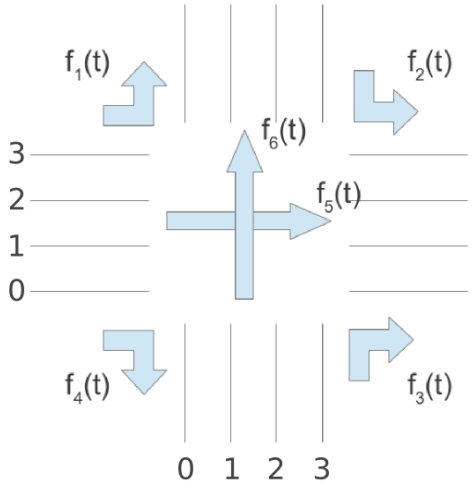

Figure 2.9: Bi-directional permutation functions are specified for each switch block side.

Table 2.1: Permutation functions for the three classic switch block patterns.

Function	Disjoint	Universal	Wilton
$f_1(t)$	t	W-t-1	W-t
$f_2(t)$	t	t	t+1
$f_3(t)$	t	W-t-1	W-t-2
$f_4(t)$	t	t	t-1
$f_5(t)$	t	t	t
$f_6(t)$	t	t	t

Permutation functions greatly simplify the specification of switch blocks, and their use for unidirectional architectures and complex interconnect topologies is further discussed in Sections 3.2.1 and 4.3.

Switch Blocks at Longer Wire Lengths

Switch blocks that were studied in the context of length-1 bidirectional architectures were easily adapted to longer wire lengths [34] [35]. Figure 2.10 illustrates how length-4 wires can be incorporated into an FPGA tile (here the channel width is 16). A twist can be applied to each wire such that one tile can be stamped across the entire FPGA while maintaining the same switch pattern. The different parts of the switch block are highlighted:

- Wire endpoint to endpoint connections can be represented with the permutation functions f_e.
- Connections between wire endpoints and midpoints can be represented by functions f_{me}.
- Connections between wire midpoints can be represented by functions f_m.

The classic switch block patterns were adapted to longer wire lengths by simply reusing the pattern at each possible endpoint and midpoint connection (since bidirectional switches are tri-statable). Notably, the Imran switch block [36] has been the only published topology that specifically exploits long wires, though it still assumes bidirectional routing. The Imran pattern uses the Wilton switch block to connect to wire endpoints and the disjoint switch block to connect to wire midpoints – since the disjoint switch

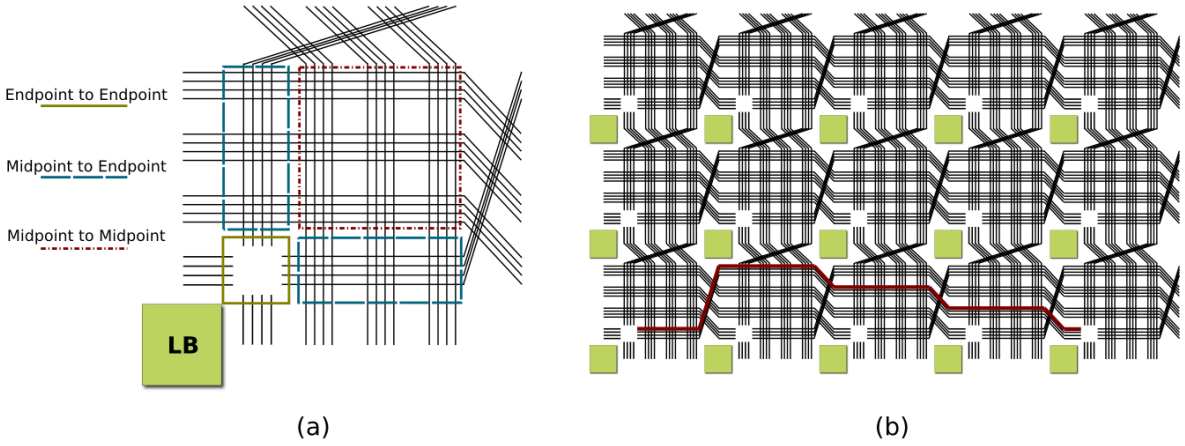

Figure 2.10: (a) Representation of a switch block with length-4 wires. (b) Each FPGA tile has the same switch block pattern; a length-4 wire is highlighted.

block connects within the same track index, using this pattern at wire midpoints eliminates a routing switch while still allowing midpoint connections to bend in both directions using one switch only.

While bidirectional switch blocks have been extensively studied, there have been no published explorations of switch block patterns for unidirectional wiring, and academic efforts have simply adapted the classic bidirectional switch blocks as best as they could [37]. Significantly, there have also been no published studies of switch block patterns to specifically exploit different types of wires (such as wires on different metal layers). We come back to these questions in Chapter 4.

2.2.4 Connection Block Patterns

Connection blocks do not have an analogue of the well-defined, named switch block patterns. While switch blocks steer signals across long distances and potentially congested areas of the chip, the purpose of the connection block is to act as a start/end point for signals looking to use the inter-LB routing fabric. Though significant improvements to connection blocks patterns may be possible, academic architectures have been content to ensure that an appropriate $F_{c,in}$ and $F_{c,out}$ are used, and that the switch pattern of each pin does not interfere significantly with that of other pins [37].

2.3 CAD for FPGAs

With hundreds of thousands of logic blocks and tens of millions of programmable switches, it is not feasible to implement circuits on FPGAs using a fully manual effort. Instead, computer-aided design (CAD) tools hide much of the implementation complexity and leave the circuit designer to focus on higher-level tasks. Figure 2.11 illustrates the basic FPGA CAD flow [7] [38].

Inputs: A hardware description language (HDL) such as Verilog or VHDL is typically used to describe the functionality of the target circuit, and software is subsequently used to infer the various hardware blocks from the programming keywords and constructs. HDLs typically mirror the structure of the intended hardware closely, but recent efforts have made practical the translation of algorithmic languages (like C++) into HDLs, and this high-level synthesis (HLS) step is sometimes included on top

of the CAD flow described here. Various performance targets that the CAD flow should strive to meet can also be specified as an input.

Elaborate: The elaborate step translates an HDL into a netlist of basic circuit elements. This step is independent of the underlying FPGA hardware, and basic hardware components such as logic gates, multiplexers, flip-flops and memories are inferred.

Optimize: The optimization step simplifies the logic relations inferred during elaboration and removes unnecessary components while preserving circuit functionality.

Map: This step maps the optimized technology-independent circuit elements into the basic hardware elements that are actually present in the FPGA (i.e. LUTs, muxes, FFs, etc).

Pack: The function of this step is to pack the various mapped primitive elements into the large physical blocks available on the FPGA. Thus the netlist of LUTs, FFs, and other basic elements is now *clustered* into logic blocks, BRAMs, DSPs, etc.

Place: The circuit netlist has now been coarsened to the large FPGA-specific logic/memory/DSP blocks. The placement step chooses which FPGA physical blocks are actually used by the target circuit.

Route: This step operates on the coarsened netlist of placed FPGA blocks to connect all the required input and output pins. The routing switches of the FPGA are selectively turned on so that each signal has a separate and contiguous path to its destination through the inter-LB wiring.

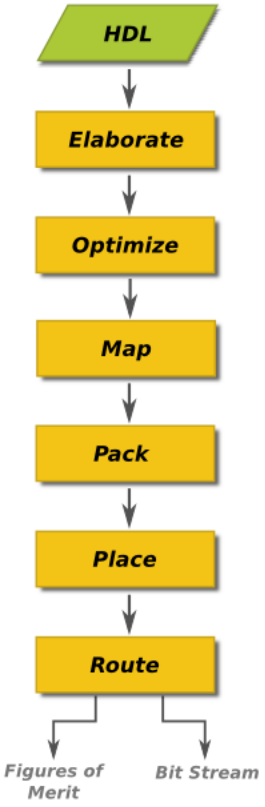

Figure 2.11: The basic FPGA CAD flow.

Outputs: The outputs of the FPGA CAD flow produce estimates of various figures of merit that help the designer determine whether the implemented circuit has met performance targets. These estimates may include circuit operating frequency, power consumption and resource usage, along with a detailed breakdown of the factors that have limited circuit performance. If the designer is satisfied, a generated bitstream can be uploaded to the FPGA device to implement the target circuit.

Optimization is key, and the primary goal of every step in the CAD flow is to to use the FPGA device as efficiently as possible to reduce the critical path delay, power consumption and area of the implemented circuit. The FPGA CAD flow is a very complicated process, and while combining multiple steps together might theoretically allow for a more holistic optimization, splitting up the CAD flow allows the tools to better manage complexity.

2.3.1 The VTR CAD Tool Flow

The Verilog-to-Routing (VTR) project implements the FPGA software flow for academic CAD and architecture research [37] [38].

As shown in Figure 2.12, the VTR flow consists of three open-source tools:

ODIN II [39] performs architecture-aware elaboration of the input HDL to parse the soft logic intended for FPGA logic blocks, as well as to infer hard FPGA blocks from the code (such as BRAMs and DSPs).

ABC [40] performs logic optimization and technology mapping to LUTs.

VPR [7] performs the pack, place and route steps of the CAD flow for the specified FPGA architecture.

In addition to the HDL file, the VTR flow accepts an architecture description file that allows users of the tools to specify a broad range of target FPGA architectures, varying parameters like LUT size, hard block architecture and routing topology.

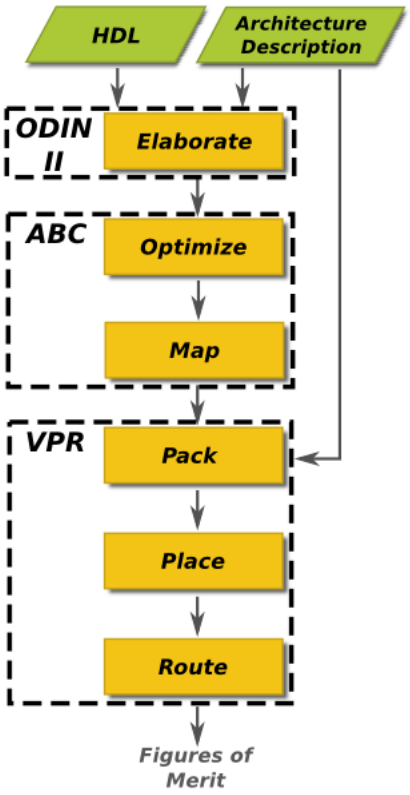

Figure 2.12: The VTR flow.

2.3.2 Architecture and CAD Exploration with VPR

The Versatile Place and Route (VPR) tool constructs a detailed FPGA architecture to pack, place and route a technology-mapped circuit netlist. The architecture description passed to VPR contains timing and area information for common FPGA circuit elements such as routing wires, switches and look-up tables, as well as the desired architecture and connectivity of various FPGA blocks, and this description is used to create a detailed software model of the target device. Though VPR does not generate a bit stream to program a physical FPGA chip, each intermediate step in the VPR flow produces quality of results (QoR) estimates, and the analysis done following the routing step gives an estimate of area usage, critical path delay and power usage (if desired) of the target circuit. This ability to model the performance of different FPGA architectures makes VPR a useful tool for exploring FPGA architecture questions. Additionally, VPR prints various metrics to estimate the quality of the algorithms used at each step in the flow, making it useful for exploring FPGA CAD problems as well.

Two common VPR flows are described below:

1. **Min-W**. VPR can be run to find the minimum channel width at which a circuit is routable. This is a measure of architecture *routability* which represents the efficiency with which FPGA routing resources can be used; a higher routability implies that a smaller channel width is needed for an FPGA device using this architecture. To find the minimum channel width, VPR performs routing multiple times after packing and placement have been completed, varying the channel width according to a binary search. The binary search produces a final routing with a lot of resource contention between signals, so to evaluate the critical path delay accurately, a $1.3 * W_{min}$ channel width is typically used afterwards.

2. **Fixed-W**. Physical FPGA devices implement a fixed channel width, so a fixed-W flow is frequently used to evaluate architectures, especially when the target circuits being used for evaluation are large, prohibiting the significantly longer runtime of a min-W flow. While a fixed-W flow is faster and is useful for quickly evaluating circuit delay, it can also be used to (more coarsely) evaluate

routability by running a large suite of circuits through the CAD flow to see what fraction of them will route.

Area, delay, power and routability can all be evaluated with a min-W or fixed-W flow. While a min-W flow estimates routability and routing area usage more accurately, a fixed-W flow is significantly faster and more-closely mirrors the end-user experience with a commercial FPGA.

2.3.3 Benchmarks and the Need to Average

Experiment results can change based on the circuit being used to evaluate a CAD algorithm or architecture feature. For example, memory-heavy circuits may emphasize the use of BRAMs and distributed memories, while bandwidth-intensive circuits can place increased pressure on the FPGA interconnect. It is therefore important to average experiment results over a large and relevant set of benchmark circuits so that accurate and general conclusions can be drawn.

As the amount and heterogeneity of FPGA resources changed, so has the size and complexity of benchmark circuits. The set of MCNC20 benchmarks [41], traditionally used for exploring FPGA architecture and CAD, contains 20 circuits of up to 8,381 4-LUTs in size. The MCNC benchmarks have since been superseded by the VTR benchmark set [38], which contains circuits with size ranging from 174 to 99,700 6-LUTs, and places increased emphasis on hard FPGA blocks like BRAMs and DSPs. The Titan benchmarks [22] have sought to follow the large circuits implemented in commercial devices with even greater fidelity, and this benchmark set has circuits ranging from 90,779 to 1,859,501 6-LUTs.

With new FPGA devices containing over 5,000,000 BLEs [42], it can be important to use large benchmark circuits to fully capture the architecture trends of large devices. However, the runtime required to evaluate algorithms and architectures using large benchmarks can be prohibitive, especially in the early design stages, and often times it is reasonable to use benchmarks that are "large enough" to capture the trends being investigated.

In addition to benchmark-dependence, experimental results also depend on the CAD algorithms and settings being used [43] [7]. For example, placement algorithms typically involve an element of randomness, so the initial seed influences the final experiment results; averaging the results over multiple seeds can mitigate this randomness. To arrive at general conclusions, the experiment can also be run for multiple types of architectures (e.g. vary the LUT size if interconnect wire length is being investigated).

2.3.4 Placement and Routing in VPR

The use of inter-logic block routing wires is most affected by the placement and routing stages and hence, it is important that these phases of the CAD flow be of high quality and fully exploit the features of the architectures investigated. The placer determines the relative position of logic blocks acting as sources/targets of signal connections, and as such can take some advantage of heterogeneous routing resources. The router marks the wires that signals should use and is therefore responsible for making sure that critical signals are given priority to take direct paths through the routing and to use fast routing wires if they are available.

Placement

Popular placement schemes include analytical placement [44] which optimizes a mathematical approximation to the placement problem with a matrix equation solver, and annealing-based placement [45]

which uses a randomized algorithm to try increasingly-restrictive placement positions that ideally converge at a near-optimal result. VPR uses an annealing algorithm [7], the basics of which are common across all annealing placers, and will be described in this section.

Simulated annealing can converge to a high-quality placement by randomly swapping/moving 2 or more FPGA blocks repeatedly. The cost of the new placement is compared to the pre-swap placement, and the new placement is accepted with some probability that decreases with (a) the number of swaps that have been attempted and (b) any negative impact that the new placement produces on the cost function being optimized. A new placement is accepted if:

$$r < e^{\frac{-\Delta C}{T}} \tag{2.2}$$

Where ΔC is the change in cost compared to the previous placement, T is the current 'temperature' of the anneal and r is a random number between 0 and 1. While VPR can compute cost according to a variety of schemes, timing-driven placement (which works to directly optimize circuit operating frequency) uses a cost function which combines overall circuit delay and wiring costs in equal measure; the delay cost is a timing criticality-weighed sum of the best-case routing delay of all netlist connections and the wiring cost represents an estimate of the total wire length that will be required to route the circuit [46].

The anneal used in VPR starts with a random placement and uses an initially high temperature T (that accepts all proposed placement moves); T is decreased to gradually converge at a solution. The way in which T decreases (a.k.a. the annealing schedule) is tailored to work well at all circuit sizes – the cooling rate adapts to how quickly the placement improves, and the placer performs the greatest number of moves at the most 'productive' temperatures (when a large number of, but not all, swap candidates are accepted) [7].

Timing-driven placement requires an estimate of delay between the source and target of each signal. It is possible, but impractical, to actually route the signal connection during placement, so VPR uses a pre-computed lower-bound delay estimate for each connection instead. Before placement, VPR invokes its timing-driven router to produce a 2D matrix of ideal delay values for each $(\Delta x, \Delta y)$ coordinate from some reference tile [46]. During placement this array is indexed to retrieve the minimum delay of each source/target connection.

The timing-driven placer is able to adapt to the available wiring resources, but is limited by the ability of the router to faithfully find the fastest-possible paths through the inter-LB routing. For a heterogeneous mix of wire types, the VPR 7 router may not always find the best possible path, and we revisit this issue in Section 3.2.4.

Routing

VPR represents the FPGA routing architecture through the routing resource graph [7] with graph nodes representing physical resources like wires and pins, and graph edges representing programmable switches. Signals start at special *source* nodes and terminate at *target* (a.k.a. sink) nodes which correspond to the inputs and outputs of LUTs, FFs and hard blocks like DSPs.

Since multiple signals must be routed on the same routing network, it's possible for multiple signals to compete for a single routing resource, and VPR uses a negotiated-congestion router based on the Pathfinder algorithm [47]. The base of this algorithm uses an A*-like search [48] to route each signal

individually. Nodes are assigned a *cost* and the routing resource graph is explored to find the lowest-cost path between the source and target nodes. The cost to use node n is defined as:

$$C(n) = C(s,n) + dir * C(n,t) \tag{2.3}$$

Where s is the source node, t is the target node, and dir is a directionality factor that specifies how targeted the graph exploration will be. While the path cost from source to n, $C(s,n)$ can be computed incrementally as each consecutive node is explored, the cost from n to target, $C(n,t)$ must be estimated, and the fidelity of this estimate influences run time and result quality. With a directionality factor of 1, a $C(n,t)$ lower than the actual cost will guarantee that an optimal routing path will be found, while an over-estimate of $C(n,t)$ will cause the search to be more directed towards the target node, but may not find the smallest-cost path.

The cost assigned to nodes during the routing is the sum of two components:

$$C(m,n) = crit * T + (1 - crit) * R \tag{2.4}$$

Where T is the delay required to reach node n from node m and R is the cost of resources used for that routing path. The timing criticality, *crit*, of each source-sink connection represents how close the delay of that combinational path is to the critical path delay, which is the longest-delay path of the circuit (and restricts the overall operating frequency). The timing criticality of all signals on a combinational path is defined as:

$$crit = \frac{T_{path}}{T_{critical\ path}} \tag{2.5}$$

Criticality takes on values between 0 and 1; timing-critical signals have a *crit* value close to 1, while lower-criticality signals have a *crit* value closer to 0 and can afford to take an indirect path if the resources on the straight-forward path need to be freed up.

The negotiated-congestion routing approach [47] allows multiple signals to use the same routing resource, and this congestion is gradually resolved over many routing iterations. Each consecutive routing iteration re-routes all signals, but increases the cost of routing resources based on their congestion history. VPR defines the resource cost of node n as in [7]:

$$R(n) = b(n)h(n)p(n) \tag{2.6}$$

In the above equation $b(n)$ is the base cost of node n and is usually set to the average delay required to travel the length of one logic block using the inter-LB wires. $p(n)$ is the present congestion and corresponds to the number of signals using node n. The historical congestion term $h(n)$ represents the congestion of node n since the first routing iteration. The resource cost R therefore grows based on the node's past and present congestion and, after a few iterations, the cost of congested resources becomes acceptable only for timing-critical signals, with *crit* near 1. While all congestion is ideally resolved after multiple routing iterations, it is possible for some routing resources to remain contested by two or more signals regardless of the number of iterations performed; in this case the circuit is unroutable with the given FPGA architecture.

Router Lookahead

In Equation 2.3 the cost of routing from an intermediate node n to the target node t, $C(n,t)$ is not known exactly until the route is completed, and must therefore be estimated while the route is still in progress.

VPR makes assumptions about the FPGA geometry and the inter-LB interconnect to quickly lower-bound the remaining cost. If the router encounters a given type of wire (for example, a slow length-4 wire), the VPR lookahead estimates the delay and resource cost to target by assuming that the smallest possible number of wires of the same type (slow length-4 wires) can be used to reach the target via the shortest Manhattan distance.

Notably, the VPR lookahead assumes that if different wire types are present, they are disconnected from each other and wires connect only within their own type and to FPGA blocks. For complex interconnect topologies this strategy is not sufficient: a slow length-4 wire may connect to a fast length-16 wire downstream, and correctly estimating the remaining delays can improve the quality of results. We revisit this topic in Section 3.2.4.

2.4 Abstractions, Metrics and Algorithms for Early-Stage Routing Architecture Design

FPGA architects are required to make decisions early in the design cycle based on limited information. Existing CAD tools may not be well-adapted to the architectures being evaluated, and the size of the design space may simply prohibit evaluation using the full synthesize/pack/place/route CAD flow. Design abstractions, metrics and algorithms that complement existing tools are therefore key for giving architects insight into the main factors that affect performance.

2.4.1 Abstractions and Stochastic Models

As discussed in Section 2.2, [20] proposed the concept of 'flexibility' for the FPGA connection block and switch block, which specified the number of programmable switches for these routing elements without considering the precise switch patterns. The concept of flexibility provided architects with an abstraction for designing the routing architecture. However, the differences between the subset, universal and Wilton switch blocks (discussed in Section 2.2.3) would later highlight the impact that the detailed switch pattern has on routability.

[49] developed stochastic techniques to evaluate the routing demands for master-slice ASICs (an ASIC precursor to FPGAs where arrays of gates or basic transistors were custom-connected with metal layers late in the fabrication process). A key result approximates the channel demand as:

$$W \approx \frac{\lambda \bar{L}}{2} \tag{2.7}$$

Where $\bar{L}$ is the average connection length between sources and sinks and the number of connections starting at a logic block is Poisson-distributed with an average of λ. While this result gives intuition as to how the channel width might vary as a function of connection requirements, it does not take into account FPGA wire segmentation or programmable switch patterns, as master-slice ASICs had no analogous restrictions.

In [50] the channel demand predicted by Equation 2.7 was used to stochastically model FPGA routability using length-1 bidirectional wires as a function of high-level parameters like F_c and F_s. While the predicted routabilities matched well with CAD tool experiments, the model assumed length-1 wire segmentation and did not take into account switch block or connection block patterns.

[51] used a combination of prior stochastic models and intuition to develop a simple model of FPGA routability, with many model parameters selected through curve-fitting to real benchmark data. While this empirical approach provided intuition, it did not model switch patterns or mixes of wire types. Furthermore, models that are based on curve-fitting to a given architecture and benchmarks carry an inherent risk that they may not be good predictors for different architecture types.

2.4.2 Algorithms and Metrics for Early-Stage Explorations

The full CAD flow of tools like VPR is often tailored for a specific kind of FPGA architecture, and may be limiting when exploring new architecture spaces. For example, the VPR 7 router lookahead makes simplifying assumptions about the connectivity of different wire types in the routing resource graph (see Section 2.3.4), and while very efficient, the lookahead has limited utility when the simplifying assumptions don't apply, such as with architectures that use many wire types connected in complex ways. In general, a judicious choice of CAD algorithms is necessary for exploring new architecture types. This section discusses algorithms and metrics that have been proposed for early-stage FPGA architecture evaluation.

Placement and Routing

The 'Independence' placement and routing algorithms proposed in [52] and [53] can be used for early-stage design explorations without making any assumptions about the underlying FPGA architecture. In [52] the authors proposed an annealing placement method where placement quality is evaluated at each iteration of the anneal by partially (and sometimes fully) re-routing the placed netlist. In [53] the authors proposed an architecture-adaptive A* router. The router lookahead uses K-means clustering to group routing nodes which have a similar delay to reach a small number of sinks in the routing graph. Sample routing from the clustered "supernodes" to a subset of graph sinks is then used to build lookup tables that are accessed during routing to estimate the remaining delay (all nodes in a supernode share a table entry). While this lookahead algorithm is very general, the lookup tables constructed can have a prohibitively large memory footprint, especially for the high graph node count of large FPGAs (multiple gigabytes for a 200x200-tile FPGA).

While these placement and routing algorithms are architecture-independent, they have very long run time requirements, restricting the size of the design space that can be considered during early-stage routing architecture exploration. In Section 2.3.4, we present a router lookahead algorithm that has a significantly smaller memory footprint compared to Independence and is more general than VPR 7.

Accelerated/Automated Routing Architecture Evaluation

Conventional exploration of interconnect generally involves a sweep of parameters that the FPGA architect considers important. The runtime required to perform a full synthesize/pack/place/route flow over a set of benchmark circuits can restrict the size of the design space that is evaluated, and techniques that either accelerate or intelligently automate exploration of the design space can be very helpful.

Das and Wilton [54] combine the channel demand model developed in [49] and [50] and the average/maximum connection length data from placed benchmark circuits with an algorithmic enumeration of minimal cutsets (see Section 2.5.3) to place an upper bound on the routability of each circuit. Each net is considered independently of all the others and the minimal cutset enumeration is performed on a graph where edges represent channel segments and nodes represent switch/connection blocks. The nets do not have a definite source/sink, but rather consecutive cutsets are enumerated up to a given length and the reliability of the net is a weighted sum over all the lengths the net might take. As in [49] and [50], this approach does not consider the detailed FPGA switch pattern or wire segmentation, and would be difficult to extend to architectures that do not reflect the underlying stochastic models of those works.

TORCH, proposed in [55], uses an annealing-based algorithm to optimize wire segmentation and switch block pattern. During each iteration of the anneal, TORCH either changes the switch block or adds/modifies a wire type. Cost is computed using benchmark circuits and VPR, and combines critical path delay with an estimate of power-usage. Although the runtime of TORCH is very significant, the authors showed that only 200 anneal iterations are required to arrive at a reasonably optimized architecture if clever move/swap strategies are employed. While the results show significant delay and power improvements, the baseline architecture against which the results are compared was unfortunately questionable, using a subset switch pattern and wire segments with extra buffering in the middle for length-3 and length-6 wires.

Methods like [56] and [57] have also been proposed to investigate routability and wiring demand, but do not directly relate routability to FPGA architecture parameters.

In Chapter 5 we investigate techniques to explore FPGA routing architectures without using benchmark circuits or a time-intensive CAD flow.

2.4.3 Crossbar Design

Crossbar structures play an important role in applications that seek to efficiently connect one set of network points to another. FPGA interconnect architectures consist of many (partial or full) crossbars in cascade, as one can think of each switch block, connection block and local cluster routing as a form of (usually depopulated) crossbar. This section discusses crossbar metrics and design algorithms, and their possible application to the FPGA interconnect fabric as a whole.

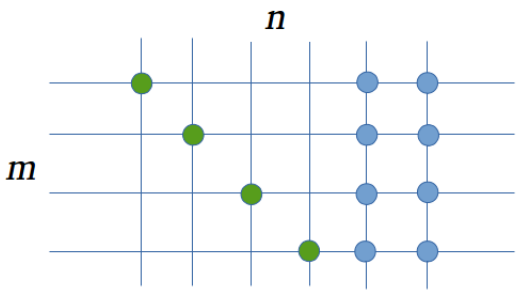

Figure 2.13: A fat-and-slim minimal full-capacity crossbar. Any four terminals in n can always be connected to m.

The capacity of a crossbar is a key measure of its performance and refers to the number of connections that can always be routed from the crossbar inputs to the crossbar outputs, regardless of which inputs are used. For example, Figure 2.13 illustrates a *full-capacity* crossbar which can connect any four of the n input terminals to the four output terminals. Full-capacity crossbars can utilize the maximum possible number of crossbar terminals, but do not guarantee that specific inputs will connect to specific outputs. Crossbars that are not full-capacity [58] are referred to as *sparse*.

Lemieux, Leventis and Lewis [59] presented an algorithm for designing sparse crossbars to optimize the area and routability of crossbar structures inside a logic block [25]. The algorithm focused on optimizing the *average* capacity of a crossbar (rather than creating structures with a specific guaranteed capacity) and relied on Hall's theorem for intuition. Hall's theorem states that, in a bipartite graph, all inputs N can be connected to some outputs M, $|N| \leq |M|$, if and only if

$$\forall S \subseteq N, \quad |S| \leq |NB(S)| \tag{2.8}$$

where $NB(S)$ are the neighbours of $S \subseteq N$ – the subset of terminals in M to which S can connect. This theorem implies that the amount of overlap between the switch patterns of the different crossbar inputs should be minimized in order to increase the likelihood that every subset of N can connect to terminals in M. [59] randomly swaps crossbar connections and accepts the swap result if the following cost function is decreased:

$$\sum_{N_i, N_j \ | \ i \neq j} \frac{1}{HammingDistance(bv_i, bv_j)} \tag{2.9}$$

Where N_i and N_j represent specific input terminals in N. bv_i and bv_j represent the bit vectors of those terminals' connections to the outputs, as illustrated in Figure 2.14. The Hamming Distance counts the number of bit vector positions that have different values; in other words it computes $XOR(bv_i, bv_j)$ and counts the number of 1s in the result.

Once the function of equation 2.9 has been minimized, crossbar routability is evaluated by routing many input test vectors of size k using a network flow algorithm. Each input test vector represents k different input terminals, and the network flow algorithm evaluates the number of these inputs that can be successfully routed to any subset of the outputs. Average routability for a certain value of k is then

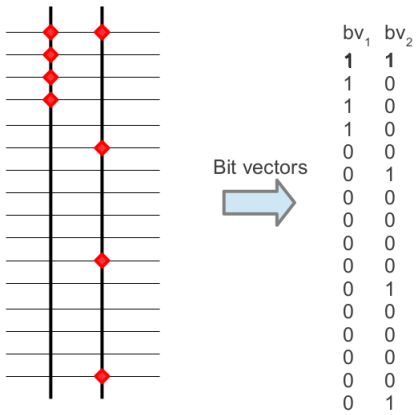

Figure 2.14: Bit vectors represent connectivity of crossbar terminals.

the percentage of input test vectors that can be connected successfully. Crossbars optimized with this approach showed a better routability than those hand-designed by a human architect, and [25] used such optimized patterns to reduce the area required by internal logic block routing (which formerly used full crossbars). However, [59] did not correlate the routability metric of Equation 2.9 to actual architecture routability, as evaluated by a full CAD flow over multiple benchmark circuits.

We considered extending the crossbar routability evaluation method proposed in [59] to evaluate the FPGA interconnect fabric as a whole. For example, it could be possible to estimate the capacity of the routing fabric by setting up sources and sinks throughout the FPGA and evaluating the fraction of routable test vectors using a network flow algorithm. However, the notion of routing capacity may not correlate well to actual routability as measured by a full CAD flow, which requires that specific block outputs be connected to specific block inputs. As well, considerations from the field of network reliability suggest that estimates of capacity represent a very small amount of information about the overall interconnect fabric (Section 2.5.4). In Chapter 5, we discuss an approach that uses efficient path enumeration through the routing resource graph to evaluate the routability of FPGA interconnect with a high degree of fidelity.

2.5 Network Reliability

Later in Chapter 5 we present methods to assign congestion probabilities to the routing resources of the FPGA, and use the congestion profiles to estimate the routability of FPGA interconnect. Network reliability investigates methods of evaluating probabilistic graphs $G(V, E)$ where nodes V and/or edges E have some probability of 'failure' (because the underlying component may fail, may be congested, etc). Network reliability assumes that probabilities placed on nodes/edges are independent of each other and seeks to evaluate whether two or more graph nodes can connect reliably. Three standard reliability measures are defined for directed graphs [60]:

- *s,t-connectedness*: the probability that a connection can be made from source node s to target node t
- *reachability*: the probability that a given source node s can connect to every other node in the graph.

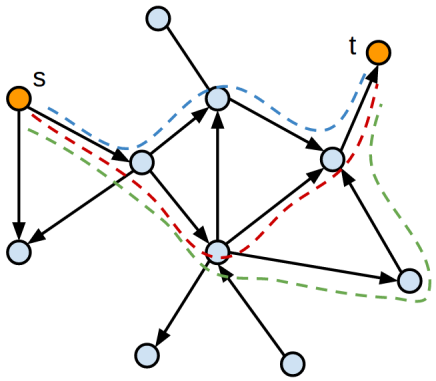

Figure 2.15: Network reliability evaluates the probability of successfully connecting two or more nodes by looking at the possible paths in a probabilistic graph.

- *s,T-connectedness*: the probability that a source node s can connect to all nodes $v \in T$ where $T \subset V$.

Definitions of network reliability measures for undirected graphs are analogous.

An example of *s,t-connectedness* is shown in Figure 2.15 and is clearly a function of both the number of paths between the source and target, and the probabilities of failure placed on the graph nodes/edges.

2.5.1 The FPGA Network Model

The inter-LB interconnect of FPGAs using unidirectional routing can be represented as a graph $G(V, E)$ with nodes V representing routing resources such as wire segments, and edges E representing programmable switch connections [7].

While each edge, $e_m, m \in 1..|E|$, is perfectly reliable, nodes, $v_n, n \in 1..|V|$, have a probability of 'failure'. FPGA signals contend for wire segments and pins (represented by nodes in a routing resource graph), and this congestion translates to some probability that a node will be unavailable for routing. $p_n = P(v_n \; operates)$ represents the probability that node v_n is available, and $q_n = 1 - p_n$ represents the probability that node v_n is congested.

2.5.2 Complexity of Network Reliability

Valiant [61] defined the class of problems *#P* and *#P-complete* and showed that all important network reliability measures are *#P-complete* to compute exactly. Whereas *NP-complete* is a class of decision problems, *#P-complete* is a class of counting problems. For example, calculating *s,T-connectedness* in an arbitrary graph requires knowledge of all shortest paths which connect the source s to the set of nodes T; but such shortest paths are shortest Steiner trees, finding which is *NP-complete*.

2.5.3 Exact Computation of Network Reliability

While exact methods of computing network reliability are exponential in complexity, all estimates of network reliability measures are an approximation to these methods in some way.

State Enumeration

A node v_n can either fail or operate, and the state of a node is therefore represented by $s(v_n) \in 0, 1$. With $|V|$ nodes in total, the number of possible states the graph can ever take is $2^{|V|}$ and the probability of a particular state is

$$\prod_{i \in operational} p_i * \prod_{j \in failed} q_j \tag{2.10}$$

And with each state representing an operational state or a failed state, the overall probability of connecting some subset of nodes is

$$Reliability = \sum_{state \; \in \; operational \; states} P(state) \tag{2.11}$$

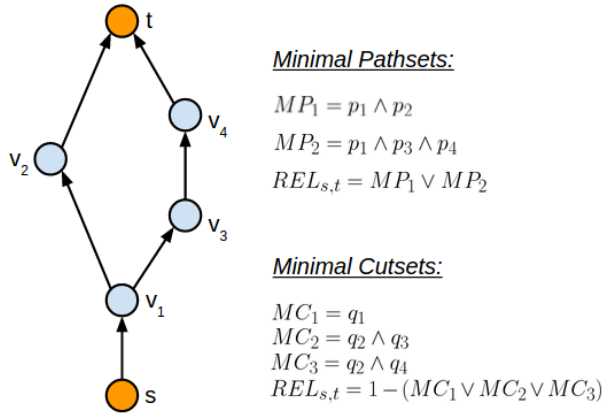

Figure 2.16: Network reliability using minimal pathsets and minimal cutsets.

Minimal Pathset Enumeration

A *pathset* is a set of nodes $S \subseteq V$ that, when operational, make graph G operational (in the sense of being able to connect a source node s to some destination nodes). A *minimal pathset* is a set of nodes $S' \subseteq V$ from which no node may be removed and still keep S' a pathset (in other words, a pathset with no superfluous nodes) [60].

Figure 2.16 shows an example of *s,t-connectedness* evaluated using minimal pathset enumeration – logical expressions are created for each path, and the overall network reliability, $REL_{s,t}$, is the probability that any one of the s, t paths is in an operational state. It is clear that paths are not independent of each other, making it necessary to store information about each path for an exact evaluation of reliability.

In Chapter 5, we make an assumption of independence between different paths and present methods to efficiently enumerate and evaluate s, t paths in order to estimate the routability of an FPGA routing architecture without benchmarks.

Minimal Cutset Enumeration

Network reliability may also be viewed in terms of cuts. A *cutset* is a set of nodes $C \subseteq V$ which, when in a failed state, will cause the entire graph to fail. A *minimal cutset* is a set of nodes $C' \subseteq V$ from which no node may be removed and still keep C' a cutset (i.e. a cutset with no superfluous nodes) [60].

An example of evaluating network reliability with minimal cutsets is shown in Figure 2.16. Analogous to minimal pathsets, the graph G will fail when the nodes in any minimal cutset fail and the graph in Figure 2.16 has size-1 and size-2 minimal cutsets. Multiple cutsets can share the same nodes, making it necessary to store information about each minimal cut for an exact reliability evaluation.

The problem of enumerating minimal cutsets of any size is *#P-Complete*, but works like [62] have presented methods to efficiently upper-bound the reliability of networks by enumerating a subset of minimal cutsets.

2.5.4 The Reliability Polynomial

The reliability polynomial, presented in [63], gives an efficient way to view the reliability information of a graph. Recall that network reliability may be evaluated by exhaustively enumerating every state a

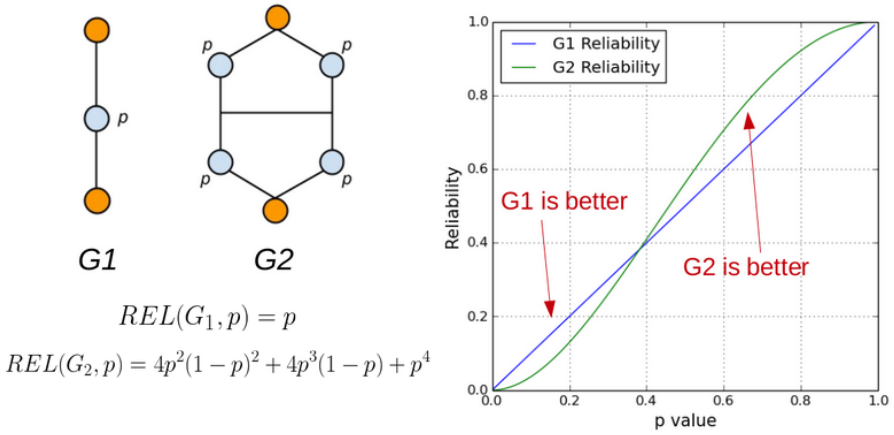

Figure 2.17: The reliabilities of the two networks are represented using the reliability polynomial. The plot of reliability shows that the networks have different relative performance depending on the probability, p, that the network nodes operate.

graph can take. If every node has the same probability of operation, p, then a state with i operational nodes will be seen with a probability of

$$p^i(1-p)^{|V|-i} \tag{2.12}$$

If there are N_i states with i operational nodes (due to which the whole network is operational), then the reliability of the network can be written as

$$REL(G, p) = \sum_{i=0}^{|V|} N_i \cdot p^i(1-p)^{|V|-i} \tag{2.13}$$

Equation 2.13 is the *reliability polynomial* and the set of coefficients $N_0 ... N_{|V|}$ hold the reliability structure of the graph. While the complexity of evaluating the values of N_i is exponential [60], it is simple to recompute network reliability for different values of p once each N_i is known.

The reliability polynomial gives interesting insights. For example, Figure 2.17 shows the reliability plots of two graphs. Plotting the reliabilities of connecting the top and bottom nodes for different values of p shows that the performance of a network has a strong dependence on the level of stress placed on it. When the networks are less stressed (higher p), G_2 is more reliable, but in high-stress situations (smaller p) G_1 performs better. This shows that the effect of short paths dominates when networks are congested, and that in low-congestion situations the overall number of paths connecting the source and the sink are more important. The reliability plot of Figure 2.17 also suggests that comparisons of routing networks should be done with a level of stress at which these networks are expected to operate.

The reliability polynomial can also give insight into the kinds of algorithms that are likely to work for evaluating network performance. With C_i representing the number of network states with cutsets of size i, the reliability polynomial can be written as

$$REL(G, p) = 1 - \sum_{i=0}^{|V|} C_i \cdot (1-p)^i p^{|V|-i} \tag{2.14}$$

Earlier in Section 2.4.3, we discussed the application of [59] to measuring FPGA routing architecture

capacity using a network flow algorithm over different input test vectors. The capacity of the interconnect measured in this way is determined by the size of the smallest cutset of the graph (for some input test vector). So out of all the information held by the network, this capacity-measurement approach would only give the subscript of the first non-zero C_i, i.e. the size of the smallest minimal cutset. Later in Chapter 5 we present efficient methods to estimate the reliability of FPGA routing networks with greater fidelity.

Chapter 3

VPR and Architecture Enhancements

In subsequent chapters we consider routing topologies that use multiple wire types connected in complex ways. For accurate results, it is important both to use a recent technology node and to enhance VPR's architecture description language and CAD algorithms in order to be able to describe and evaluate these topologies.

3.1 22nm Architectures

The 22nm FPGA architecture on which we build our interconnect explorations is based on the flagship VPR 7.0 [37] 40nm architecture and uses a logic block similar to that of popular commercial devices [6] [64]. The logic block is composed of ten 6-input LUTs which are fracturable into two 5-input LUTs, so long as they use no more than 6 inputs. Similar to Figure 2.5, a fracturable 6-LUT together with a pair of registers form a BLE, and a full input crossbar is present to connect any of the 40 logic block inputs, or any BLE output, to any BLE input (but there is no output crossbar or carry chains). We also check our main results using a logic block architecture similar to [65] which has eight 4-input LUTs and no internal crossbars. These architectures are summarized in Table 3.1.

To have timing and area parameters representative of the 22nm node, the transistor-sizing tool

Table 3.1: Logic architecture for most of our interconnect explorations.

	Architectures Using 6-LUTs	*Architectures using 4-LUTs*
LB Basic Parameters	($K = 6$, $N = 10$, $I = 40$, $O = 20$)	($K = 4$, $N = 8$, $I = 32$, $O = 8$)
LB Input Crossbar	Full Crossbar	None
LB Output Crossbar	None	None
LB Internal Feedback	Through Input Crossbar	None
DSP Elements	36x36 Fracturable Multipliers (36x36, two 18x18, four 9x9)	36x36 Fracturable Multipliers (36x36, two 18x18, four 9x9)
Memories	32Kb Block RAMs (512 x 64 to 32K x 1)	16Kb Block RAMs (512 x 36 to 16K x 1)
$F_{c,in}$	0.1	0.2
$F_{c,out}$	0.1	0.2
W	300	200

Table 3.2: Metal stack data based on the 2014 entry of the ITRS 2011 interconnect report.

Metal Layer	Half-Pitch (nm)	Aspect Ratio	R (ohm/um)	C (fF/um)
Intermediate	24	1.9	54.825	0.175
Semi-Global	48	2.12	7.862	0.215
Global	96	2.34	1.131	0.250

Table 3.3: Extracted 22nm delay and area data for the semi-global and global metal layers across different wire segment lengths. Each entry shows data for the (*semi-global / global*) metal layers. VPR and COFFE measure area in Minimum-Width Transistor Areas (MWTAs).

	Length 1	**Length 2**	**Length 4**	**Length 8**	**Length 16**
Routing Switch Output Resistance (Ω)	1110 / 840	740 / 500	520 / 300	500 / 230	350 / 200
Routing Switch Intrinsic Delay (*ps*)	57 / 50	64 / 52	80 / 60	132 / 83	235 / 107
Routing Mux Pass Gate Area (MWTAs)	1.5 / 1.5	1.5 / 1.7	1.7 / 1.7	1.5 / 1.7	1.5 / 2.0
Routing Buffer Area (MWTAs)	11 / 13.6	15 / 19	23 / 27	25 / 39	38 / 50
Wire Resistance Per Tile (Ω)	232 / 34				

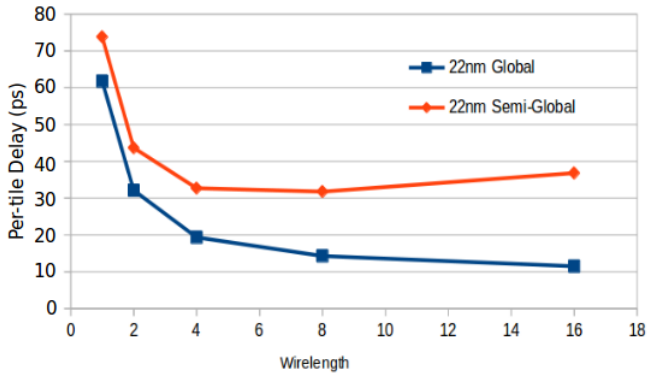

Figure 3.1: Delay per tile versus wire length on the semi-global and global layers.

COFFE [66] was used to size a number of architectures over different wire segment lengths, metal layers and routing flexibilities. We extracted the metal stack data for the 22nm process node using the 2014 entry of the 2011 ITRS Interconnect Report [67]. The metal stack data is shown in Table 3.2

By default COFFE uses the intermediate metal layer to implement the wiring inside the logic block and the semi-global layer to implement the general routing outside of it. In Chapter 4, we explore wire length and topology choices for inter-LB interconnect on the global metal layer, and we used COFFE to extract delay and area parameters for each combination of {*semi-global, global*} metal layer and length-{*1, 2, 4, 8, 16*} general routing wires at a channel width of $W = 300$ (which aligns well with popular commercial FPGA devices [22]). For each combination of metal layer and wire segment length, we varied the connection block and switch block flexibilities to capture the delay impact of different routing mux sizes. Area and delay parameters extracted from COFFE were then used for VPR architecture files. Representative parameters for the different wire types are shown in Table 3.3. Longer wires and wires

on the global metal layer present a larger capacitive load and so require bigger, less resistive routing switches[1]. The last row of Table 3.3 shows that wires on the global metal layer have significantly lower resistance per tile, where the dimensions of a tile were reported by COFFE as $30\mu m$ by $30\mu m$.

Figure 3.1 contrasts the per-tile delays between the semi-global and global metal layers for multiple wire lengths. Per-tile delay refers to the delay through a loaded wire segment (and the associated driver) divided by the number of tiles the segment spans. Figure 3.1 thus shows that global-layer length-16 segments are approximately twice as fast as global-layer length-4 segments when travelling the same (long) distance, while on the semi-global layer length-16 wires yield no delay benefit over length-4's or length-8's. The increasing wire delay of newer technology nodes clearly shows its impact – in contrast with some notable earlier routing architecture studies [7], semi-global length-8 and length-16 wires are now too resistive to provide a delay benefit over length-4's without decreasing the capacitive loading on the wires (for example by removing SB or CB connections).

3.2 VPR Enhancements

3.2.1 Enhanced Switch Block Descriptions

VPR 7.0 can specify three kinds of switch blocks in the architecture file – subset, universal and Wilton. While these switch patterns have a significant impact on routability for bidirectional topologies, they do not have a clear analogue for unidirectional wires which can only be driven at their start point. VPR adapts the traditional switch patterns to unidirectional topologies by snapping switch block connections to the nearest wire start point to legalize the pattern, and it is ambiguous what kind of switch block this produces. More importantly, the switch blocks used by VPR do not allow us to specify some of the complex interconnect topologies that we wish to explore. For example in Section 4.3 we explore a routing architecture with "regular" and "fast" wires. The regular wires are driven by both regular and fast wires, but a signal can find its way onto a fast wire only through the output connection block or another fast wire. The switch block patterns in VPR 7 would not be able to implement this kind of interconnect hierarchy as they make no distinction between wire types, and would connect regular and fast wires in a semi-random manner. To overcome this limitation we have created the new switch block specification method outlined below and incorporated it into a new version of VPR.

While the complete switch block format is described in the latest VTR documentation [68], a few key points are discussed here. The format is an extension of the permutation function tables described in [35] and discussed in Section 2.2.3. We classify the different switch blocks along a wire segment's length into switch points as shown in Figure 3.2 – the switch point at the start of the wire segment is given an index of 0 which is incremented by 1 until the segment end point, which also receives an

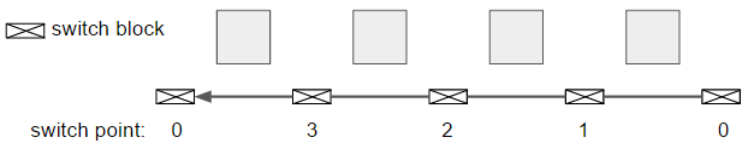

Figure 3.2: Diagram of switch points for a length-4 unidirectional wire that's driven from right to left.

[1]The routing switch intrinsic delay was modelled to include the wire resistance and capacitance (but not connection block & switch block loading). This still allows us to accurately model the effects of capacitive loading from additional switches, but simplifies parameter extraction from COFFE HSPICE files.

index of 0 (the segment endpoint and the startpoint of the next segment share the same switch block). The switch block format allows a user to specify mathematical permutation functions that describe how different wire types (defined in the architecture file) will connect to each other at different switch points.

An example of the switch block format is shown in Figure 3.3. The XML code specifies a pattern where length-2 wires on the semi-global layer (identified by their name "L2") and length-4 wires on the global layer ("L4global") are connected in a hierarchy where the global length-4 wires can jump down

```
<switchblocklist>
   <!-- Define connections between adjacent SB sides -->
   <switchblock name="wilton_turn" type="unidir">
      <switchblock_location type="EVERYWHERE"/>
      <switchfuncs>
         <func type="lt" formula="W-t"/>        <!-- Left-to-top -->
         <func type="lb" formula="t-1"/>        <!-- Left-to-bottom -->
         <func type="rt" formula="t-1"/>        <!-- Right-to-top -->
         <func type="rb" formula="W-t-2"/>      <!-- Right-to-bottom -->
         <func type="br" formula="W-t-2"/>      <!-- Bottom-to-right -->
         <func type="bl" formula="t+1"/>        <!-- Bottom-to-left -->
         <func type="tl" formula="W-t"/>        <!-- Top-to-left -->
         <func type="tr" formula="t+1"/>        <!-- Top-to-right -->
      </switchfuncs>
      <wireconn from_type="L2" to_type="L2" from_switchpoint="0,1" to_switchpoint="0"/>
      <wireconn from_type="L4global" to_type="L4global" from_switchpoint="0" to_switchpoint="0"/>
      <wireconn from_type="L4global" to_type="L2" from_switchpoint="0" to_switchpoint="0"/>
   </switchblock>

   <!-- Define connections between opposite SB sides -->
   <switchblock name="wilton_straight" type="unidir">
      <switchblock_location type="EVERYWHERE"/>
      <switchfuncs>
         <func type="lr" formula="t"/>          <!-- Left-to-right -->
         <func type="rl" formula="t"/>          <!-- Right-to-left -->
         <func type="bt" formula="t"/>          <!-- Bottom-to-top -->
         <func type="tb" formula="t"/>          <!-- Top-to-bottom -->
      </switchfuncs>
      <wireconn from_type="L2" to_type="L2" from_switchpoint="0" to_switchpoint="0"/>
      <wireconn from_type="L4global" to_type="L4global" from_switchpoint="0" to_switchpoint="0"/>
      <wireconn from_type="L4global" to_type="L2" from_switchpoint="0" to_switchpoint="0"/>
   </switchblock>
</switchblocklist>
```

Figure 3.3: A description of a switch block in a VPR architecture file that defines connections between semi-global length-2 and global length-4 wire types. The wires connect within their own type with a flexibility of $F_s = 3$, and the global wires connect to the semi-global wires with $F_s = 3$.

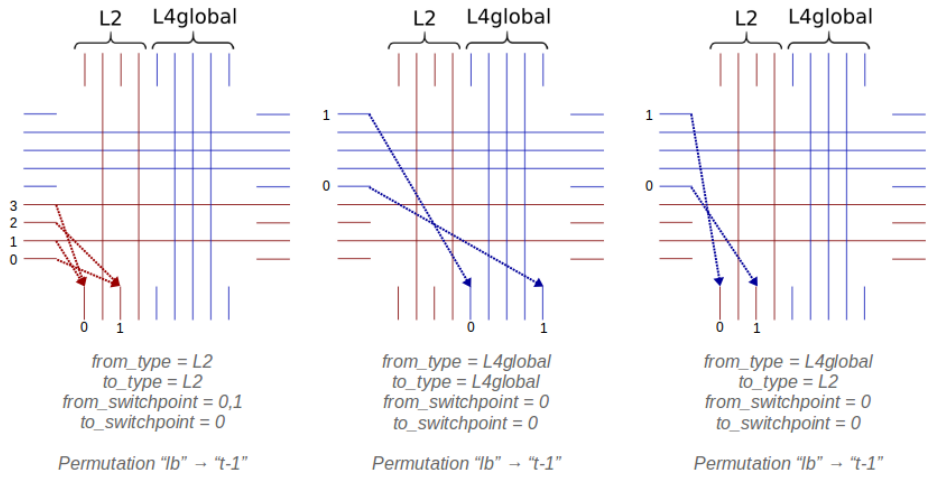

Figure 3.4: An example of the switch block patterns produced by the *left-to-bottom* permutation function specified in Figure 3.3 for all three *wireconn* entries. Note that *from_type* and *from_switchpoint* define the set of source tracks, and the source track numbering is relative to this set (destination tracks are numbered similarly).

to the regular (semi-global) routing. The entries under *switchfuncs* define the permutation functions along different directions (i.e. left-to-right, bottom-to-left, etc) while the *wireconn* entries specify the source/destination wire types and switch points. Note that the permutation functions also implicitly control the switch block flexibility, F_s – by specifying zero or more permutation functions for each direction (e.g. for "*lt*" or "*br*"), switch block flexibility can be controlled for each wire type on a per-side basis. The final switch block built by VPR is a union of all the descriptions specified under the *switchblocklist* entry.

Figure 3.4 shows some of the switch block patterns created by this specification for the *left-to-bottom* permutation function. Note that each *wireconn* entry specifies a set of source and destination wires, which in turn defines the track indices being connected. Also note that the permutation functions are implicitly modulo W, which is redefined in Figure 3.3 to refer to the number of tracks in the destination track set.

Our new switch block specification format provides a flexible and concise way to describe virtually any switch block connecting any mix of wire types, including the traditional switch blocks supported by VPR 7.

3.2.2 Enhanced Connection Block Descriptions

We have enhanced VPR's connection block format to allow varying connection block flexibility based on the type of wire to which a pin connects. For example, this enhancement allows us to describe architectures where fast wires on the global metal layer can be driven from a connection block with a higher flexibility than the rest of the routing but have no direct connection to input pins, forcing routed paths to reach inputs through regular wires on the semi-global layer instead.

3.2.3 Variable Routing Switch Delays Based on Fan-In

While VPR 7 accounts for area differences due to a variable number of mux inputs, it does not adjust mux delay which also grows with fan-in. We enhanced the VPR architecture specification format to allow a list of *(fanin, delay)* pairs to describe the delay behaviour of each specified routing switch as shown in Figure 3.5. This enhancement is necessary to accurately model the delay behaviour of complex interconnect topologies where multiple interconnected wire types commonly result in a range of mux input sizes. During the generation of the VPR routing resource graph, interpolation (or extrapolation) in this delay table is used to choose the delay of each multiplexer, as only at that point is the precise fan-in of each mux known. As mentioned in Section 3.1, COFFE was used to extract timing parameters for switches driving each kind of wire type over a range of fan-in values.

```
<switch type="mux" name="my_mux" R="521.95" Cin="3.134e-15" Cout="0" mux_trans_size="1.74" buf_size="22.800000">
   <Tdel num_inputs="12" delay="8.02e-11"/>
   <Tdel num_inputs="15" delay="8.35e-11"/>
   <Tdel num_inputs="20" delay="9.47e-11"/>
   <Tdel num_inputs="25" delay="9.67e-11"/>
</switch>
```

Figure 3.5: VPR adjusts mux delay during RR graph generation based on mux fan-in and the list of (fan-in, delay) pairs, shown here, specified in the architecture file.

```
1: function GENERATE_LOOKAHEAD(routing resource graph)
2:     REF_X=3, REF_Y=3, M = 10
3:     lookahead_map = NULL
4:
5:     For each wire_type in the VPR architecture file{
6:       For chan_type in {x-chan, y-chan}{
7:         For M wires at REF_X, REF_Y{
8:           n = get_routing_resource_graph_node( wire )
9:           priority_queue = NULL
10:          priority_queue.push(n)
11:
12:          //Run Dijkstra's algorithm sorted on path delay
13:          While priority_queue not empty{
14:            n = get_lowest_delay_node(priority_queue)
15:            if (n is input pin &&
16:            n.x >= REF_X && n.y >= REF_Y){
17:              dX = n.x - REF_X
18:              dY = n.y - REF_Y
19:              old_T = get_path_delay(
20:                lookahead_map[wire_type][chan_type][dX][dY])
21:
22:              // lookahead_map to contain smallest-delay entry
23:              if (old_T == UNDEFINED ||
24:              old_T > n.path_delay){
25:                lookahead_map[wire_type][chan_type][dX][dY] =
26:                  add_entry(n.path_delay, n.base_path_cost)
27:              }
28:            }
29:            add_neighbors_by_path_delay(n, priority_queue)
30:          } //While
31:        } //For M wires
32:      } //For chan_type
33:    } //For each wire_type
34:
35:    return lookahead_map
36: end function
```

Figure 3.6: Algorithm to generate look-up tables to be used by the router lookahead.

3.2.4 Enhanced Router Lookahead

As discussed in Section 2.3.4, the VPR 7 router lookahead assumes that different wire types do not connect to each other; this assumption is invalid for the complex routing architectures we seek to explore and most commercial architectures. An alternative would be to use the Independence routing lookahead algorithm of Section 2.4.2 but that technique, while very general, has a very high memory footprint. We propose a new lookahead that is well-suited to the regularity of island-style FPGA architectures. The new lookahead is more general than the lookahead used in VPR 7.0 and has a very low memory requirement compared to Independence.

Figure 3.6 summarizes the lookahead generation algorithm. This algorithm computes look-up tables by running Dijkstra's algorithm [48] sorted on delay from a small subset of wires belonging to each *wire type* that has been defined in the VPR architecture file (i.e. the algorithm is run from a few slow length-4's, a few fast length-16's, or any other wire types that were defined). The search is performed at one reference coordinate from both x-directed and y-directed channels and notes two pieces of information each time an input pin is encountered above and to the right of the starting tile: the *path delay* and the

Table 3.4: Memory, compute requirements and generality of different router lookaheads for a 200x200 island-style FPGA.

Router Lookahead Algorithm	Memory Footprint	Lookup Table Compute Time	Generality
VPR 7.0	0	0	✓
Independence	> 6GB	Not Listed	✓✓✓
Proposed	< 10MB	30s	✓✓

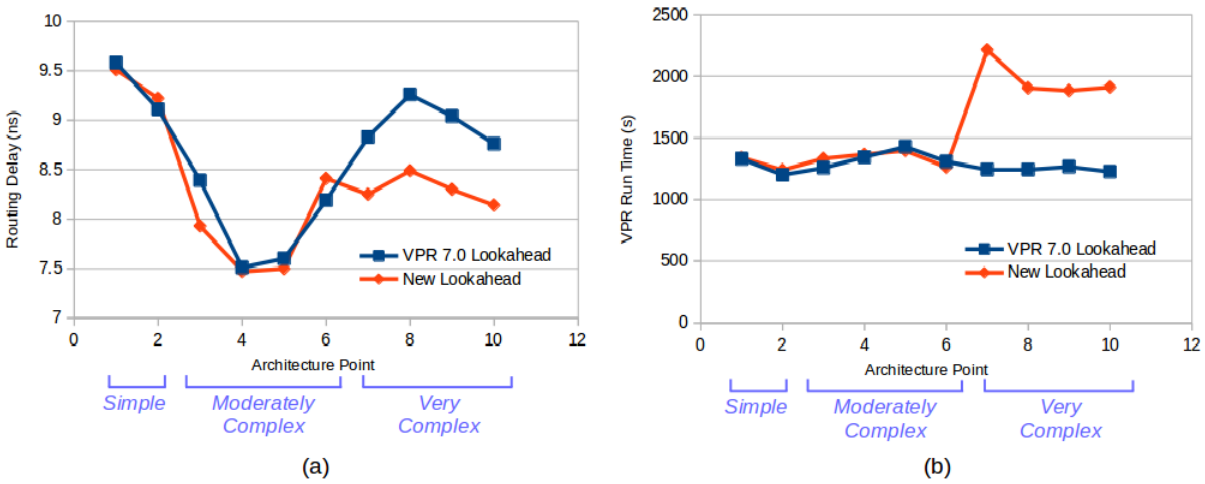

Figure 3.7: Average critical path routing delay (a) and VPR runtime (b) over 9 largest VTR benchmarks, for two lookaheads, over 8 distinct architectures.

base path cost to arrive at that pin – these are the two costs (timing and resource) that VPR needs to estimate during routing. Note that the search is performed away from the edge of the chip to avoid fringe effects, and we assume symmetry by only searching up and to the right of the starting coordinate. This search records the minimum *path delay* and associated *base path cost* required to reach each coordinate relative to the starting tile.

The proposed lookahead takes advantage of the regularity of the island-style FPGA architecture to build small look-up tables that can be rapidly indexed by the router. For example, to estimate the delay to travel 3 tiles in the x direction and 5 tiles in the y direction starting from an x-directed wire of type *len4*, the router would lookup the *base path cost* and *path delay* information under index *[len4][x-chan][3][5]*. Table 3.4 compares the compute and memory requirements of our lookahead to that of VPR 7 and Independence. While not as general as Independence, our lookahead achieves a good trade-off point due to its much more manageable memory footprint. With a computational complexity of $O(Nlog(N))$ (N is the number of nodes in the routing graph), our lookahead scales well to very large chips and adapts to arbitrary wire type mixes and switch patterns, assuming only that the chip is translationally invariant (which island-style FPGAs are).

Figure 3.7 compares our lookahead with VPR 7.0 over two simple, four moderately complex and four very complex architectures. The first two points represent simple architectures with only one type of wire available. The next four architectures (points 3-6) have 85% of wires on the semi-global layer and 15% on the global layer; the global wires can only be driven from block output pins and can only drive semi-global wires. The last four points (points 7-10) represent architectures where the global wires

(which comprise 15% of the channel width) can be driven, and can only drive, a small fraction of the regular wires on the semi-global layer. The proposed lookahead has quality and runtime equivalent to the VPR lookahead for simple and moderately complex architectures (points 1-6), and has significantly better result quality for complex architectures at the expense of route time (points 7-10). Despite the longer runtime for the last four architecture points, our lookahead adapts well to their complex and irregular structure, and we use it for our interconnect explorations in Chapter 4.

Chapter 4

Complex Unidirectional Routing Topologies

For a typical FPGA design, most of the delay and 50% or more of the area [7] [6] is due to programmable interconnect, so its optimization is a priority for FPGA architects. Moreover, as discussed in Section 2.1, interconnect RC delay has increased rapidly with process scaling, particularly for the thin wires at the bottom of the metal stack, and the challenges of poor interconnect resistance scaling require FPGA architects to use the metal layers with greater efficiency. Interconnect architecture is a complex topic, and includes many interacting architectural decisions such as wire electrical characteristics, wire segment length and the switch pattern between wires. While each of these topics has been studied before on its own, e.g. in [31] [8] [10], the interaction between switch patterns and wire types has not been explored in published literature. In this chapter we use the architecture and VPR enhancements discussed in Chapter 3 to explore the impact of unidirectional wiring on switch pattern, as well as suitable interconnect hierarchies to take advantage of fast, scarce wiring on the upper metal layers.

4.1 Methodology

We use VPR along with the VTR benchmark set [37] to evaluate routability and delay; the benchmarks used are summarized in Table 4.1. In Section 4.2 we evaluate routability by finding W_{min}, the minimum routable channel width of each circuit. Finding W_{min} requires a computationally-intensive binary search over many channel widths, so a subset of medium-sized VTR benchmark circuits is used. In Sections 4.2 and 4.3, critical path routing delay (the portion of the critical path through the inter-LB routing) is evaluated over the nine largest VTR benchmarks for a constant channel width of 300, which provides some extra flexibilty over the W_{min} required by our most complex benchmark circuit. We consider timing results in a fixed channel width the most representative as all commercial FPGA families ultimately choose a single channel width, and $W = 300$ aligns well with popular commercial FPGA devices [22] with a 10-BLE fracturable 6-LUT logic block similar to the one we use (see Table 3.1).

In our architecture explorations we have treated the fastest metal layers as a scarce resource – in addition to being used for fast general routing wires, these layers are utilized for power, ground and clocking. Furthermore, the via stack required to connect from the highest metal layers to silicon presents significant layout challenges since vias must traverse through all the intermediate metal layers.

Table 4.1: *Benchmarks for evaluating routing delay. †Benchmarks for evaluating routability.

Circuit	#6-LUTs	Min W (len-4 wires)
mcml*	99700	144
LU32PEEng*	75530	204
bgm*†	30089	168
stereovision2*	29849	172
LU8PEEng*†	21954	136
stereovision0*†	11462	78
stereovision1*†	10366	120
blob_merge*†	6016	100
mkDelayWorker32B*†	5580	110
or1200†	2963	90
boundtop†	2921	72
sha†	2212	64
raygentop†	2134	74
mkSMAdapter4B†	1977	80

Commercial architectures have used 10-20% of the available channel width for fast long wires, and have restricted wires on upper metal layers to have vias approximately every four logic blocks [30] [22].

For our delay experiments, we use architectures with a channel consisting of 300 unidirectional wires where 15% of the channel width can be used for wires on the global metal layer. To reflect the layout difficulties of deep via stacks, wires on the global metal layer are allowed to have connection block and/or switch block connections only once every four tiles.

4.2 Effect of Unidirectional Wires on Switch Pattern

In this section, we look at the best single wirelength and switch block topology to be used for the semi-global metal layer in our subsequent explorations; this is the "workhorse" metal layer for inter-block connectivity. Commercial architectures have typically based their routing around a shorter (approximately length-4) wirelength and a switch block topology that allows signals to have access to a diverse set of tracks during the course of a route; we validate this choice of semi-global wirelength at the 22nm node and make some interesting observations about switch block topology in unidirectional architectures.

In Figure 4.1 we evaluate the routability of the subset [20], universal [33] and Wilton [10] switch blocks implemented with unidirectional wires of varying length on the semi-global layer. As expected, longer wire lengths are generally less routable because they decrease the granularity with which signals can utilize the routing resources.

With regard to switch blocks, at length-1 the Wilton topology is much more routable. While the universal and subset switch blocks restrict a signal to a small subset of possible wires, the Wilton switch block permutes the switch pattern so that signals are able to find their way onto virtually any track in the FPGA[1]. However, at longer wire lengths the routability of the different switch block patterns begins to converge.

[1]The significantly lower routability of the subset and Universal switch blocks at the length-1 point is due in large part to the low connection block flexibility in our architectures (see Table 3.1). Signals from an output pin are confined to a small fraction of available wires in the FPGA due to the restrictive nature of these switch blocks. The wires that an output pin connects to would hence not be guaranteed to line up with the switches connected to the input pins of I/O, RAM or DSP blocks (which do not have an input crossbar in our architecture) causing some nets to be unroutable even without congestion. Increased F_c would likely improve the overall area usage of these patterns, though they would still remain inferior choices to the Wilton switch block at length 1.

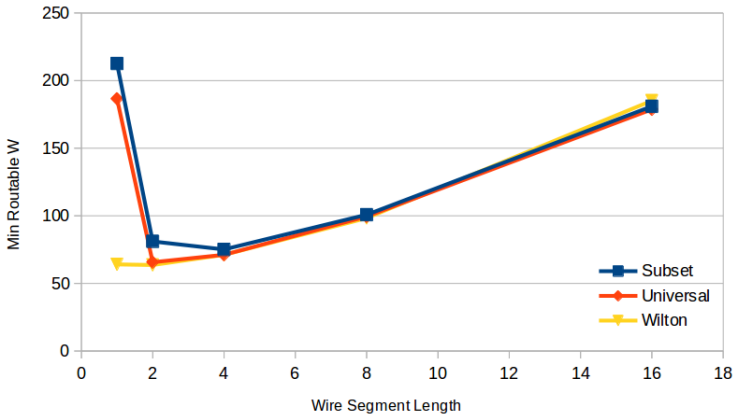

Figure 4.1: Routability of switch block patterns converges for longer unidirectional wires.

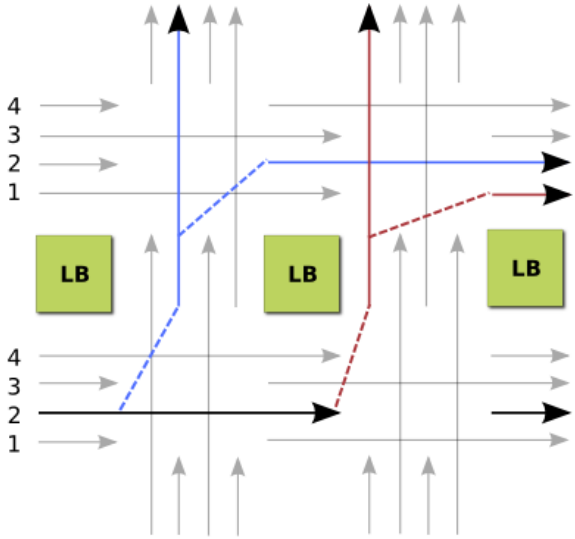

Figure 4.2: Unidirectional routing can increase the diversity of tracks reachable by a signal. A signal starting on track 2 has access to track indices 1 and 2; it is impossible to specify the traditional subset switch pattern that will confine the signal to a single track index.

In unidirectional architectures wires can only be driven at their start point which eliminates some of the area required for bidirectional routing [31]. We observe that the restriction of connecting only to the start points of unidirectional wires permutes the switch block connections beyond what the switch block pattern specifies for segment lengths greater than 1. As an example, consider the length-2 unidirectional architecture in Figure 4.2 and the sample routing shown. While a traditional subset switch pattern would confine a signal on track i to always stay on tracks with indices i, this restriction is not possible in a unidirectional topology with segment lengths greater than 1. Figure 4.2 shows that a signal starting on track 2 can stay on track 2 in the first vertical channel, but is forced onto track 1 in the second vertical channel. Regardless of how we arrange the actual switch block pattern, a signal starting on track 2 will always be able to access at least two different track indices with length-2 unidirectional wires. In general, with length-L unidirectional wires, a signal will have access to at least L track indices, leading to the routability convergence of the traditional switch patterns in Figure 4.1.

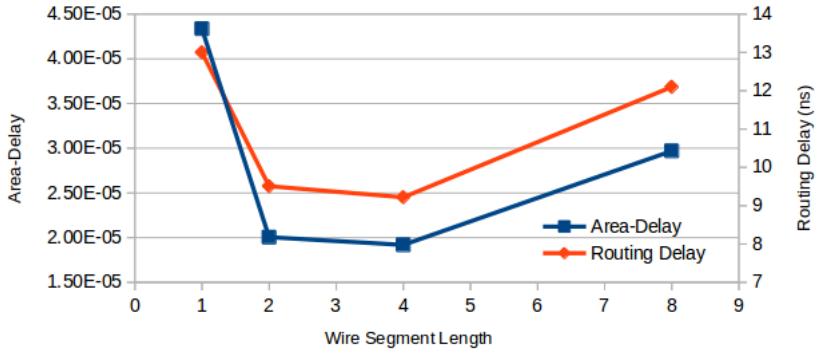

Figure 4.3: Area-delay (blue) and critical path routing delay (red) of architectures with a single type of semi-global wire using the Wilton switch block.

In our multi-wirelength explorations we choose the Wilton switch block but focus on how different routing resources connect instead of the specific switch pattern that implements the connection.

Figure 4.3 verifies that length-4 wires still achieve the best area-delay tradeoff at 22nm, and length-4's form the basis of our complex architecture explorations.

4.3 Complex Routing Topologies

As shown in the previous section, unidirectional wires can add diversity to the tracks that a signal can reach and, at moderate and long wire lengths, analogues of low-routability patterns like the subset are impossible to create. Moving on from detailed switch patterns, in this section we explore high-level hierarchies that connect fast wires on the global metal layer with the rest of the routing. Figure 4.4 illustrates the topologies; the thick arrows represent wire types and the thin arrows represent how each wire type connects to other wires and FPGA blocks. Each connection between wire types is implemented with a Wilton switch pattern adapted to that length of unidirectional wire but, as shown in Section 4.2, other patterns can likely be used without significantly affecting the result. Figures 4.5 and 4.6 illustrate the "*On-CB, Off-CB*" and "*On-SB, Off-SB*" topologies in more detail by showing how they connect two logic blocks in a channel through a global-layer wire segment.

At a high-level, the topologies explored in this section represent three kinds of wire type hierarchies

- *No distinction between wire types.* The *VPR 7 default* switch pattern does not consider separate wire types and connects all wires in the channel using the Wilton switch pattern, with legalization for unidirectional segments. As a result, some segments will connect within their own wire type while others will connect to other wire types, producing a somewhat irregular interconnect where connections between wire types are not well defined.

- *Isolated wire types.* The (a) *On-CB/Off-CB* topology of Figure 4.4 does not implement connections between wire types, and wires connect only within their own type and to FPGA blocks. This topology was also used in [9] to evaluate a mix of regular wires and fast (widely-spaced) wires.

- *Connected wire types.* Topologies (b), (c), (d) and (e) in Figure 4.4 have various forms of guaranteed connections between wire types, and also vary the connectivity between wire types and FPGA blocks.

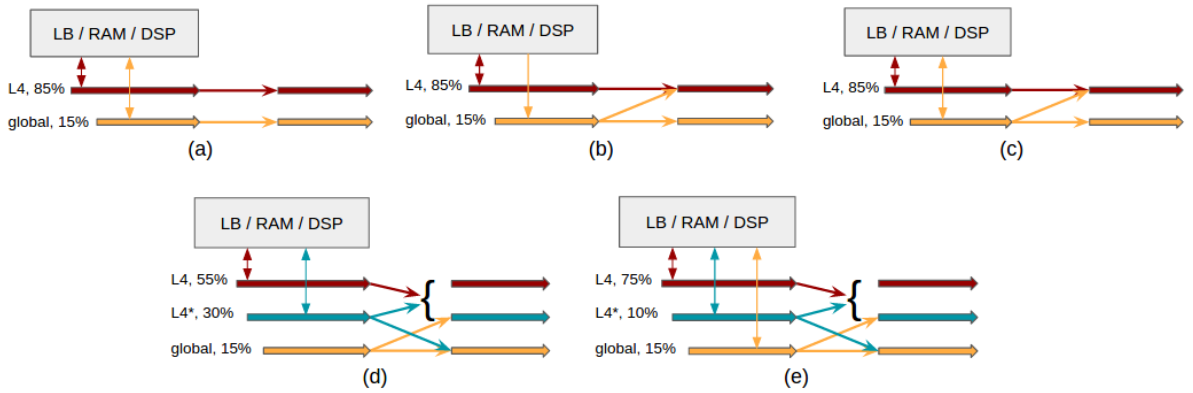

Figure 4.4: Different routing topologies with regular and fast wires. Topology names refer to the connectivity of global wires. (a) On CB, Off CB topology (b) On CB, Off SB topology. (c) On CB, Off CB/SB topology. (d) On SB, Off SB topology; only a fraction of regular L4 wires can drive global wires. (e) On CB/SB, Off CB/SB topology.

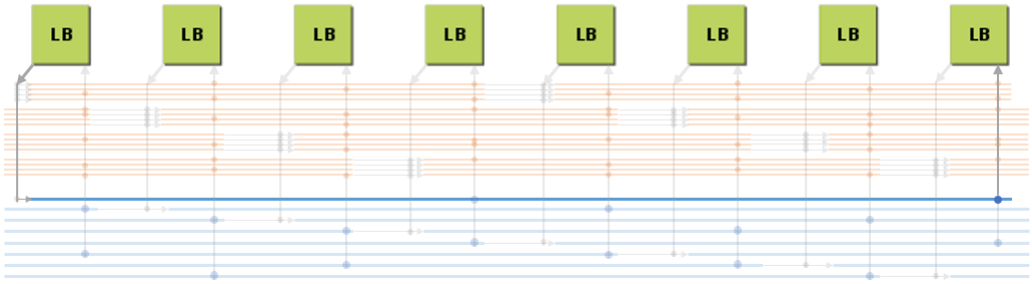

Figure 4.5: Using a global-layer wire (blue) to connect the two endpoint logic blocks with an "*On-CB, Off CB*" topology. The signal accesses the global wire through the source connection block, and arrives via the destination connection block.

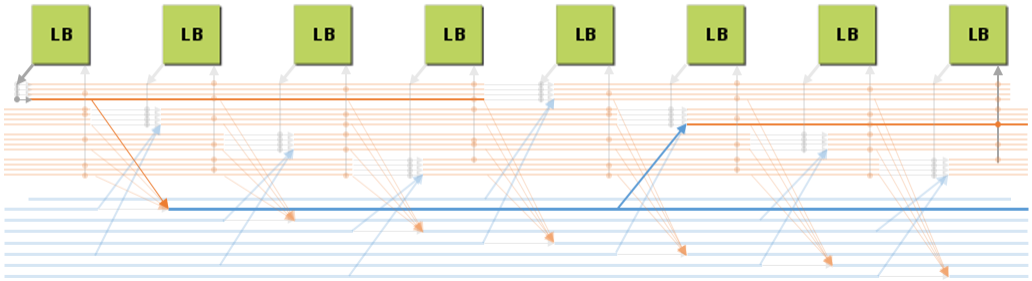

Figure 4.6: Using a global-layer wire (blue) to connect the two endpoint logic blocks with an "*On-SB, Off-SB*" topology. The signal must access the global-layer wire through the switch block using the regular semi-global wire segments.

As mentioned in Section 4.1, each topology allocates 15% of the available channel width to wires on the global metal layer, and connections to/from global-layer wires are restricted to one in every four tiles to reflect the layout difficulties of deep via stacks. The delay and per-tile routing area results are shown in Figures 4.7 and 4.8. Each topology is evaluated over a number of global-layer wire lengths and is compared to an architecture using the *VPR 7 default* switch pattern with the same wire mix (—), as well as to an architecture using only length-4 semi-global wires (- - -). The *VPR 7 default* (—) switch

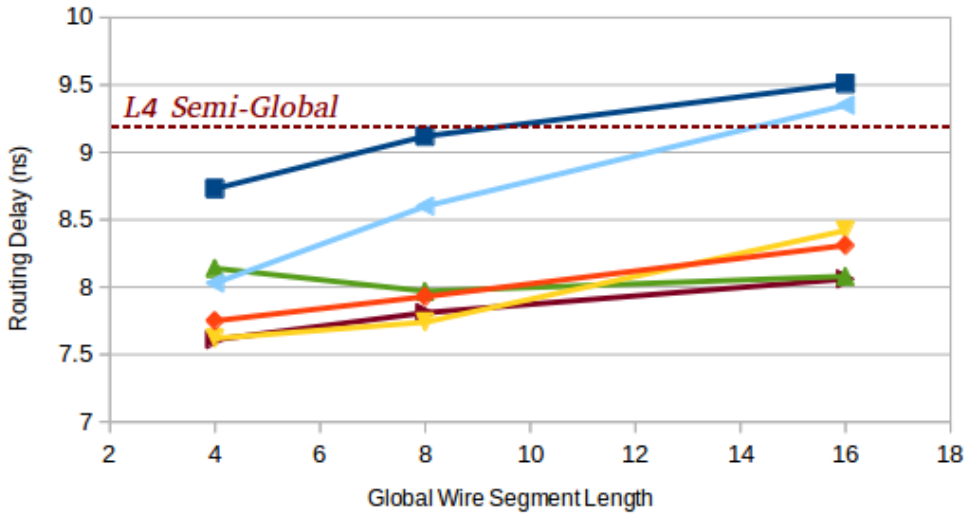

Figure 4.7: Critical path routing delay for different interconnect topologies and global metal layer wire lengths.

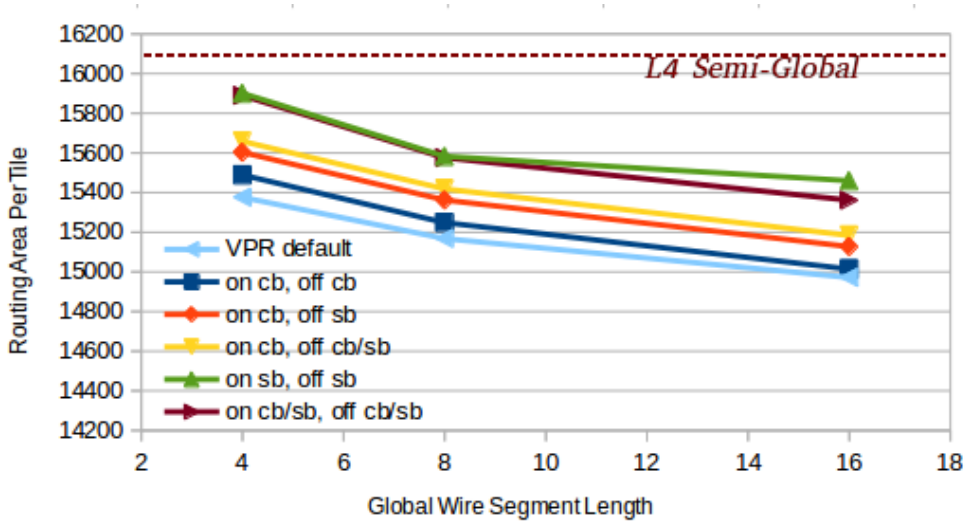

Figure 4.8: Per-tile routing area (in MWTAs) for different interconnect topologies and global metal layer wire lengths.

pattern makes no distinction between different wire types, considering only the tracks in the channel and connecting them with the Wilton switch pattern with unidirectional legalization. The line (-----) labelled *L4 Semi-Global* also uses the *VPR 7 default* switch pattern, but does not include global layer wires in the interconnect.

With reference to topology illustrations in Figure 4.4 and the results in Figures 4.7 and 4.8, the topologies are discussed below:

(a) *On CB, Off CB* (—■—): In this topology global wires are only driven by output pins and can only drive input pins and other global wires; essentially the global wires form a completely distinct routing network from regular (semi-global) wires. Figure 4.7 shows that length-4 global wires achieve a slight critical path routing delay reduction over architectures with only semi-global wires, while longer wire lengths don't provide much benefit. While this topology worked well in [9] (13% speedup), it does

not work well with global-layer wire segments, which can not be used efficiently due to the layout difficulties of deep via stacks.

(b) *On CB, Off SB* (): Global wires are only driven by output pins and can only drive other global/semi-global wires. As in (a), each output pin still has a dedicated connection to a fast wire on the global layer, but the ability of global wires to "jump down" to semi-global wires adds an extra level of flexibility to the routing and allows signals to use the global wires with greater efficiency. For the same wire type mix, this topology reduces critical-path routing delay by 4-12% compared to the default VPR switch pattern.

(c) *On CB, Off CB/SB* (): Global wires are driven only by output pins and drive global/semi-global wires and LB input pins. Compared to (b), the ability of global wires to drive some LB input pins allows some nets to be routed using the fast global layer wires exclusively which further reduces the routing delay (5-13% compared to the default VPR switch pattern) at the slight expense of extra per-tile routing area.

(d) *On SB, Off SB* (): With this topology global wires can drive, and can be driven from a fraction of the semi-global wires. Compared to (c), length-16 global wires provide a greater delay benefit since their connection with semi-global wires provides an extra degree of flexibility. On the other hand, length-4 and length-8 wires have a higher routing delay compared to (c) because signals seeking to use the global layer must first traverse semi-global wires.

(e) *On CB/SB, Off CB/SB* (): Global wires can be driven and can drive both LB pins and a fraction of the available semi-global wires. This topology uses more routing area due to the extra routing switches but provides no delay benefits over (c) for shorter wire lengths and no benefits over (d) for length-16 wires.

The results show the importance of connections between different wire types, and all topologies with guaranteed connections between wire types performed relatively well. However, the exact choice of topology depends on global-layer wire length. Two important observations are:

- Shorter global-layer wires are best driven directly from the output pins of FPGA blocks to give signals immediate access to fast routing resources ("*On-CB, Off-CB/SB*" topology in Figure 4.7). On the other hand, longer unidirectional wires have fewer start points in each channel segment, and should be driven from wires on the semi-global layer to provide a greater degree of flexibility ("*On-SB, Off-SB*" topology in Figure 4.7). Lastly, global-layer wires of all lengths should drive regular wires on the semi-global layer ("*On-CB, Off-CB*" performed poorly across all segment lengths); the restriction of having via connections every four tiles makes it impractical for global wires to drive pins directly through the input connection block without traversing the semi-global layer wires first.

- While popular commercial architectures use long wire lengths (length 16+) on the least-resistive metal layers, our explorations show that with the appropriate topology, shorter wire lengths can be made surprisingly fast. Just as short wires are more routable, shorter wires on the global metal layer allow a greater number of signals to take advantage of the fast routing resources, potentially speeding-up a larger number of timing-critical connections.

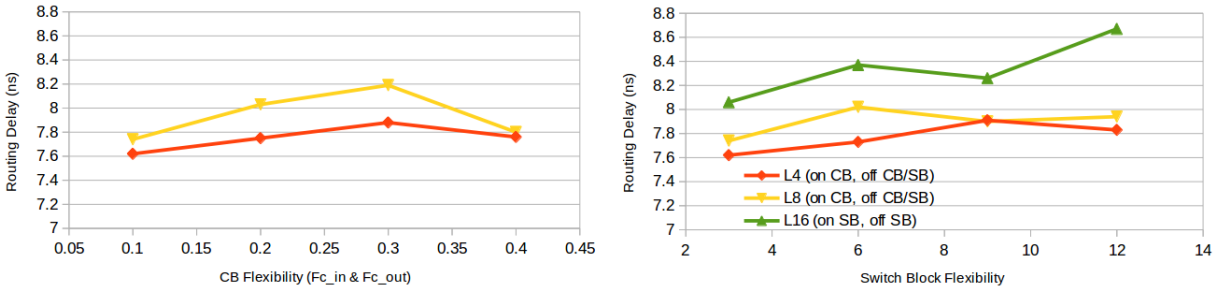

Figure 4.9: (a) Sweeping input/output connection block flexibility for best interconnect topologies at each global metal wire length (best architecture with global length 16 wires does not have global CB connections and is not included here). (b) Sweeping switch block flexiblity for best interconnect topologies at each global metal wire length.

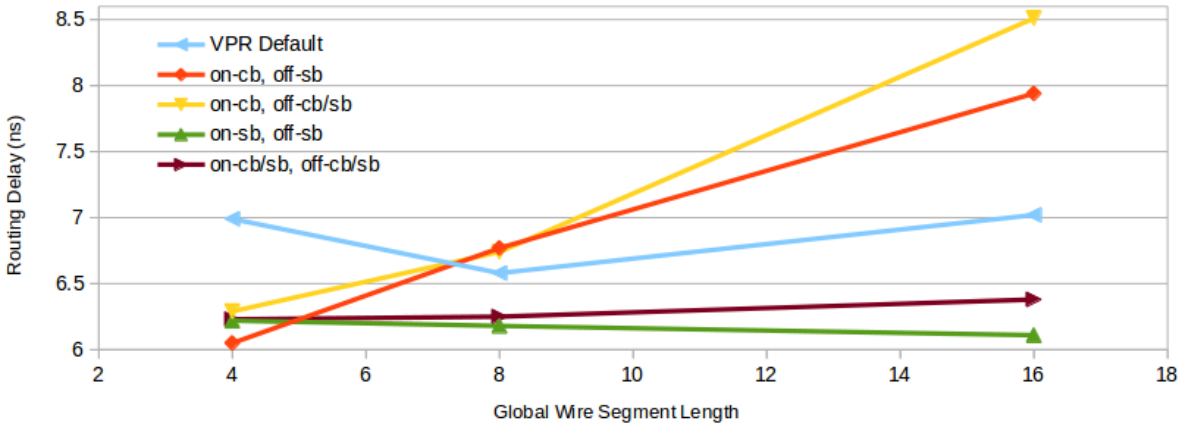

Figure 4.10: Critical path routing delay results for a 4-input LUT logic block architecture without internal crossbars.

The earlier results in this chapter were gathered with an $F_c = 0.1$ and $F_s = 3$. In the context of interconnect hierarchies using multiple wire types, we used F_s to mean the number of outgoing connections from each wire type to every other valid wire type (for example, a global length-16 wire in an *Off-SB* topology connects to $F_s = 3$ regular segments and $F_s = 3$ global-layer segments at each valid switch block). Figure 4.9 shows CB and SB flexibility sweeps for the best topology at each global wirelength – increased F_c and F_s appear to add capacitive loading without improving critical path delay through extra routing flexibility.

The sensitivity of architecture explorations to experimental setup is well known [43], and in addition to averaging our results over multiple benchmark circuits, we have also repeated our topology explorations using $W = 200$ and a logic block architecture similar to [65], which has eight 4-input LUTs and no internal crossbars (see Table 3.1). Figure 4.10 shows the critical path routing delays for this architecture over a subset of medium-size VTR benchmarks. While our main observations remain the same, one important difference is present: the lack of internal crossbars places an increased emphasis on routing flexibility, and both length-8 and length-16 global-layer wires must now be driven by regular wires on the semi-global layer; only length-4 wires see a delay benefit using *On-CB* topologies.

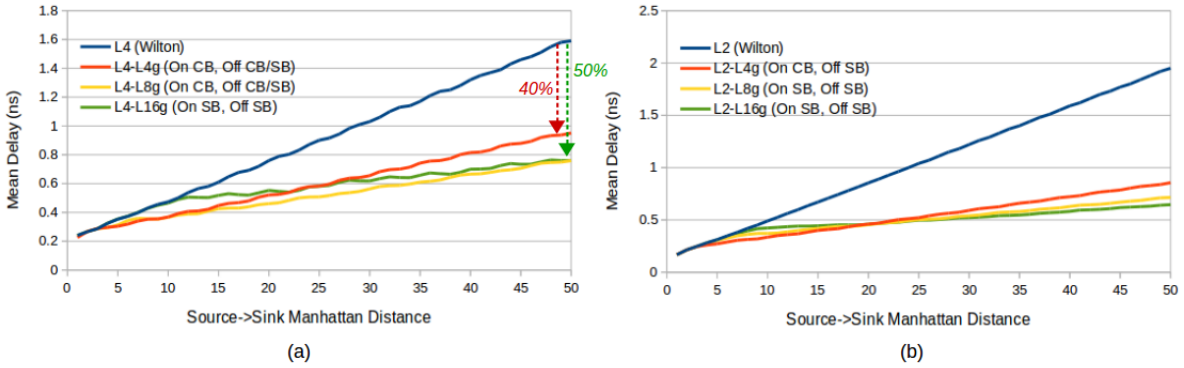

Figure 4.11: Minimum path delay (averaged over many source-sink connections) to traverse the specified Manhattan distances for (a) 6LUT-based architectures and (b) 4LUT-based architectures.

4.4 Verifying Interconnect with Best-Case Path Maps

We have also added a feature to VPR to find and print best-case delay data in order to verify the delay performance of a given interconnect topology. Selecting 50 random LB sources from the VPR routing resource graph, we use an A* search to analyze the minimum delay required to route each source to upwards of 200 random LB sinks (for each possible Manhattan distance) on a 100x100 FPGA.

The minimum delays (averaged over many source-sink connections) for different connection lengths are shown in Figure 4.11 for the best topologies explored in the previous section. This figure puts into context the surprisingly good performance of global-layer length-4 wires (Section 4.3) – they show better or equivalent delay compared to longer wire architectures over short distances while also significantly improving delay over purely semi-global topologies (along with increased routing flexibility). Only for longer distances (15 to 20 logic blocks, or more) do the longer wire architectures achieve better delays.

The best case delays can also be displayed as a 2-D map. Figure 4.12 (on page 45) shows such delay profiles for the complex 6LUT topologies, averaged over many source-sink connections (note that each sub-plot has a different scale for better delay contrast). Aside from verifying the symmetry of the delay profiles, these maps highlight the differences in routing flexibility between the complex topologies, with longer global-layer wire segments showing an increased irregularity in their delay profile.

The 2-D delay maps for 4LUT-based topologies are shown in Figure 4.13 (on page 46) and exhibit the same trends. However, the lack of internal LB crossbars in these architectures leads to a higher delay variability across the same Manhattan distance.

The best-case delay maps presented in this section can be used to verify the regularity and symmetry of the routing delays of different interconnect patterns. A similar approach may also be considered as an enhancement to the current VPR placer lookup, which checks best-case delay from only one source for each $(\Delta x, \Delta y)$ offset, and may not catch some of the consistent irregularities that appear in the delay profiles of complex interconnect.

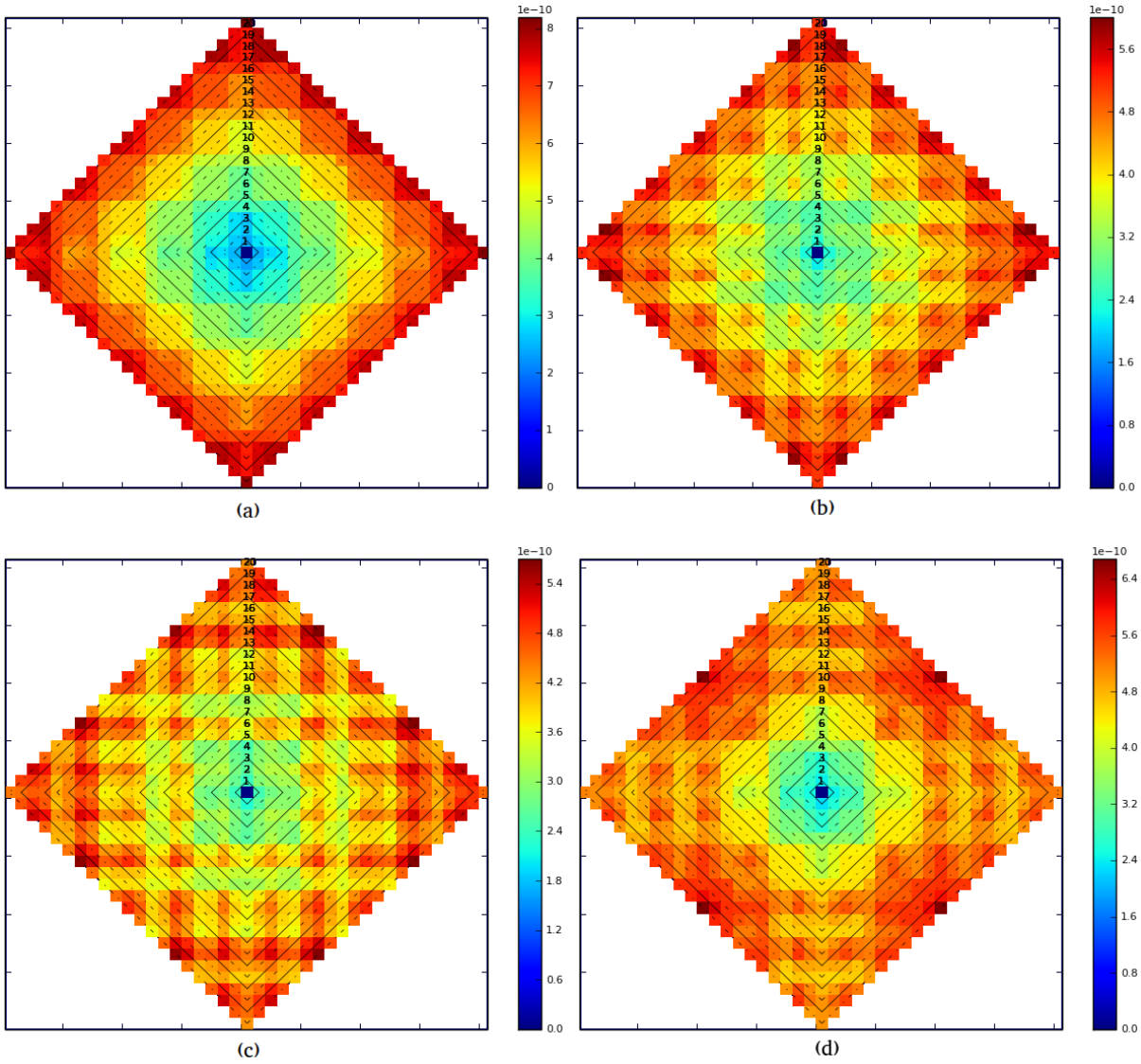

Figure 4.12: Maps of minimum delay (averaged over many source-sink connections) required to travel the relative Manhattan distances shown for 6LUT architectures. (a) L4 topology, (b) L4-L4g *On CB, Off CB/SB* topology, (c) L4-L8g *On CB, Off CB/SB* topology and (d) L4-L16g *On SB, Off SB* topology. Note that each subfigure has a different scale.

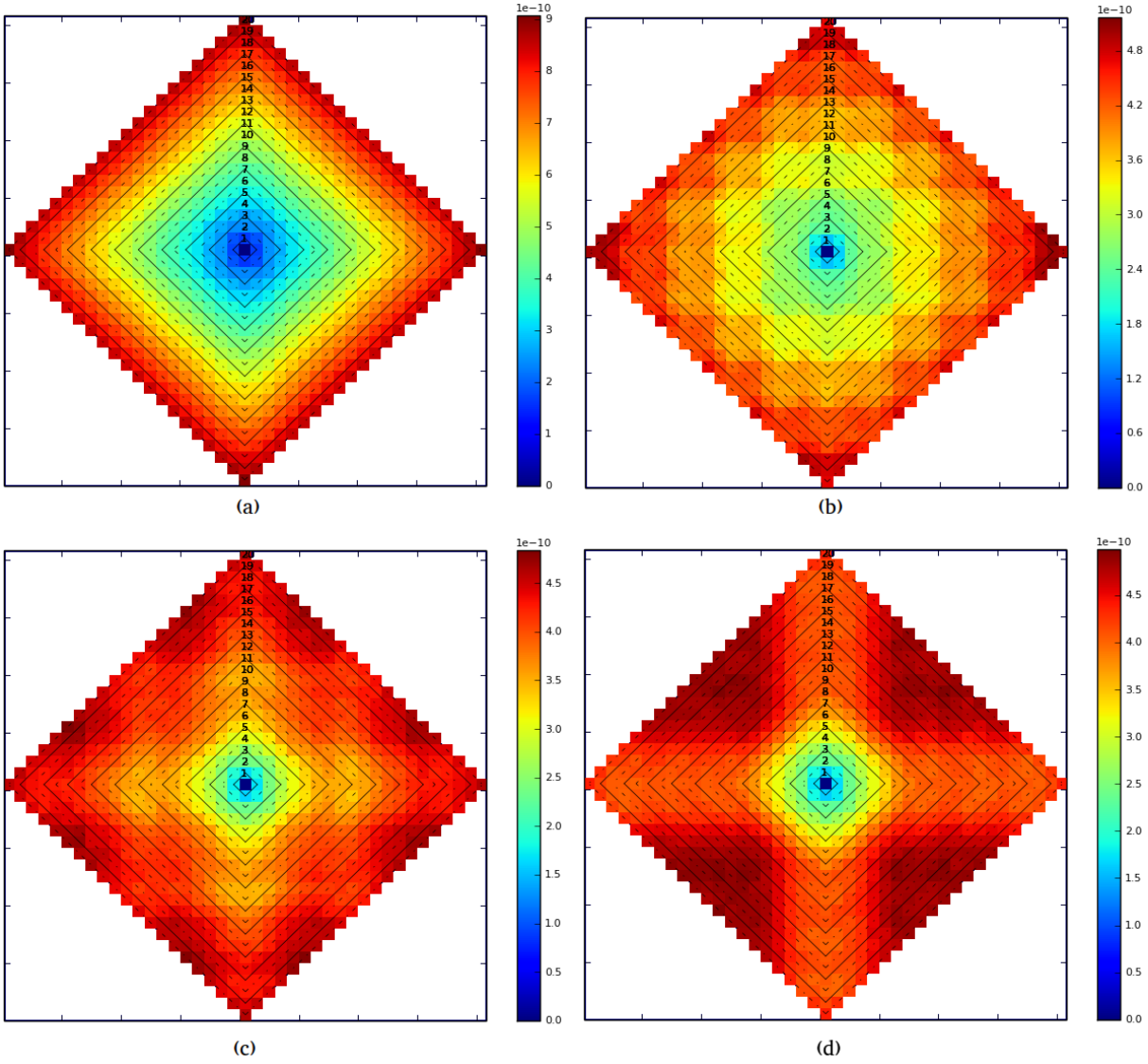

Figure 4.13: Maps of minimum delay (averaged over many source-sink connections) required to travel the relative Manhattan distances shown for 4LUT architectures. (a) L2 topology, (b) L2-L4g *On CB, Off CB/SB* topology, (c) L2-L8g *On SB, Off SB* topology and (d) L2-L16g *On SB, Off SB* topology. Note that each subfigure has a different scale.

Chapter 5

Wotan: Routing Architecture Evaluation Without Benchmarks

Traditionally, routing architectures have been evaluated using academic tools such as VTR [37] or commercial tools such as Altera's FMT [6]. As described in Section 2.3, these tools are part of a full CAD flow that synthesizes, packs, places and routes a set of benchmark circuits and evaluates the area and delay of each circuit to assess the overall architecture quality. While quite accurate, the traditional CAD flow poses three main challenges for exploring routing architectures early in the design process:

1. *Slow speed.* Multiple benchmarks must be used to obtain statistically valid results. Running the full CAD flow over each benchmark circuit and architecture option can take a long time.

2. *Limited insight.* A full CAD flow can accurately estimate the area, delay and routability of each architecture. However, such figures of merit do not give insight into the reasons why one architecture performed better than another.

3. *Tuned to an architecture.* Traditional CAD tool flows are typically targeted at a specific architecture type. Evaluating a different type of architecture with the same CAD tool may not accurately represent an architecture's potential. For example, VPR assumes that placing netlist blocks closer together according to Manhattan distance is better for the router, but if signals have to be routed around hard obstacles (e.g. large IP cores), such a placement will increase the difficulty of the routing problem.

In this chapter we present Wotan, a tool that evaluates architecture routability without benchmarks using a combination of analytic and heuristic methods. Our routability predictor represents significant improvement over prior work in this area: unlike probability-theoretic methods like [49] [50], or algorithmic approaches like [54], our predictor does not make simplifying assumptions about the routing topology, and takes into account the wire length distribution, switch pattern and the crossbar connectivity inside the FPGA logic block.

The primary focus of this chapter is to show the feasibility of our routing predictor, and we demonstrate that Wotan's routability estimates closely match the results of the full VPR flow with a significantly smaller computational effort. While not the main focus, we also suggest how Wotan can be adapted as

an efficient engine for automatic interconnect optimization, and how Wotan can be used to gain insight into bottlenecks that can potentially be present in a routing topology.

5.1 Routability Predictor – Overview

This section describes the high-level features of our routability predictor along with an example. Later in Section 5.2 we describe important parts of the predictor implementation in greater detail. The main inputs and outputs of the algorithm are described below.

- **Inputs**: The three main predictor inputs are a routing resource graph that represents the architecture being evaluated, a profile of the connection length probability distribution (e.g. short connections are more likely than long connections), and a profile of the probabilities that different logic block source nodes will be used. The routing resource graph is parsed from a text file that contains a description of the graph nodes and how they connect; this graph can be generated by VPR, or passed-in from another source. The connection length probabilities and source probabilities can be profiled from real benchmark circuits, or set based on intuition or stochastic models (Section 5.4 shows that Wotan results are not very sensitive to these particular inputs).
- **Outputs**: The main output of the Wotan flow is a single number that represents the overall routability metric of the FPGA architecture represented by the routing resource graph.

At a conceptual level, the basic principle of Wotan's operation relies on an efficient enumeration of paths between a fraction (typically 10%) of randomly chosen source-sink pairs in the routing resource graph. This efficient path enumeration is first used to set the probabilities of congestion for pins and wire segments (a node carrying many paths has a higher chance of being a chokepoint), and is then used again to estimate the probabilities that different source-sink pairs can be routed successfully (can at least one of the enumerated source-sink paths successfully navigate areas of high congestion?). Lastly, the probabilities of routing success are combined into an overall routability metric for the interconnect topology being evaluated.

Terminology and the Predictor Flow

Figure 5.1 summarizes the overall predictor flow. The predictor operates on a directed routing resource graph [7] $G(V, E)$ with a set of vertices V and edges E. Each node $v \in V$ receives a cost defined by

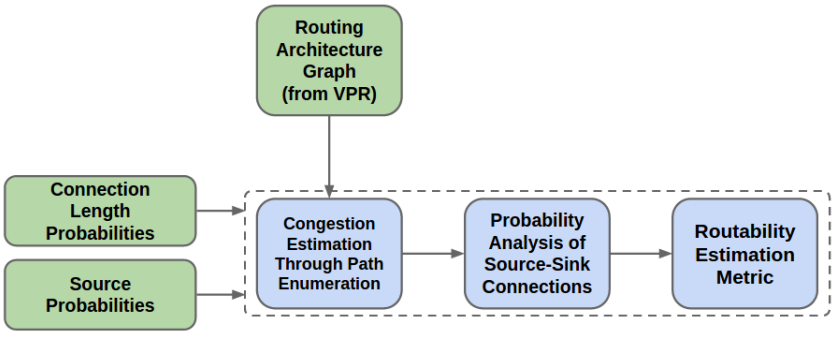

Figure 5.1: Predictor flow. Routing graph read-in from VPR.

$c(v)$ which reflects the wire length and congestion of that node (see Section 5.2.3). Sources and sinks are represented by the set of nodes $S \subseteq V$ and $T \subseteq V$ respectively; in the VPR routing graph sources and sinks represent inputs and outputs from primitives such as flip-flops or LUTs. A directed path through the graph can be written as an ordered set of nodes $(v_0, v_1, ...)$ provided that there are edges which connect each pair of consecutive nodes. Since nodes have cost, the cost of a path, $c(v_0, v_1, ...)$ is defined as the sum of the costs of the constituent nodes. We can also talk about the distance from one node to another, $d(v_i, v_j)$, as the cost of the minimum-cost path that connects v_i to v_j. Lastly, demands (congestion probabilities) are placed on nodes; a demand $De(v)$ represents the probability that the node v is congested and is unavailable for routing.

The predictor analyzes routability using efficient path enumeration through the routing resource graph based on pairwise connections between source nodes $s \in S$ and a set of sink nodes associated with each source, $T(s) \subseteq T$. The predictor steps are summarized below:

1. *Estimate congestion:* Wotan estimates node demand $De(v)$ (the probability that a given node is congested) by enumerating paths from sources in the routing resource graph to nearby sinks. To manage runtime, only source-sink connections up to a certain Manhattan distance are considered and only a fraction of those connections are enumerated. Each source-sink connection is assigned a source-sink probability (in accordance with the probability that such a connection would occur in a real circuit, as specified by the user), and that value is distributed evenly among all the paths that have been enumerated between the source and the sink to assign demands to nodes on those paths. Considering every path connecting (s, t) is exponential in complexity and may even include zig-zag paths that would be unlikely in a real routed circuit; the cost of paths considered is therefore restricted to $\leq c_b(s, t) = d(s, t) * \rho$, where ρ is a flexibility factor that allows sub-optimal paths to be considered. The final node congestion $De(v)$ is the sum of demands due to all enumerated source-sink paths that use this node.

2. *Analyze source-sink pair routability:* Wotan calculates the probability of successfully routing each source-sink connection, which is influenced by the node congestion probabilities estimated in Step 1. The graph traversal techniques and data structures from Step 1 are reused to estimate the probability that at least one uncongested path is available between each (s, t) being analyzed.

3. *Compute routability metric:* The final routability metric is computed as a weighed sum of the connection routing probabilities from Step 2. Topologies that are more efficient at distributing node demands and avoiding choke points receive a better routability metric.

Aside from the routing resource graph, Wotan also accepts two major user inputs to account for some of the behaviour of real circuit netlists and CAD tools by applying appropriate weights in the steps outlined above.

- The connection probability distribution, $P(l)$, is intended to capture the likelihood of different source-sink connection lengths occurring in actual placed circuits. For island-style architectures, connection length is defined as the Manhattan distance between the source and the sink. The connection length distribution can be profiled from real benchmark circuits, or set through stochastic models or intuition. Section 5.4 shows that our predictor results remain robust across different possible connection length distribution profiles, and a geometric distribution [50] is a very reasonable choice.

- The probability of using different sources, $P(s)$, is meant to reflect behaviour where certain source classes are more likely to be used than others. For example, this could be useful if a logic block had separate LUT and FF outputs with different probabilities of being used. While $P(s)$ can be profiled from real benchmark circuits, we have chosen to assign the same probability to all sources, as this was an intuitive assumption for the cluster-based logic blocks that we use in our interconnect topology evaluations.

The following sub-sections illustrate the predictor steps.

5.1.1 Congestion Estimation – Overview

Node demands (probabilities of congestion) are assigned based on efficient path enumeration between source/sink pairs in the routing resource graph (described in more detail under 5.2.1). A node carrying many paths receives a correspondingly higher demand which identifies it as a potential chokepoint in the routing architecture. To assign node demands, $De(v)$, the predictor enumerates all paths bounded by $c_b(s,t) = d(s,t) * \rho$ for each (s,t) pair, where $d(s,t)$ is the shortest distance between s and t and $\rho \geq 1$ is a flexibility factor that limits the sub-optimality of the paths being enumerated. Path enumeration between each source/sink pair is independent.

Each source is assigned a probability by the user-specified $P(s)$ term and this value is distributed across all the connections $T(s)$, and the paths by which these connections are implemented. Let NP_{s,t_l} be the number of paths enumerated between source s and a sink $t_l \in T(s)$, which is at a Manhattan distance l away from s. Also let $NT_{s,l}$ be the number of sinks in $T(s)$ that are a Manhattan distance l away from s. Each path enumerated between s and t_l is weighed according to:

$$W_{s,t_l} = \frac{1}{NP_{s,t_l}} * \frac{1}{NT_{s,l}} * P(l) * P(s) \tag{5.1}$$

The term $P(l) * P(s)$ first assigns a total weight to the source node's length-l connections and then distributes this weight among all the paths ($1/NP_{s,t_l}$) of each length-l connection ($1/NT_{s,l}$). These scaling factors ensure that the user-specified length probabilities $P(l)$ and source probabilities $P(s)$ are correctly reflected during path enumeration.

To illustrate how demands are assigned to routing nodes based on enumerated paths, consider the example in Figure 5.2 which shows part of an island-style routing architecture with one source s and two sinks t_0 and t_1. The source connects to two track segments through a logic block pin and is able to reach the sinks through the routing resources shown. For the sake of example, assume that:

- The cost of each wire segment is 1.
- The cost of each pin is 0.
- The probability of the source being used is $P(s) = 0.5$.
- The connection length probabilities are $P(l = 1) = 0.6$ and $P(l = 2) = 0.4$, implying that a source in this type of circuit would connect to a sink at a Manhattan distance of 1 60% of the time, and 40% of the time to a sink at a Manhattan distance of 2.
- The path cost bound is $c_b(s,t) = 2 * d(s,t)$.

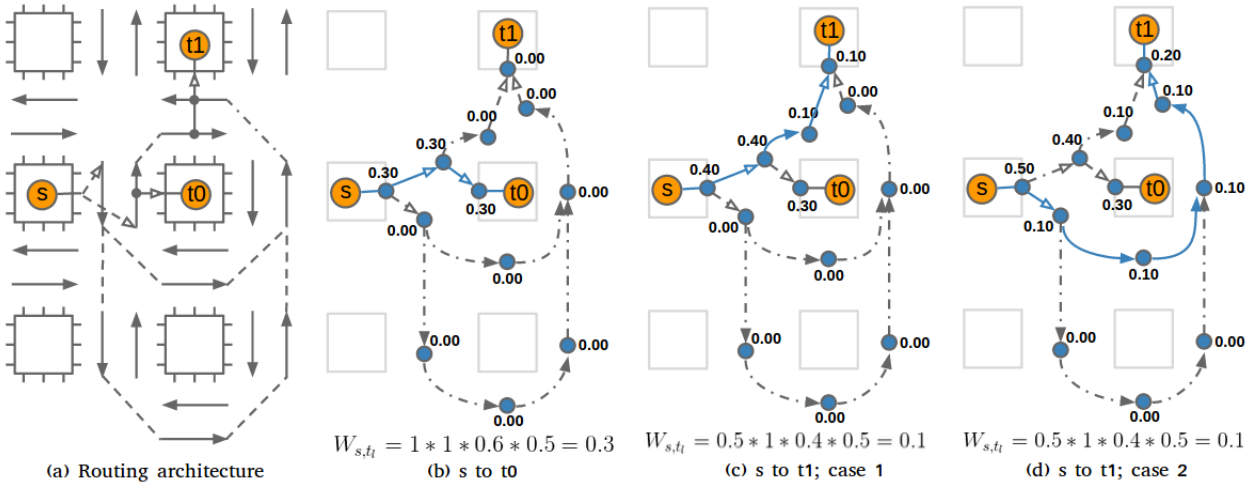

Figure 5.2: Example of demand assignment during path enumeration. Resource cost is 1 for wire segments and 0 for pins, $P(s) = 0.5$, $P(l = 1) = 0.6$, $P(l = 2) = 0.4$, and the path cost bound is $2 * d(s, t)$.

Figures 5.2.b through 5.2.d illustrate how enumerated paths add demand to nodes that they traverse:

- **5.2.b**: t_0 is a Manhattan distance of 1 away from the source s. With $P(s) = 0.5$ and $P(l = 1) = 0.6$, the total demand to be distributed among all length-1 connections is $0.5 * 0.6 = 0.3$. t_0 is the only sink at this connection length and only one path connects (s, t_0); therefore all nodes on this path receive the full demand of 0.3.
- **5.2.c & 5.2.d**: t_1 is a Manhattan distance of 2 away from the source s. With $P(s) = 0.5$ and $P(l = 2) = 0.4$, the total demand to be distributed among all length-2 connections from s is $0.5 * 0.4 = 0.2$. t_1 is the only sink at this connection length, but there are three possible (s, t) paths through the routing graph. However, only two of the paths are legal: the third path has a path cost of 6 which is above the bound $c_b(s, t_1) = 2 * d(s, t_1) = 4$. The demand of 0.2 due to length-2 connections from s is therefore distributed evenly amongst the only two legal (s, t_1) paths.
- **5.2.d**: The final demand on the routing resource nodes is the sum of the demands due to all enumerated paths.

Enumerating paths is exponentially complex if done naively. Section 5.2.1 describes how path enumeration can be performed efficiently using a topological traversal with appropriate data structures.

5.1.2 Source/Sink Routability Analysis – Overview

After node demands have been assigned through efficient path enumeration, the predictor analyzes the probability of successfully routing each source s to each of the corresponding sinks in $T(s)$. Recall from Section 2.5.2 that it is *#P-complete* to exactly compute the likelihood of connecting two nodes in a probabilistic graph. Therefore we use a simple approximation to evaluate this reliability measure. The probability of successfully routing s to a node v depends on the probability of routing s to the parents of v. As in Figure 5.3 if node v has predecessors u_0 and u_1 with probabilities $P_R(s, u_0)$ and $P_R(s, u_1)$ that they can be routed to from s respectively, then the probability of routing to node v can be approximated as in Eq. 5.2 below.

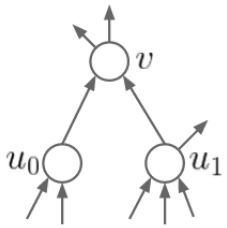

Figure 5.3: The probability of successfully routing from a source to v depends on the probability of successfully routing to v's parents u_0 and u_1.

$$P_R(s,v) \approx [1 - De(v)] * [P_R(s,u_0) + P_R(s,u_1) - P_R(s,u_0)P_R(s,u_1)] \tag{5.2}$$

With node demands having been set through efficient path enumeration, the probability of routing s to t can now be approximated by starting at s and propagating probabilities through the routing resource graph until we reach t.

Returning to the example of Figure 5.2, consider Figure 5.4 where more node demands have been assigned by enumerating other source-sink connections. We analyze the probabilities of routing $P_R(s,t_0)$ and $P_R(s,t_1)$ as in Eq. 5.2. Note that self-congestion is a potential problem during routing probability analysis because paths enumerated from a source (to a sink) can conflict with routing probability analysis from that source (to that sink); for example if the output pin to which s connects in Figure 5.4 had a demand of 1.0, all connections evaluated from s will have a routing probability of 0 unless the demand contributed to that pin by s is somehow discounted. This problem is most acute on logic block inputs and outputs as routing paths between sources and sinks generally converge there. An easy way to solve this problem for logic block pins is by associating a look-up table with each logic block pin with demand

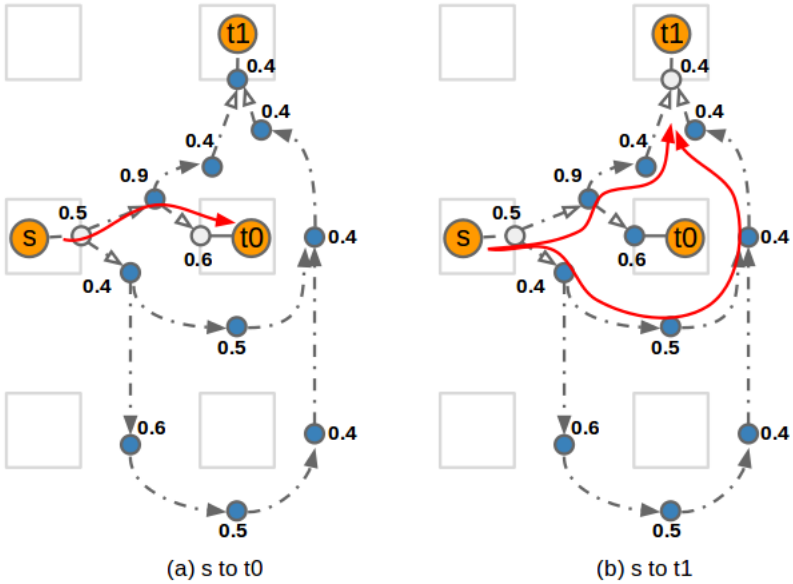

Figure 5.4: Analyzing probability of routing s to t_0 and t_1 after node demands have been fully assigned. (a) Only one path exists between s and t_0. (b) Two legal paths exist between s and t_1; the third path along the bottom falls above the $2 * d(s,t_1)$ cost bound.

to be discounted due to the sources/sinks in that logic block. The routing probabilities $P_R(s, t_0)$ and $P_R(s, t_1)$ are computed below:

- $P_R(s, t_0) = (1 - 0.9) = 0.1$. Note that we have discounted entirely the demands of 0.5 on the output/input pins of the logic blocks containing s and t_0; in this example we assume that the demand of 0.5 on the output pin is due entirely to connections originating at s and all of the demand on the input pin is due to connections terminating at t_0.

- $P_R(s, t_1) = (1 - 0.9) * (1 - 0.4)$ *OR* $(1 - 0.4) * (1 - 0.5) * (1 - 0.4) * (1 - 0.4) = 0.16$. Note that the demands due to the input/output pins have once again been discounted due to the same assumptions used in the (s, t_0) computation above. If a portion of the demand on the output/input pins was *not* due to s/t_1, that portion would not be discounted and would factor into the probability estimation.

5.1.3 Absolute Routability Metric – Overview

Recall from Section 2.5.4 that the congestion level of the routing networks being compared can influence their reliability – one interconnect topology might be more reliable than another when routing resource nodes have a low probability of congestion, but worse when congestion levels are high. Later on we compare architectures that may fall at either extreme of the reliability range, so computing the reliability of different topologies at the same values of $P(s)$ and $P(l)$ simply does not provide a good result sensitivity. To solve this problem, we compute an absolute routability metric using a reverse approach – the maximum level of congestion tolerable while achieving a predetermined value of network reliability.

Wotan applies a demand multiplier, α, as a scaling factor to the demand on each routing resource node. That is, the probability of node v being congested is scaled from $De(v)$ to $\alpha * De(v)$. The *routability* of a routing architecture is then defined as the value of α required to achieve a pre-defined architecture *reliability*:

$$Routability\ Metric = \alpha(Reliability_{target}) \tag{5.3}$$

Computing reliability and using the reliability values to compute an absolute routability metric is discussed in the following two subsections.

Computing Reliability

The reliability of a routing architecture at a given demand multiplier α is computed as a weighed sum of a percentile of the worst connection probabilities:

$$Reliability = \frac{\sum_{\substack{percentile\ worst\ s \to t \\ connections\ at\ each\ length\ l}} WR_{s,l} * P_R(s,t)}{\sum_{\substack{percentile\ worst\ s \to t \\ connections\ at\ each\ length\ l}} WR_{s,l}} \tag{5.4}$$

where the denominator is used to normalize the reliability to a range between 0 and 1 and the weight $WR_{s,l}$ is defined similarly to Equation 5.1:

$$WR_{s,l} = \frac{1}{NT_{s,l}} * P(l) * P(s) \tag{5.5}$$

In Equation 5.4 we only consider connections with poor routability because such connections ultimately dictate whether a real circuit can be routed. We have found experimentally that considering approximately 30% of the least-routable connections at each connection length yields the best agreement with routability results from the full VPR CAD flow (parameter sensitivity is discussed in Section 5.4).

Computing an Absolute Routability Metric

To compute an absolute routability metric, Wotan performs a binary search at different values of α until the reliability measure computed by Equation 5.4 matches the target value. We have found through experiments that a target reliability around 0.5 yields the best agreement with routability results from the full VPR CAD flow (see Section 5.4).

Revisiting the example of the previous two sections, we'd like to compute the absolute routability metric for the routing architecture from Figures 5.2 and 5.4 based on the $s \rightarrow t_0$ (length 1) and $s \rightarrow t_1$ (length 2) connections. The weights for each connection are:

- Weight $WR_{s,t_0} = 1 * 0.6 * 0.5 = 0.3$
- Weight $WR_{s,t_1} = 1 * 0.4 * 0.5 = 0.2$

We perform a binary search and continuously recompute the probability analysis of Section 5.1.2 until the reliability given by Equation 5.4 is close to the target value of 0.5. The computation of reliability

$$Reliability = \frac{0.3 * P_R(s,t_0) + 0.2 * P_R(s,t_1)}{0.3 + 0.2}$$

evaluates to 0.5 when the demand multiplier is approximately $\alpha = 0.6$, which is the level of congestion required to achieve the target reliability, and is the absolute routability metric of this architecture.

Note that we use α to compute the routability metric for all our results because the topologies we consider can lie on either end of the routability extreme, and so require an absolute score for accurate results. However the time-intensive binary search required to compute $\alpha(Reliability_{target})$ can be avoided with other types of experiments. For instance, if we want to compare a few candidate architectures to a baseline architecture (as part of an automatic optimization engine, for example), it can be sufficient, and significantly less time-consuming, to compute an appropriate value of α for the baseline architecture, and then use it to compute the reliabilities of the other topologies relative to the baseline.

5.2 Routability Predictor – Implementation Details

The previous section gave an overview of the three major predictor steps. However, naive path enumeration is exponential in time complexity and is further complicated by the requirement that only paths with cost $\leq c_b(s,t)$ be enumerated. This section describes some important implementation details.

5.2.1 Congestion Estimation – Implementation Details

Path enumeration between a source/sink pair (s,t) is performed in two steps using multiple forwards and backwards graph traversals over a subgraph V' connecting s and t. The first step identifies the legal

set of nodes V' and sets the node distances $d(s,v)$ and $d(v,t)$ using one forward traversal from s and one backward traversal from t. The second step enumerates the number of paths through each legal node and increments node demands, also using two traversals of the subgraph between s and t

Step 1 - Set Node Distances

The first stage identifies the set of 'legal' nodes $V' \subseteq V$ such that the smallest-cost path from s to t through $v \in V'$ is $\leq c_b(s,t)$. Clearly the cost of the minimum path from s to t through v is

$$c_{min}(s,..,v,..,t) = d(s,v) + d(v,t) - c(v) \tag{5.6}$$

where $c(v)$ is subtracted to avoid double-counting the cost of node v. Node v is part of the legal set $V' \subseteq V$ if the minimum-cost path through v falls within the path cost constraint

$$d(s,v) + d(v,t) - c(v) \leq c_b(s,t) \tag{5.7}$$

We identify the legal set of nodes $V' \subseteq V$ according to Equation 5.7 by using Dijkstra's algorithm to compute $d(s,v)$ on a forward traversal from s and then $d(v,t)$ on a backward traversal from t; the algorithm is summarized in Figure 5.5. Note that it is not necessary to traverse all the nodes V as expansion along a node can be terminated early if $d(s,v) \geq c_b(s,t)$ or if $d(s,v)$ and $d(v,t)$ are already known and the condition of Equation 5.7 can be checked. In identifying the legal set of nodes V'

```
1: //from_node: Dijsktra's traversal performed from this node
2: //dir: direction of traversal (forward/back)
3: function SET NODE DISTANCES(from_node, dir)
4:     PQ is a priority queue; lowest-cost node is always at the front
5:     PQ.push(from_node, cost = 0)
6:
7:     //Do Dijkstra's traversal
8:     while PQ not empty do
9:         n ← pop front node from PQ
10:        N ← children (or parents) of n //children or parents based on dir
11:
12:        for n' in N do
13:            if n' already visited from from_node then
14:                continue
15:            end if
16:
17:            //Set distance to n' and check legality
18:            n'.d(from_node) ← n.d(from_node) + c(n')
19:            if n'.d(from_node) > c_b(s,t) then
20:                continue //s and t cannot connect through n' within the cost bound c_b(s,t)
21:            end if
22:
23:            //Mark n' as visited and push it onto the priority queue
24:            n'.visited ← visited from from_node
25:            PQ.push(n', cost = n.d(from_node))
26:        end for
27:    end while
28: end function
```

Figure 5.5: Algorithm to identify the legal set of nodes V' between s and t and to set node distances, $d(s,v)$ and $d(v,t)$ for each $v \in V'$.

we traverse a set of nodes V'', $V' \subseteq V'' \subseteq V$, containing both legal and illegal nodes, and the time complexity of the Dijkstra's traversal is thus $O(V''log(V''))$.

Step 2 - Enumerate Paths

In the second step, we find the number of paths through each node $v \in V'$ due the source/sink pair (s,t) via one forward and one backward topological traversal. A topological traversal (or topological sort) [48] visits all the parents of v before visiting v itself, which is necessary so that a child node can account for all path counts due to its parents before propagating them downstream.

Recall that each path enumerated between s and t receives an equal weight (according to Equation 5.1) and increments the demand of all nodes it traverses by this weight. The total demand increment of node v due to the (s,t) connection is therefore based on the number of paths between s and t, and we do not track the individual paths themselves. Finding the number of paths through v due to (s,t) is based on the usual method of finding the number of paths from s to v (forward topological traversal) and from v to t (backward topological traversal) and multiplying them together [69]. However, two enhancements are necessary:

1. Paths enumerated between s and t must have a cost $\leq c_b(s,t)$. During the course of path enumeration we therefore count paths separately for each discreet cost $0, 1, ..., c_b(s,t)$.

2. Topological traversal algorithms are clear for directed acyclic graphs (DAGs), but the FPGA routing topology is obviously cyclic (it is possible to return to a starting wire segment by performing multiple turns through the routing). To deal with graph cycles, we introduce heuristics to break a child node's dependence on it's unvisited parents if we detect that the topological graph traversal has stalled.

Both enhancements are further discussed below.

Each $v \in V'$ is assigned two bin structures, one for counting paths from s to v, sp_v, and one for counting paths from v to t, tp_v. Each bin can be indexed by $sp_{v,i}$ ($tp_{v,i}$) and contains the number of paths from s to v (v to t) that have a path cost of i. So given two bin structures sp_v and tp_v, the paths of cost i in $sp_{v,i}$ can only continue onto t through the paths $tp_{v,j}$ if

$$i + j - c(v) \leq c_b(s,t) \tag{5.8}$$

The total number of paths through v due to (s,t) is then[1]

$$NP_v = \sum_{i=d(s,v)}^{c_b(s,t)} (sp_{v,i} * \sum_{j=0}^{c_b(s,t)+c(v)-i} tp_{v,j}) \tag{5.9}$$

The above two equations basically say that the paths from s to v can only continue onto t via routes through which their total cost will be $\leq c_b(s,t)$.

During the course of forward (analogous for backward) topological traversal, v's direct predecessor node u propagates its paths $sp_{u,i}$ to $sp_{v,i+c(v)}$ only if

$$i + d(v,t) \leq c_b(s,t) \tag{5.10}$$

[1]This equation can be computed at each node in $O(c_b(s,t))$ time since the second summation term can be computed incrementally.

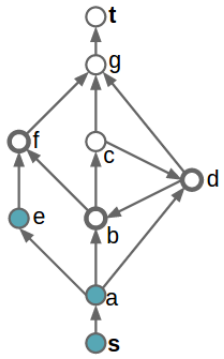

Figure 5.6: A topological traversal stalls after visiting $\{b, d, e\}$ if there is no method to deal with graph cycles. Assume $c_b(s,t) = 5$ and each node other than s and t has a cost of 1.

If the above condition is met then cost-i paths can be propagated from u to v with a guarantee that they will reach the target node within the cost bound, $c_b(s,t)$.

Next we describe how graph cycles are handled during the course of forward or backward topological traversal. Consider the graph in Figure 5.6 that shows source and sink nodes s & t and a number of intermediate nodes that can be used to connect them. A conventional topological traversal will stall because nodes b, d are part of the cycle b, c, d and will have unmet dependencies unless the cycle is broken in some way. We introduce a secondary priority queue which will hold nodes with unmet dependencies. If the traversal stalls and the expansion queue becomes empty, we then look to the secondary priority queue to select the next node to expand from, disregarding its unmet dependencies.

The nodes placed on the secondary priority queue are sorted by two factors:

- In descending order by the cost of the minimum path through v, $d(s,v) + d(v,t) - c(v)$.
- In ascending order by the distance of v to the traversal source, $d(s,v)$ (or $d(v,t)$ in the case of a backwards topological traversal).

In the case of a stall, we re-expand from the first node on the priority queue – this will be a node with unmet dependencies that is part of the most suboptimal path; the secondary sort picks nodes that are closer to the traversal source. We have found this ordering to produce a more thorough path enumeration as suboptimal paths link back to nodes that are part of more-direct paths (we do not want to give re-expansion priority to low-cost nodes such as the input pin that connects directly to the sink). Referring back to Figure 5.6, node b will be chosen for re-expansion which will then fulfill the dependencies of nodes f and d, but as a consequence paths from d to b will not be enumerated.

The path enumeration algorithm is summarized in Figure 5.7 and has a complexity of $O(E'log(V'))$, where E' is the set of edges visited while traversing V' and the $log(V')$ term is due to the priorty queue used for dealing with graph cycles.

5.2.2 Source/Sink Routability Analysis – Implementation Details

As mentioned in section 5.1.2 the probability of successfully routing a source/sink pair (s,t) can be approximated by starting at s and estimating the probability of successfully routing to each of the intermediate nodes during the course of a topological traversal. As in the previous section 5.2.1 the topological traversal takes place in a cyclic graph and we would like to propagate routing probabilities

```
1: //Require: V' identified via Dijkstra's algorithm; sp_{s,0} (or tp_{t,0}) initialized
2: //from_node & to_node: the start/end nodes of the traversal
3: //dir: direction of traversal (forward/back)
4: function Enumerate Paths(from_node, to_node, dir)
5:     Q is regular expansion queue
6:     PQ is secondary priority queue for dealing with graph cycles
7:
8:     //Do topological traversal
9:     Q.push(from_node)
10:    while Q not empty do
11:        n ← pop front node from Q
12:        N ← gets children (or parents) n //choice of children/parents depends on dir
13:
14:        if (sp_n and tp_n have both been set) && (n is not a source or sink) then
15:            increment demand at n according to Equation 5.9
16:        end if
17:
18:        //Propagate paths from n
19:        for n' in N do
20:            skip if n can't propagate legal paths to n' (Equation 5.8)
21:            skip if n' marked as done
22:
23:            propagate paths from sp_n (or tp_n) to sp_{n'} (or tp_{n'}) (Equation 5.10)
24:
25:            //Priority queue is used to deal with graph cycles
26:            if n' not in PQ then
27:                push n' onto PQ if n' has unmet dependencies
28:            else
29:                push n' onto Q (and remove from PQ) if dependencies are resolved
30:            end if
31:        end for
32:
33:        mark n as done
34:
35:        //Normal traversal has stalled due to a graph cycle. Choose node to re-expand.
36:        if Q empty && PQ not empty then
37:            move front PQ node to Q
38:        end if
39:    end while
40: end function
```

Figure 5.7: Algorithm for enumerating paths between s and t. Path weights applied by running backwards traversal to get total paths $NP_{s,t}$ and then forwards traversal with $sp_{s,0}$ weighed as in Eq.5.1

only along the legal paths that have cost $\leq c_b(s,t)$. We can reuse the bin structures from the congestion estimation step to do this; whereas previously sp_v and tp_v were used to propagate weighed path counts between s and t, they are now used to propagate probabilities from s to t. $sp_{v,i}$ will represent the probability that a path of cost i can be successfully routed to node v; this probability is approximated as the probability that a path of cost $i - c(v)$ can be routed to one of v's predecessors *and* that node v is available for routing:

$$sp_{v,i} = [1 - De(v)] * [1 - \prod_{u \in parents} (1 - sp_{u,i-c(v)})] \tag{5.11}$$

The topological traversal algorithm used for path enumeration described in Figure 5.7 can be directly adapted to propagate routing probability estimates from s to t using Eq. 5.11, instead of propagating

enumerated path counts. After the topological traversal is complete, the probability that s can successfully route to t is estimated as

$$P_R(s,t) \approx 1 - \prod_{\forall i}(1 - sp_{t,i}) \tag{5.12}$$

5.2.3 Routing Resource Costing Scheme

The simple path enumeration example from Section 5.1.1 assumed that the length-1 wire segments all had a cost of 1. Initial iterations of Wotan assigned resource costs based on the number of logic blocks spanned by each resource, reflecting the overall wiring required to use that resource; length-1 wires therefore received a cost of 1 and length-n wires received a cost of n:

$$c(v) = span(v) \tag{5.13}$$

However, with wire type mixes that use a combination of shorter and longer wires, such a costing scheme would mean that longer wire segments would be left unused. For example, with a mix of length-1 and length-16 wires, the path enumeration engine would always choose length-1 wires unless the enumerated path spanned a multiple of 16 logic blocks.

Wotan therefore uses a dynamic costing scheme to evenly distribute demands among different wire types. Wotan assigns routing resource costs based on both the span of the resource and the current resource demand demand:

$$c(v) = 1 + min[De(v), 1.0] * span(v) \tag{5.14}$$

Figure 5.8 uses Wotan graphics to illustrate the congestion patterns of an architecture using a channel with 85% length-2 wires and 15% length-8 wires. The dynamic costing scheme accounts for present congestion during path enumeration, and encourages the use of longer wire segments when the shorter segments are too congested. Unfortunately this method can be non-deterministic if Wotan is run using multiple CPU threads; Section 6.3.1 discusses other possible approaches to resource costing.

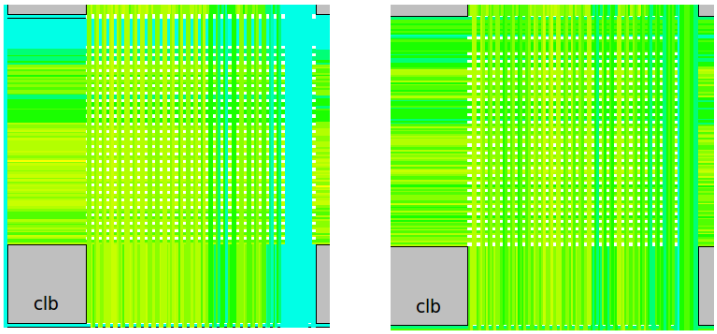

Figure 5.8: Wotan graphics showing single tile congestion for different resource costing schemes; teal wires have low congestion while green and yellow wires have higher congestion. (a) A costing scheme based on wire length under-utilizes longer wire segments (top and right sections of the tile). (b) A costing scheme that accounts for resource congestion during path enumeration helps to spread demands across all wire types.

Table 5.1: Parameters of architectures evaluated by Wotan.

Parameter	Value	Comment
FPGA size	20x20 LBs	
FPGA channel width	100	
FPGA logic architectures	6-LUT, 4-LUT	See Table 3.1
FPGA blocks evaluated	LB	Only LB-to-LB connections were analyzed.
$F_{c,in}$	0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6	
$F_{c,out}$	0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6	
Wire type mixes (6LUT)	1, 2, 4, 8, 16, 4-4g, 4-8g, 4-16g	
Wire type mixes (4LUT)	1, 2, 4, 8, 2-4g, 2-8g, 2-16g	
Routing topologies	wilton, universal, subset, on-cb/off-cb, on-cb/off-cbsb, on-cbsb/off-cbsb, on-sb/off-sb	Topology selected has to match wire type mix. See Section 4.3 for topology descriptions.

Table 5.2: Wotan parameters.

Parameter	Value	Comment
Max path length	8	Analyze (s,t) connections up to this Manhattan distance.
Path flexibility, ρ	2	Sets maximum cost of enumerated paths; $c_b(s,t) = \rho * d(s,t)$.
Connection length distribution	Geometric	Based on profiled benchmark circuit (CLMA).
Target reliability	0.5	Wotan's binary search seeks to match this reliability target.
Percentile worst connections	0.3	Percentile of least-routable connections used for metric.

5.3 Predictor Results

We use Wotan to estimate the routability of 100 6-LUT architecture points and 100 4-LUT architecture points, selecting random routing architecture parameters for each one according to Table 5.1. Selecting from among the listed $F_{c,in}$, $F_{c,out}$, wire type mix and routing topology values, the 100 6-LUT and 100 4-LUT architecture points represent a mix of both simple and complex topologies. Note also that, as discussed in Section 3.1, the 6-LUT logic blocks have a full input crossbar while the 4-LUT logic blocks only have input equivalence in groups of four pins that connect to the same LUT.

We ran Wotan using the input parameters shown in Table 5.2. The meaning of these parameters was discussed in Section 5.1 and the parameter values were selected experimentally (see Section 5.4). The connection length distribution was profiled from the CLMA benchmark circuit by decomposing the

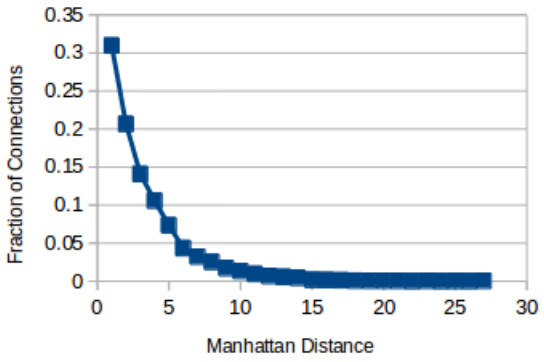

Figure 5.9: Probability distribution of 2-point MST connection lengths for the placed CLMA benchmark circuit.

placed circuit netlist into a minimum spanning tree (MST) and profiling the lengths of the 2-point MST connections; this distribution is shown in Figure 5.9.

Figures 5.10 and 5.11 show the architecture routability ordering according to Wotan and VPR (see Appendix A for detailed data tables). The x-axis in the graphs represents the architecture points sorted by VPR routability and the points are coloured according to the wire type mix used[2]. VPR evaluated routability using a subset of the largest VTR benchmarks and is the standard against which the Wotan ordering is compared. Table 5.3 quantifies the level of agreement between Wotan and VPR; the entries are explained below:

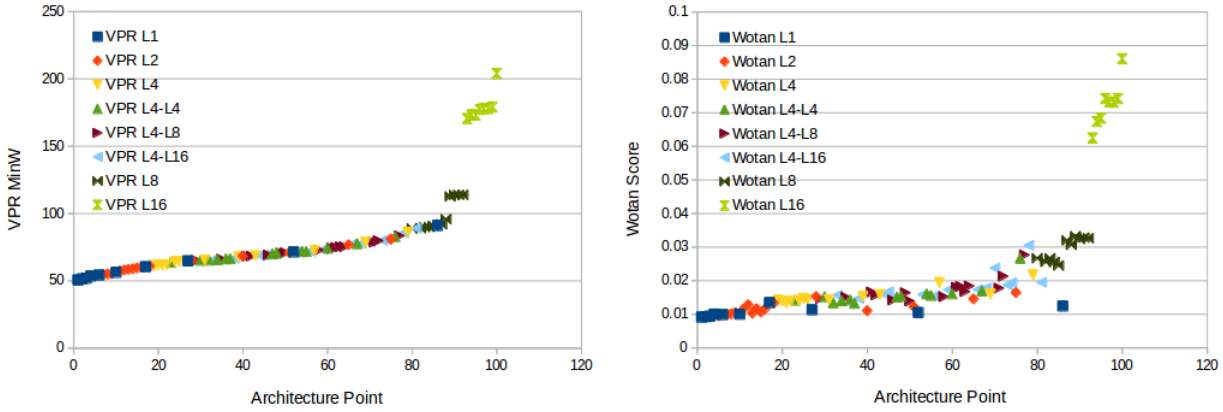

Figure 5.10: Routability results for 6-LUT architectures.

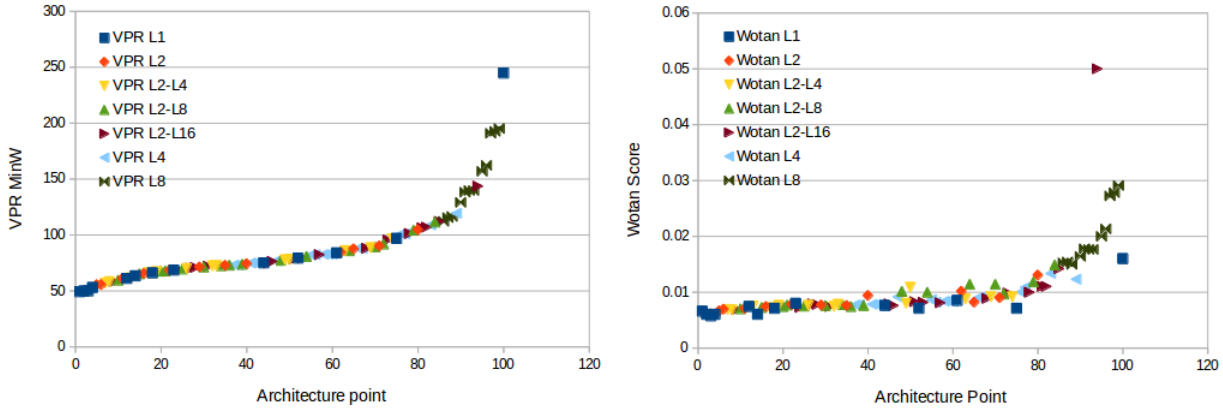

Figure 5.11: Routability results for 4-LUT architectures

Table 5.3: Comparing Wotan's result quality relative to VPR.

Spearman's Rank Correlation (6LUT)	0.91
Spearman's Rank Correlation (4LUT)	0.92
Pairwise Comparisons in Agreement (6LUT)	4438/4950 (0.90)
Pairwise Comparisons in Agreement (4LUT)	4444/4950 (0.90)
VPR Runtime / Architecture Point	20min
Wotan Runtime / Architecture Point	67s

[2]Note that $1/\alpha$ is used as the Wotan routability metric so that a higher value represents a less-routable architecture, as with VPR's minimum routable W.

- *Spearman's rank correlation*: computed by ranking every architecture according to routability and correlating the rankings between the VPR and Wotan data sets.
- *Pairwise comparisons in agreement*: computed by comparing the routability between every possible data point pair and checking how many of those comparisons agree between Wotan and VPR.
- *Wotan runtime / architecture point*: the time required to compute an absolute metric, running Wotan using 10 CPU threads on a 12-core machine (24 hyperthreads).
- *VPR runtime / architecture point*: the time required to pack, place and route the most compute-intensive benchmark (assumes that each benchmark can be analyzed on a separate CPU thread).

Table 5.3 shows that Wotan routability estimates are in good agreement with VPR, and supports the visual observation from Figures 5.10 and 5.11 that Wotan assigns the same general ranking as VPR to architecture points across a wide range of wire type mixes and switch patterns. Table 5.3 also shows that Wotan is able to a achieve a speed-up of approximately 20x over VPR with the input parameters used (Section 5.4 explores the impact of Wotan inputs on runtime and quality). Note that the Wotan runtime reflects the time requried for the binary search to compute an absolute metric; a relative comparison of architecture points (as discussed at the end of Section 5.1.3) can yield a further 10x speed-up.

While Wotan's routability estimates closely match VPR, further improvement is necessary. Figures 5.10 and 5.11 show that length-1 architectures using the subset switch block were the largest source of disagreement between VPR and Wotan routability evaluation. For both 6-LUT and 4-LUT architectures, VPR ranked some length-1 subset architectures on the same level as length-4's or length-8's; this included 6-LUT architecture points 27, 52 and 86, and 4-LUT architecture points 75 and 100 (see Appendix A). On the other hand, Wotan significantly overestimated the routability of these architectures in comparison with VPR, and while the reasons for this are not entirely clear, we can offer two possible explanations:

- Wotan enumerates and analyzes connections between one source and sink at a time. Fanout in real circuits adds "branches" to an existing route, which can increase congestion in specific FPGA locations – such extra congestion can add to bottlenecks in the routing architecture that Wotan's path enumeration does not spot. An efficient way to emulate the effects of fanout in Wotan could be to increase $P(s)$ – the demand contributed by logic block sources to pins and routing wires – in select parts of the chip. The path flexibility factor can also be increased for certain parts of the chip, allowing paths of greater sub-optimality to further increase demand in those areas. A third option, possibly offering greater fidelity at the expense of compute effort, is to instantiate *virtual* sources connected to wire segments throughout the FPGA and perform a second stage of path enumeration and analysis based on these sources.
- VPR calculates the minimum routable channel width of an architecture by performing a binary search using a set of benchmark circuits. Low routability length-1 architectures that use the subset switch block and low values of F_c can have significantly different routability at similar channel widths depending on the circuit being routed. VPR's binary search skips over a significant range of W values and may not detect that a lower W is routable. We attempted to verify the final minW values in VPR by using the *–verify_binary_search* option[3], and while agreement between VPR and

[3]With the *–verify_binary_search* option, VPR checks smaller channel widths starting from the minimum W found by the binary search and terminating after two successive channel widths fail to route.

Wotan increased slightly (Spearman's rank correlation of 0.92 for 6-LUT architectures), Wotan still significantly overestimated the routability of the length-1 subset architectures in question. However, this verification of minimum channel width is not extensive and may still have missed smaller routable channel widths.

5.4 Input Parameter Sweeps

In this section we explore the sensitivity of Wotan's routability estimates to various input parameters using the same architecture points as Section 5.3. This is an important measure to determine how useful Wotan is for evaluating new architecture types which do not have a clear choice of initial settings.

Except for the parameters being swept, the baseline inputs match those of Table 5.2. Once again, the Spearman rank correlation is used to compare Wotan's routability estimates to VPR. Overall, the parameter sweep results in this section imply that Wotan routability estimates remain robust over a large input range.

5.4.1 Maximum Path Length and Path Flexibility

Maximum path length and *path flexibility* have the largest effect on Wotan runtime as they set a bound on the maximum Manhattan distance between (s, t) connections analyzed and the sub-optimality of paths enumerated respectively. Tables 5.5 and 5.4 show Spearman's rank correlation between Wotan and VPR and Wotan runtime as a function of both *maximum path length* and *path flexibility*.

Wotan runtime increases as either of the swept parameters is increased, and is significantly smaller when a lower *maximum path length* is used. Looking at result quality, we see that sub-optimal paths should be allowed for better results, but result sensitivity is small beyond a *path flexibility* factor of 1.0. Furthermore, Wotan shows a low sensitivity to the maximum length of connections enumerated and analyzed, which suggests that Wotan run time can be decreased by lowering the *maximum path length* factor without significantly affecting result quality. This observation is in line with the profiled connection length distribution from Figure 5.9, suggesting that routability is primarily a function of many short connections.

Table 5.4: Runtime (seconds) for different *max path length* and *path flexibility* settings

Max Path Length / Path Flexibility	1	2	3
2	27	33	50
4	27	55	88
8	33	67	116
16	41	100	184

Table 5.5: Spearman's rank correlation relative to *maximum path length* and *path flexibility*

Max Path Length / Path Flexibility	1	2	3
2	0.82	0.9	0.89
4	0.81	0.9	0.89
8	0.83	0.91	0.89
16	0.83	0.91	0.9

Table 5.6: Sensitivity to *connection length distribution*.

Connection Length Distribution	Geometric	Unifrom	Geometric Inverse
Spearman's Rank Correlation	0.91	0.89	0.69

Table 5.7: Sensitivity to *target reliability* during binary search.

Target Reliability	0.05	0.3	0.5	0.7	0.95
Spearman's Rank Correlation	0.85	0.90	0.91	0.90	0.88

Table 5.8: Sensitivity to *percentile of least-routable connections* considered when calculating reliability.

Percentile Worst Connections	0.05	0.1	0.3	0.5	0.7	0.9
Spearman's Rank Correlation	0.89	0.90	0.91	0.9	0.9	0.88

5.4.2 Connection Length Distribution

The *connection length distribution* represents the probability that a given (s, t) connection length will occur; this influences the amount of congestion contributed by connections of that length, and also the weight given to those connections when calculating the final routability metric. Section 5.3 used a geometric distribution based on a profiled benchmark circuit (CLMA). Table 5.6 shows the Wotan result quality versus a few alternate distributions. The *uniform* distribution assigns the same probability to each connection length. The *geometric inverse* distribution assigns high probabilities to longer connections (with a *max path length* of eight, length-8 connections would have the highest probability which then decreases with decreasing connection length).

The uniform distribution shows a reasonable quality of results, which suggests that Wotan can be effectively used when only educated guesses about the length distribution are available; the inverse geometric distribution results imply that such guesses must be reasonable.

5.4.3 Target Reliability

The Wotan binary search which computes the absolute routability metric varies the demand multiplier α and terminates when the architecture reliability is approximately equal to the *target reliability*. Table 5.7 shows that Wotan has a low sensitivity to different values of *target reliability*, aside from either extreme where result quality drops off more sharply.

5.4.4 Percentile of Worst Connections Analyzed

Going on the intuitive observation that a small fraction of low-routability connections are sufficient to make an entire architecture unroutable, Wotan computes architecture reliability using only a percentile of the least-routable connections at each connection length. Table 5.8 shows that computing reliability using 30% of the least-routable connections yields the best results, but that other choices are also reasonable.

Chapter 6

Conclusions and Future Work

The programmable wiring of FPGAs allows them to be configured into virtually any target circuit, and accounts for the majority of the critical path delay and approximately half of the area. The optimization of FPGA interconnect is therefore a priority for architects. Moreover, RC delay has increased rapidly with process scaling, particularly for the thin wires at the bottom of the metal stack, and the challenges of poor interconnect resistance require FPGA architects to use the metal layers with greater efficiency.

In this thesis, we have sought to address some of the challenges of FPGA interconnect design by enhancing architecture models and tools, exploring complex interconnect topologies for different metal layers, and developing metrics and algorithms to promote more efficient early-stage interconnect exploration.

6.1 Architecture Models and Tool Enhancements

We have used the FPGA transistor sizing tool COFFE to create VPR architecture files for the 22nm node. The VPR architecture file format was enhanced to allow flexible and concise switch block descriptions, which can specify not only prior published switch patterns but also almost any arbitrary pattern connecting any mix of wire types. We have also improved the connection block specification format and have enhanced the routing switch specification which now allows for a list of (fanin, delay) pairs to account for variable routing switch delay with fanin.

We enhanced the VPR lookahead to handle complex interconnect topologies using a mix of wire types. Whereas the VPR 7.0 lookahead assumed that a route can be completed in an optimal number of segments using the starting wire type, the new lookahead performs sample routings to build accurate look-up tables of delay for each wire type specified in the architecture file. Our proposed lookahead has equivalent performance to VPR 7.0 for simple and moderately complex interconnect topologies, but improves delay by 8% for complex patterns at the expense of runtime. Assumptions of symmetry and translational invariance allow our lookahead to keep a low memory footprint compared to the very general, but very memory-intensive lookahead of Independence (10s of MBs versus multiple GBs for a 200x200 FPGA).

Lastly, we have enabled VPR to compute and print best-case path delay profiles. These 2D heat maps represent the minimum delay required to reach different coordinates relative to the starting tile (averaged over many connections), and are useful to verify the delay behaviour, such as regularity and

symmetry, of FPGA interconnect.

6.1.1 Future Work

Further possible enhancements related to the current work include:

- An improved connection block specification to allow a higher degree of control over the connection block switch patterns.
- An improved VPR placement delay profiler that uses best-case delay maps as described in Section 4.4.

6.2 Complex Unidirectional Interconnect Topologies

We explored the effect of unidirectional interconnect on switch patterns and showed that the restrictive nature of unidirectional wires can permute connections beyond what the switch pattern specifies, making low-routability switch blocks like the subset impossible to create for wire segments of length-2 and above. We then explored interconnect hierarchies to take advantage of the scarce but fast global metal layer, and make a few important observations about exploiting the characteristics of different wire segment lengths:

- Shorter global-layer wires are best driven directly from the output pins of FPGA blocks to give signals immediate access to fast routing resources. On the other hand, longer unidirectional wires have fewer start points in each channel segment, and should be driven from wires on the semi-global layer to provide a greater degree of flexibility. Lastly, global-layer wires of all lengths should drive regular wires on the semi-global layer; global-layer wire connections were restricted to one in four tiles to account for the layout difficulties of deep via stacks, and cannot efficiently drive pins through the input connection block.
- While popular commercial architectures use long wire lengths (length 16+) on the least-resistive metal layers, our explorations show that with the appropriate topology, shorter wire lengths can be made surprisingly fast. Just as short wires are more routable, shorter wires on the global metal layer allow a greater number of signals to take advantage of the fast routing resources, potentially speeding-up a larger number of timing-critical connections.

Our best routing hierarchies reduce delay by 5-14% compared to VPR's best prior switch patterns using the same wire type mix.

6.2.1 Future Work

While our explorations of complex interconnect topologies fill an important gap in published literature, further work is necessary:

- Our improved architecture models and tool enhancements can be used to investigate interconnect topologies with a greater number of wire types. Chapter 4 demonstrated the relative merits of short and long global-layer wire segments for improving the routing flexibility of fast interconnect and the delay of long connections respectively. Using a mix of short and long wire types on the global metal layer may provide the benefits of both, and should be investigated further.

- Chapter 4 investigated interconnect hierarchies using global-layer wire segments with connections only every 4 tiles, reflecting the layout difficulties of deep via stacks between lower and upper metal layers. Investigating appropriate interconnect hierarchies when global-layer wire segments are depopulated every 1, 2, 3, 4, or more tiles would yield additional insight.

- We improved VPR's mux delay model and architecture file format to account for fanin. Along with the enhanced switch block format, this enables the exploration of heterogeneous interconnect topologies with wire types using a combination of fast, low-fanin muxes and slower but more-flexible high-fanin muxes. Taking advantage of unbalanced mux sizing can further decrease critical path delay, especially for interconnect architectures using multiple wire types, for which larger mux fanins are possible.

6.3 Wotan: Routing Architecture Evaluation Without Benchmarks

Early stage exploration of FPGA routing architectures requires insight and rapid design iteration, motivating the exploration of fast, accurate metrics. We have developed Wotan, a tool for evaluating FPGA routability without benchmarks using a combination of analytic and heuristic methods.

We showed that Wotan's routability estimates are in good agreement with results from the full VPR CAD flow, with a Spearman's rank correlation of 0.91 for a range of complex interconnect architectures using a 6-LUT logic block, and a Spearman's rank correlation of 0.92 for complex interconnect using a 4-LUT logic block.

Wotan is approximately 18x faster than the full VPR CAD flow, and further speed-up can be achieved as discussed in Section 5.4.

6.3.1 Future Work

Many further enhancements are possible to increase Wotan's utility for FPGA architects. Interesting directions of future work include:

- Emulating the effects of fanout may further increase result quality. As discussed at the end of Section 5.3, source probability and/or path flexibility can be increased in specific chip areas to emulate the increased congestion due to high signal fanout. Fanout effects can also be emulated with a higher level of fidelity by instantiating *virtual* sources on FPGA wires and performing a second stage of path enumeration and analysis based on these sources.

- Currently, Wotan only analyzes connections that start & end with a logic block. Wotan's algorithms should be enhanced to analyze connectivity between different FPGA resources.

- Wotan uses a dynamic scheme to assign costs to routing resources based on their span and current congestion (Section 5.2.3). An alternate scheme that assigns cost based on the fraction of the resource used by a signal (e.g. two sub-segments of a length-4 wire) can also distribute demand evenly between different wire types, while maintaining determinism when running path enumeration with multiple CPU threads.

- Improving Wotan's graphics is a key area of future work. Wotan currently keeps track of low-routability connections to calculate the architecture reliability; presenting this information in a user-friendly graphical format can give an FPGA architect insight into the effect that design decisions have on the routability of specific connections. Going further, Wotan might be enhanced to compactly present this information overlaid on a single FPGA tile (if the design is tileable).

- Wotan can be enhanced to assign routing resource costs as a function of their delay. The path enumeration algorithms discussed in this chapter can then be used to spot connections that have consistently poor delays.

- It may be possible to use Wotan to estimate an architecture's minimum routable channel width. Instead of changing the demand multiplier to vary congestion in a constant channel width, channel width could instead be varied given a constant demand multiplier.

- TORCH [55] used VPR as an engine for automatic optimization of wire type mixes and switch patterns (see Section 2.4.2). While TORCH could indeed optimize FPGA interconnect fabrics automatically, it optimized power-delay without considering interconnect routability and required an extremely high computational time when exploring a limited number of interconnect architecture axes. By computing relative routability metrics and tuning performance as discussed in Sections 5.1.3 and 5.4, Wotan may be used as a routability engine requiring a significantly smaller compute effort.

Appendix A

Wotan and VPR Architecture Ranking Data

Tables A.1, A.2, A.3 and A.4 show the data for the routability estimate results discussed in Section 5.3 (Figures 5.10 and 5.11). Architecture points are sorted by name and the index of each entry corresponds to VPR's ranking of that topology. The wire length mix in the architecture name is represented by an *s#__g#* substring, denoting the length of the semi-global and global-layer wire segments respectively.

Table A.1: Wotan and VPR 6-LUT architecture results (part 1/2).

Index	Architecture Name	$1/\alpha$	VPR MinW
52	k6_s1_subset_topology-single-wirelength_fcin0.05_fcout0.4	0.0104918033	71.449684712
6	k6_s1_subset_topology-single-wirelength_fcin0.2_fcout0.4	0.0098841577	54.2465278746
3	k6_s1_subset_topology-single-wirelength_fcin0.2_fcout0.6	0.0094117647	52.003805677
86	k6_s1_subset_topology-single-wirelength_fcin0.4_fcout0.1	0.0124271767	91.0697458554
2	k6_s1_subset_topology-single-wirelength_fcin0.6_fcout0.4	0.0092754053	51.3919679482
10	k6_s1_universal_topology-single-wirelength_fcin0.1_fcout0.6	0.01	56.2870471977
1	k6_s1_universal_topology-single-wirelength_fcin0.4_fcout0.6	0.0091428571	50.5919274427
27	k6_s1_wilton_topology-single-wirelength_fcin0.05_fcout0.4	0.01132744	64.7966089044
4	k6_s1_wilton_topology-single-wirelength_fcin0.2_fcout0.6	0.0100392132	53.5971600489
17	k6_s1_wilton_topology-single-wirelength_fcin0.6_fcout0.05	0.0134736751	60.4428939919
99	k6_s16_subset_topology-single-wirelength_fcin0.1_fcout0.6	0.0742026921	179.085813071
94	k6_s16_subset_topology-single-wirelength_fcin0.2_fcout0.2	0.0673681941	173.113191741
95	k6_s16_subset_topology-single-wirelength_fcin0.2_fcout0.4	0.068380744	173.113191741
100	k6_s16_universal_topology-single-wirelength_fcin0.05_fcout0.05	0.0860503739	203.949723207
96	k6_s16_universal_topology-single-wirelength_fcin0.1_fcout0.4	0.0742026921	177.24714624
97	k6_s16_wilton_topology-single-wirelength_fcin0.1_fcout0.1	0.0731427234	177.973321749
98	k6_s16_wilton_topology-single-wirelength_fcin0.1_fcout0.6	0.0731427234	177.973321749
93	k6_s16_wilton_topology-single-wirelength_fcin0.4_fcout0.4	0.0624391219	170.415351176
65	k6_s2_subset_topology-single-wirelength_fcin0.05_fcout0.1	0.0145454545	76.7465548164
75	k6_s2_subset_topology-single-wirelength_fcin0.1_fcout0.05	0.0164102564	80.9941808074
12	k6_s2_subset_topology-single-wirelength_fcin0.2_fcout0.1	0.0128	58.0075203671
40	k6_s2_universal_topology-single-wirelength_fcin0.05_fcout0.4	0.0110344828	68.2928034386
16	k6_s2_universal_topology-single-wirelength_fcin0.1_fcout0.2	0.0116628685	60.1340006993
13	k6_s2_universal_topology-single-wirelength_fcin0.1_fcout0.4	0.0103225806	58.4638814471
7	k6_s2_universal_topology-single-wirelength_fcin0.2_fcout0.4	0.01	54.7959447547
8	k6_s2_universal_topology-single-wirelength_fcin0.2_fcout0.6	0.0100589353	55.0582800542
14	k6_s2_universal_topology-single-wirelength_fcin0.6_fcout0.05	0.0116363636	58.9408813586
11	k6_s2_universal_topology-single-wirelength_fcin0.6_fcout0.1	0.0116363636	57.3603536418
5	k6_s2_universal_topology-single-wirelength_fcin0.6_fcout0.6	0.0095167399	53.8160696448
51	k6_s2_wilton_topology-single-wirelength_fcin0.05_fcout0.2	0.0121904836	70.9317324023
28	k6_s2_wilton_topology-single-wirelength_fcin0.1_fcout0.05	0.0152380952	65.0137206843
15	k6_s2_wilton_topology-single-wirelength_fcin0.1_fcout0.6	0.0106334336	59.4332395665
18	k6_s2_wilton_topology-single-wirelength_fcin0.2_fcout0.05	0.0133333333	60.9239310784
9	k6_s2_wilton_topology-single-wirelength_fcin0.4_fcout0.2	0.0105133456	55.2899764581
66	k6_s4_g16_subset_topology-on-cb-off-cb_fcin0.4_fcout0.05	0.0172972973	77.3964301948
56	k6_s4_g16_subset_topology-on-cb-off-cbsb_fcin0.4_fcout0.1	0.0155151575	71.9628092682
70	k6_s4_g16_subset_topology-on-cbsb-off-cbsb_fcin0.1_fcout0.05	0.0237037037	78.6618630323
74	k6_s4_g16_universal_topology-on-cb-off-cb_fcin0.1_fcout0.1	0.0193939394	80.4882265281
68	k6_s4_g16_universal_topology-on-cb-off-cb_fcin0.1_fcout0.4	0.0177777778	77.7639619236
53	k6_s4_g16_universal_topology-on-cb-off-cbsb_fcin0.2_fcout0.2	0.0158024816	71.6278917297
78	k6_s4_g16_universal_topology-on-cbsb-off-cbsb_fcin0.05_fcout0.05	0.0304761905	85.5405439108
59	k6_s4_g16_universal_topology-on-cbsb-off-cbsb_fcin0.1_fcout0.6	0.0171812154	73.2696048873
44	k6_s4_g16_universal_topology-on-cbsb-off-cbsb_fcin0.2_fcout0.2	0.0159501589	68.8641313696
73	k6_s4_g16_universal_topology-on-sb-off-sb_fcin0.05_fcout0.4	0.0186861401	79.9819572554
81	k6_s4_g16_wilton_topology-on-cb-off-cbsb_fcin0.05_fcout0.2	0.0194676759	89.018844311
38	k6_s4_g16_wilton_topology-on-cbsb-off-cbsb_fcin0.6_fcout0.2	0.0146678538	66.7075106867
33	k6_s4_g16_wilton_topology-on-cbsb-off-cbsb_fcin0.6_fcout0.6	0.0156097561	65.7645825997
45	k6_s4_g16_wilton_topology-on-sb-off-sb_fcin0.6_fcout0.05	0.0166233904	69.1896575039
47	k6_s4_g4_subset_topology-on-cb-off-sb_fcin0.2_fcout0.1	0.0151479347	70.1379963737
60	k6_s4_g4_subset_topology-on-cbsb-off-cbsb_fcin0.1_fcout0.1	0.0161006225	74.3221501694
23	k6_s4_g4_subset_topology-on-sb-off-sb_fcin0.4_fcout0.4	0.0142024877	63.7947579851

Table A.2: Wotan and VPR 6-LUT architecture results (part 2/2).

Index	Architecture Name	$1/\alpha$	VPR MinW
55	k6_s4_g4_universal_topology-on-cb-off-cb_fcin0.4_fcout0.05	0.0156097561	71.867889963
37	k6_s4_g4_universal_topology-on-cb-off-cb_fcin0.6_fcout0.2	0.0132642531	66.4839209166
54	k6_s4_g4_universal_topology-on-cb-off-cbsb_fcin0.1_fcout0.4	0.0161006225	71.8130220815
34	k6_s4_g4_universal_topology-on-cb-off-cbsb_fcin0.4_fcout0.6	0.0140659242	65.861237155
67	k6_s4_g4_universal_topology-on-cbsb-off-cbsb_fcin0.05_fcout0.6	0.0169536352	77.585463433
30	k6_s4_g4_universal_topology-on-sb-off-sb_fcin0.4_fcout0.05	0.0152380952	65.2941523502
48	k6_s4_g4_wilton_topology-on-cb-off-cb_fcin0.2_fcout0.6	0.0153293354	70.5851151946
36	k6_s4_g4_wilton_topology-on-cb-off-sb_fcin0.4_fcout0.6	0.0143016709	66.4580917531
32	k6_s4_g4_wilton_topology-on-cb-off-sb_fcin0.6_fcout0.2	0.0134031369	65.4411775739
76	k6_s4_g4_wilton_topology-on-sb-off-sb_fcin0.05_fcout0.05	0.0266666667	82.8732512362
63	k6_s4_g8_subset_topology-on-cb-off-cbsb_fcin0.1_fcout0.2	0.0167320331	75.3306920257
61	k6_s4_g8_subset_topology-on-cb-off-cbsb_fcin0.2_fcout0.05	0.0180281528	74.4401182538
58	k6_s4_g8_subset_topology-on-cb-off-cbsb_fcin0.6_fcout0.05	0.0152380952	72.815556192
71	k6_s4_g8_subset_topology-on-sb-off-sb_fcin0.05_fcout0.6	0.0177777778	78.7375105048
41	k6_s4_g8_subset_topology-on-sb-off-sb_fcin0.4_fcout0.05	0.0166233904	68.3678491552
42	k6_s4_g8_subset_topology-on-sb-off-sb_fcin0.6_fcout0.05	0.0157296513	68.3796326557
64	k6_s4_g8_universal_topology-on-cb-off-cb_fcin0.2_fcout0.05	0.0184172577	75.3346743639
50	k6_s4_g8_universal_topology-on-cb-off-cb_fcin0.6_fcout0.2	0.0139130435	70.8352738283
72	k6_s4_g8_universal_topology-on-cb-off-sb_fcin0.1_fcout0.05	0.0213333333	79.845519598
77	k6_s4_g8_universal_topology-on-cbsb-off-cbsb_fcin0.05_fcout0.05	0.0276756852	83.4659775999
49	k6_s4_g8_universal_topology-on-cbsb-off-cbsb_fcin0.1_fcout0.6	0.0164102564	70.6127820698
62	k6_s4_g8_wilton_topology-on-cb-off-sb_fcin0.2_fcout0.05	0.0182857143	74.7954642597
46	k6_s4_g8_wilton_topology-on-cb-off-sb_fcin0.6_fcout0.1	0.0142222222	69.2221058772
29	k6_s4_g8_wilton_topology-on-cbsb-off-cbsb_fcin0.6_fcout0.4	0.0147126329	65.1939220226
35	k6_s4_g8_wilton_topology-on-sb-off-sb_fcin0.4_fcout0.6	0.0150588349	66.0970110822
19	k6_s4_subset_topology-single-wirelength_fcin0.4_fcout0.4	0.0142320999	61.1930248805
39	k6_s4_subset_topology-single-wirelength_fcin0.6_fcout0.05	0.0152380952	67.7065996305
20	k6_s4_subset_topology-single-wirelength_fcin0.6_fcout0.6	0.0139130435	61.5873409635
57	k6_s4_universal_topology-single-wirelength_fcin0.1_fcout0.05	0.0193939394	72.2787096969
25	k6_s4_universal_topology-single-wirelength_fcin0.2_fcout0.6	0.0147126329	64.4045020188
21	k6_s4_universal_topology-single-wirelength_fcin0.4_fcout0.2	0.0133333333	61.6242726564
31	k6_s4_universal_topology-single-wirelength_fcin0.6_fcout0.05	0.0143016709	65.388725983
22	k6_s4_universal_topology-single-wirelength_fcin0.6_fcout0.4	0.0137819638	61.8291374957
79	k6_s4_wilton_topology-single-wirelength_fcin0.05_fcout0.05	0.0216891511	86.0715711577
69	k6_s4_wilton_topology-single-wirelength_fcin0.05_fcout0.4	0.016	78.4730122837
43	k6_s4_wilton_topology-single-wirelength_fcin0.1_fcout0.6	0.0158024816	68.5083479403
26	k6_s4_wilton_topology-single-wirelength_fcin0.2_fcout0.2	0.0143016709	64.5188856363
24	k6_s4_wilton_topology-single-wirelength_fcin0.4_fcout0.1	0.0140659242	64.1655716748
89	k6_s8_subset_topology-single-wirelength_fcin0.05_fcout0.2	0.0332467809	112.642395113
87	k6_s8_subset_topology-single-wirelength_fcin0.2_fcout0.05	0.032	91.931093608
82	k6_s8_universal_topology-single-wirelength_fcin0.4_fcout0.1	0.0256	89.1204522657
85	k6_s8_universal_topology-single-wirelength_fcin0.6_fcout0.1	0.0244976152	90.8507389304
83	k6_s8_universal_topology-single-wirelength_fcin0.6_fcout0.2	0.0266710052	89.8241373821
90	k6_s8_wilton_topology-single-wirelength_fcin0.05_fcout0.2	0.032405037	113.797869428
91	k6_s8_wilton_topology-single-wirelength_fcin0.05_fcout0.4	0.032820459	113.797869428
92	k6_s8_wilton_topology-single-wirelength_fcin0.05_fcout0.6	0.0326114251	113.797869428
88	k6_s8_wilton_topology-single-wirelength_fcin0.1_fcout0.4	0.0306586709	95.8062052936
84	k6_s8_wilton_topology-single-wirelength_fcin0.4_fcout0.1	0.0255800271	89.9306773016
80	k6_s8_wilton_topology-single-wirelength_fcin0.6_fcout0.2	0.0266666667	88.9088085185

Table A.3: Wotan and VPR 4-LUT architecture results (part 1/2).

Index	Architecture Name	$1/\alpha$	VPR MinW
100	k4_s1_subset_topology-single-wirelength_fcin0.2_fcout0.1	0.016	244.795537952
75	k4_s1_subset_topology-single-wirelength_fcin0.2_fcout0.3	0.0071111111	96.9075695802
14	k4_s1_subset_topology-single-wirelength_fcin0.6_fcout0.3	0.0060377358	63.4560857464
4	k4_s1_subset_topology-single-wirelength_fcin0.6_fcout0.4	0.0060377358	53.007008124
3	k4_s1_subset_topology-single-wirelength_fcin0.6_fcout0.6	0.0057657491	50.0039396352
52	k4_s1_universal_topology-single-wirelength_fcin0.1_fcout0.6	0.0071111111	79.2879930716
61	k4_s1_universal_topology-single-wirelength_fcin0.4_fcout0.1	0.0085332969	83.9332311349
18	k4_s1_universal_topology-single-wirelength_fcin0.4_fcout0.2	0.0071111111	66.0508359575
23	k4_s1_universal_topology-single-wirelength_fcin0.6_fcout0.1	0.008	68.6192276048
2	k4_s1_universal_topology-single-wirelength_fcin0.6_fcout0.4	0.0060377358	49.8632336272
44	k4_s1_wilton_topology-single-wirelength_fcin0.1_fcout0.4	0.0076190476	75.0790439733
12	k4_s1_wilton_topology-single-wirelength_fcin0.3_fcout0.2	0.0074418605	61.1624911767
1	k4_s1_wilton_topology-single-wirelength_fcin0.6_fcout0.3	0.0065979599	49.306740542
46	k4_s2_g16_subset_topology-on-cb-off-cb_fcin0.3_fcout0.4	0.0076190476	76.3664856882
68	k4_s2_g16_subset_topology-on-cb-off-cbsb_fcin0.2_fcout0.2	0.0088888889	88.1269621706
31	k4_s2_g16_subset_topology-on-cb-off-cbsb_fcin0.4_fcout0.3	0.0074418605	72.0693641623
51	k4_s2_g16_subset_topology-on-cb-off-sb_fcin0.3_fcout0.2	0.0083117229	78.4314302459
24	k4_s2_g16_subset_topology-on-cbsb-off-cbsb_fcin0.4_fcout0.4	0.0072727273	68.965605611
9	k4_s2_g16_subset_topology-on-cbsb-off-cbsb_fcin0.6_fcout0.6	0.0068817441	58.3260139316
94	k4_s2_g16_subset_topology-on-sb-off-sb_fcin0.1_fcout0.1	**0.05**	143.669076375
27	k4_s2_g16_subset_topology-on-sb-off-sb_fcin0.4_fcout0.2	0.0079012658	70.3380938032
82	k4_s2_g16_universal_topology-on-cb-off-cb_fcin0.1_fcout0.2	0.0110344828	106.970463569
57	k4_s2_g16_universal_topology-on-cb-off-cbsb_fcin0.2_fcout0.4	0.008101233	82.4470096391
78	k4_s2_g16_universal_topology-on-cb-off-sb_fcin0.1_fcout0.3	0.01	101.35067385
45	k4_s2_g16_universal_topology-on-cb-off-sb_fcin0.3_fcout0.3	0.0079012658	75.0872282957
28	k4_s2_g16_universal_topology-on-cbsb-off-cbsb_fcin0.3_fcout0.3	0.007804878	71.0079771526
85	k4_s2_g16_universal_topology-on-sb-off-sb_fcin0.1_fcout0.1	0.0142222222	112.153181403
73	k4_s2_g16_universal_topology-on-sb-off-sb_fcin0.1_fcout0.3	0.0098462023	95.5223731866
81	k4_s2_g16_wilton_topology-on-cb-off-sb_fcin0.1_fcout0.2	0.0110344828	106.404564258
53	k4_s2_g16_wilton_topology-on-cbsb-off-cbsb_fcin0.2_fcout0.4	0.0082051282	79.4299443314
32	k4_s2_g4_subset_topology-on-cb-off-cb_fcin0.3_fcout0.6	0.0074418605	72.2365329957
74	k4_s2_g4_subset_topology-on-cb-off-sb_fcin0.1_fcout0.4	0.0091428571	96.6298481258
49	k4_s2_g4_subset_topology-on-sb-off-sb_fcin0.2_fcout0.3	0.008	77.6018332739
50	k4_s2_g4_subset_topology-on-sb-off-sb_fcin0.3_fcout0.1	0.0108474576	78.0684763363
17	k4_s2_g4_subset_topology-on-sb-off-sb_fcin0.3_fcout0.6	0.0071910371	66.006187796
8	k4_s2_g4_subset_topology-on-sb-off-sb_fcin0.6_fcout0.3	0.0068085106	58.045299807
33	k4_s2_g4_universal_topology-on-cb-off-sb_fcin0.3_fcout0.3	0.007804878	72.3632973076
63	k4_s2_g4_universal_topology-on-cbsb-off-cbsb_fcin0.1_fcout0.6	0.0087671617	85.925043959
7	k4_s2_g4_universal_topology-on-sb-off-sb_fcin0.6_fcout0.6	0.0068817441	56.9210723661
19	k4_s2_g4_wilton_topology-on-cb-off-cb_fcin0.4_fcout0.6	0.0076190476	67.0398893742
69	k4_s2_g4_wilton_topology-on-cb-off-sb_fcin0.1_fcout0.6	0.0091428571	88.5046887787
26	k4_s2_g4_wilton_topology-on-cb-off-sb_fcin0.3_fcout0.6	0.0076190476	69.86458562
13	k4_s2_g4_wilton_topology-on-cb-off-sb_fcin0.6_fcout0.2	0.0075294401	61.5000167145
79	k4_s2_g8_subset_topology-on-cb-off-cbsb_fcin0.1_fcout0.2	0.0118518519	104.281874009
64	k4_s2_g8_subset_topology-on-cb-off-sb_fcin0.3_fcout0.1	0.0114285714	86.0244335959
36	k4_s2_g8_subset_topology-on-cb-off-sb_fcin0.3_fcout0.6	0.0074418605	73.0704319782
30	k4_s2_g8_subset_topology-on-cbsb-off-cbsb_fcin0.3_fcout0.3	0.0076190476	71.4252994677
39	k4_s2_g8_universal_topology-on-cb-off-cb_fcin0.3_fcout0.6	0.0076190476	73.8701772403
34	k4_s2_g8_universal_topology-on-cb-off-cb_fcin0.4_fcout0.2	0.007804878	72.5572543702

Table A.4: Wotan and VPR 4-LUT architecture results (part 2/2).

Index	Architecture Name	$1/\alpha$	VPR MinW
54	k4_s2_g8_universal_topology-on-cb-off-sb_fcin0.3_fcout0.1	0.01	80.8912733506
10	k4_s2_g8_universal_topology-on-cb-off-sb_fcin0.6_fcout0.6	0.0070329423	59.6040085217
20	k4_s2_g8_universal_topology-on-cbsb-off-cbsb_fcin0.3_fcout0.6	0.0074418605	67.7855645558
21	k4_s2_g8_universal_topology-on-cbsb-off-cbsb_fcin0.4_fcout0.2	0.0077108136	67.9620544826
15	k4_s2_g8_universal_topology-on-cbsb-off-cbsb_fcin0.4_fcout0.4	0.0072727273	65.3232002243
25	k4_s2_g8_universal_topology-on-sb-off-sb_fcin0.3_fcout0.4	0.0075294401	69.536538066
84	k4_s2_g8_wilton_topology-on-cb-off-cb_fcin0.1_fcout0.1	0.0148837209	111.585831393
70	k4_s2_g8_wilton_topology-on-cb-off-sb_fcin0.2_fcout0.1	0.0114285714	89.5014576233
48	k4_s2_g8_wilton_topology-on-cbsb-off-cbsb_fcin0.3_fcout0.1	0.0101587302	77.3590310575
72	k4_s2_g8_wilton_topology-on-sb-off-sb_fcin0.1_fcout0.3	0.0096969697	91.9591444764
65	k4_s2_subset_topology-single-wirelength_fcin0.1_fcout0.6	0.0082051282	87.6123898627
35	k4_s2_subset_topology-single-wirelength_fcin0.2_fcout0.3	0.0076190476	72.8856520835
16	k4_s2_subset_topology-single-wirelength_fcin0.4_fcout0.2	0.0074418605	65.5558407295
80	k4_s2_universal_topology-single-wirelength_fcin0.1_fcout0.1	0.0130612245	104.709260878
71	k4_s2_universal_topology-single-wirelength_fcin0.1_fcout0.3	0.0090140439	90.2483155494
22	k4_s2_universal_topology-single-wirelength_fcin0.3_fcout0.2	0.007804878	68.527869615
5	k4_s2_universal_topology-single-wirelength_fcin0.6_fcout0.4	0.0066666667	55.5194660758
62	k4_s2_wilton_topology-single-wirelength_fcin0.2_fcout0.1	0.0101587302	84.4795681418
29	k4_s2_wilton_topology-single-wirelength_fcin0.2_fcout0.6	0.0077108136	71.3959397826
40	k4_s2_wilton_topology-single-wirelength_fcin0.3_fcout0.1	0.0094117647	74.3205485919
11	k4_s2_wilton_topology-single-wirelength_fcin0.4_fcout0.4	0.0071111111	61.0261371513
6	k4_s2_wilton_topology-single-wirelength_fcin0.6_fcout0.6	0.0069565217	55.5925188475
55	k4_s4_subset_topology-single-wirelength_fcin0.4_fcout0.2	0.0086486486	82.1458188204
58	k4_s4_subset_topology-single-wirelength_fcin0.4_fcout0.6	0.0083660306	82.6124909129
41	k4_s4_subset_topology-single-wirelength_fcin0.6_fcout0.3	0.007804878	74.8558562157
42	k4_s4_subset_topology-single-wirelength_fcin0.6_fcout0.4	0.007804878	74.8558562157
43	k4_s4_subset_topology-single-wirelength_fcin0.6_fcout0.6	0.0077108136	74.8558562157
56	k4_s4_universal_topology-single-wirelength_fcin0.4_fcout0.6	0.0082051282	82.420133003
47	k4_s4_universal_topology-single-wirelength_fcin0.6_fcout0.1	0.0091428571	77.0318802473
89	k4_s4_wilton_topology-single-wirelength_fcin0.1_fcout0.6	0.0123076923	119.267988189
83	k4_s4_wilton_topology-single-wirelength_fcin0.2_fcout0.1	0.0133333333	108.88557138
77	k4_s4_wilton_topology-single-wirelength_fcin0.2_fcout0.2	0.0108474576	100.928975541
76	k4_s4_wilton_topology-single-wirelength_fcin0.2_fcout0.3	0.01	100.756877771
66	k4_s4_wilton_topology-single-wirelength_fcin0.3_fcout0.3	0.0090140439	87.9711375936
67	k4_s4_wilton_topology-single-wirelength_fcin0.3_fcout0.6	0.0091428571	87.9711375936
59	k4_s4_wilton_topology-single-wirelength_fcin0.4_fcout0.4	0.0084210526	82.8155709897
60	k4_s4_wilton_topology-single-wirelength_fcin0.4_fcout0.6	0.0083117229	82.8155709897
37	k4_s4_wilton_topology-single-wirelength_fcin0.6_fcout0.3	0.0077108136	73.1266402244
38	k4_s4_wilton_topology-single-wirelength_fcin0.6_fcout0.4	0.0077575908	73.1266402244
98	k4_s8_subset_topology-single-wirelength_fcin0.1_fcout0.2	0.027826087	192.713604652
91	k4_s8_subset_topology-single-wirelength_fcin0.3_fcout0.6	0.0177777778	138.242982015
86	k4_s8_subset_topology-single-wirelength_fcin0.6_fcout0.6	0.0152380952	112.212683442
96	k4_s8_universal_topology-single-wirelength_fcin0.2_fcout0.1	0.0213333333	162.247207893
92	k4_s8_universal_topology-single-wirelength_fcin0.3_fcout0.2	0.0176551802	139.761909095
93	k4_s8_universal_topology-single-wirelength_fcin0.3_fcout0.4	0.0176551802	139.761909095
88	k4_s8_universal_topology-single-wirelength_fcin0.6_fcout0.3	0.0149707546	116.565741327
99	k4_s8_wilton_topology-single-wirelength_fcin0.1_fcout0.1	0.0290909091	195.17459131
97	k4_s8_wilton_topology-single-wirelength_fcin0.1_fcout0.4	0.0272340055	190.920517741
95	k4_s8_wilton_topology-single-wirelength_fcin0.2_fcout0.3	0.02	156.931849036
90	k4_s8_wilton_topology-single-wirelength_fcin0.4_fcout0.2	0.0164102564	129.0886269
87	k4_s8_wilton_topology-single-wirelength_fcin0.6_fcout0.1	0.0154216749	115.542295807

Bibliography

[1] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. In *FPGA*, pages 21–30, 2006.

[2] Justin Richardson, Steven Fingulin, Diwakar Raghunathan, Chris Massie, Alan George, and Herman Lam. Comparative Analysis of HPC and Accelerator Devices: Computation, Memory, I/O, and Power. In *HPRCTA*, pages 1–10, 2010.

[3] Gordon Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 8:114–117, 1965.

[4] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, pages 365–376, 2011.

[5] Andrew Putnam et al. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ISCA*, pages 13–24, 2014.

[6] David Lewis et al. Architectural Enhancements in Stratix V. In *FPGA*, pages 147–156, 2013.

[7] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[8] Vaughn Betz and Jonathan Rose. FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density. In *FPGA*, pages 59–68, 1999.

[9] Vaugh Betz and Jonathan Rose. Circuit Design, Transistor Sizing and Wire Layout of FPGA Interconnect. In *CICC*, pages 171–174, 1999.

[10] Steven J E Wilton. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*. PhD thesis, University of Toronto, 1997.

[11] Intel Corporation. Advancing Moore's Law on 2014, 2014.

[12] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, 2011.

[13] ITRS 2013. Interconnect Report.

[14] Geoffrey Yeap. Smart Mobile SoCs Driving the Semiconductor Industry: Technology Trend, Challenges and Opportunities. In *IEDM*, 2013.

[15] Yuji Awano, Shintaro Sato, Mizuhisa Nihei, and Tadashi Sakai. Carbon Nanotubes for VLSI: Interconnect and Transistor Applications. *Proceedings of the IEEE*, 98:2015–2031, 2010.

[16] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel Loh, Don McCauley, Pat Morrow, Donald Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. Die Stacking (3D) Microarchitecture. In *MICRO*, pages 469–479, 2006.

[17] W.J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *DAC*, pages 684–689, 2001.

[18] Oleg Petelin and Vaughn Betz. The Speed of Diversity: Exploring Complex FPGA Routing Topologies for the Global Metal Layer. In *FPL*, 2016.

[19] Altera Corporation. Floating-Point IP Cores User Guide, 2015.

[20] Jonathan Rose and Stephen Brown. Flexibility of Interconnection Structures for Field-Programmable Gate Arrays. *JSSC*, pages 277–282, 1991.

[21] Vaughn Betz and Jonathan Rose. Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size. In *CICC*, pages 551–554, 1997.

[22] Kevin Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. Titan: Enabling Large and Complex Benchmarks in Academic CAD. In *FPL*, 2013.

[23] David Lewis et al. The Stratix II Logic and Routing Architecture. In *FPGA*, pages 14–20, 2005.

[24] Elias Ahmed and Jonathan Rose. The Effect of LUT and Cluster Size on Deep-Submicron FPGA Performance and Density. *TVLSI*, pages 288–298, 2004.

[25] Guy Lemieux and David Lewis. Using Sparse Crossbars within LUT Clusters. In *FPGA*, pages 59–68, 2001.

[26] Jonathan Greene et al. A 65nm Flash-Based FPGA Fabric Optimized for Low Cost and Power. In *FPGA*, pages 87–95, 2011.

[27] Mike Hutton et al. Improving FPGA Performance and Area Using an Adaptive Logic Module. In *FPL*, pages 135–144, 2004.

[28] Xilinx. 7 Series FPGAs Configurable Logic Block: User Guide, 2014.

[29] Hadi Parandeh-Afshar, Hind Benbihi, David Novo, and Paolo Ienne. Rethinking FPGAs: Elude the Flexibility Excess of LUTs with And-Inverter Cones. In *FPGA*, pages 119–128, 2012.

[30] David Lewis et al. The Stratix Routing and Logic Architecture. In *FPGA*, pages 12–20, 2003.

[31] G. Lemieux, E. Lee, M. Tom, and A. Yu. Directional and Single-Driver Wires in FPGA Interconnect. In *FPT*, pages 41–48, 2004.

[32] Xilinx. UltraScale Architecture and Product Overview.

[33] Yao-Wen Chang and D F Wong. Universal Switch Modules for FPGA Design. *ACM TODAES*, 1(1):80–101, 1996.

[34] Guy Lemieux and David Lewis. Analytical Framework for Switch Block Design. In *FPL*, pages 122–131, 2002.

[35] Guy Lemieux and David Lewis. *Design of Interconnection Networks for Programmable Logic*. Springer New York, 2004.

[36] Muhammad Imran Masud. FPGA Routing Structures: A Novel Switch Block and Depopulated Interconnect Matrix Architectures. Master's thesis, University of British Columbia, 1999.

[37] Jason Luu et al. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM TRETS*, 7(2), 2014.

[38] Jonathan Rose et al. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. In *FPGA*, pages 77–86, 2012.

[39] Peter Jamieson, Kenneth Kent, Farnaz Gharibian, and Lesley Shannon. Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research. In *FCCM*, pages 149–156, 2010.

[40] Alan Mishchenko et al. ABC: A System for Sequential Synthesis and Verification. `https://www.eecs.berkeley.edu/alanmi/abc`, 2012.

[41] Saeyang Yang. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0, 1991.

[42] Altera Corporation. Stratix 10: The Most Powerful, Most Efficient FPGA for Signal Processing, 2015.

[43] Andy Yan, Rebecca Cheng, and Steven Wilton. On the Sensitivity of FPGA Architectural Conclusions to Experimental Assumptions, Tools, and Techniques. In *FPGA*, pages 147–156, 2002.

[44] Subodh Gupta, Jason Anderson, Linda Farragher, and Qiang Wang. CAD Techniques for Power Optimization in Virtex-5 FPGAs. In *CICC*, pages 85–88, 2007.

[45] Adrian Ludwin and Vaughn Betz. Efficient and Deterministic Parallel Placement for FPGAs. *TDAES*, 16, 2011.

[46] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-Driven Placement for FPGAs. In *FPGA*, pages 203–213, 2000.

[47] Larry McMurchie and Carl Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs. In *FPGA*, pages 111–117, 1995.

[48] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Ed.* The MIT Press, 2009.

[49] Abbas El Gamal. Two-Dimensional Stochastic Model for Interconnections in Master Slice Integrated Circuits. *IEEE CAS*, pages 127–138, 1981.

[50] Stephen Brown, Jonathan Rose, and Zvonko G Vranesic. A Stochastic Model to Predict the Routability of Field-Programmable Gate Arrays. *IEEE TCAD*, 12(12):1827–1838, 1993.

[51] Wei Fang and Jonathan Rose. Modeling Routing Demand for Early-Stage FPGA Architecture Development. In *FPGA*, pages 139–148, 2008.

[52] Akshay Sharma, Carl Ebeling, and Scott Hauck. Architecture Adaptive Routability-Driven Placement for FPGAs. In *FPL*, pages 427–432, 2005.

[53] Akshay Sharma and Scott Hauck. Accelerating FPGA Routing Using Architecture-Adaptive A* Techniques. In *FPT*, pages 225–232, 2005.

[54] Joydip Das and Steven Wilton. An Analytical Model Relating FPGA Architecture Parameters to Routability. In *FPGA*, pages 181–184, 2011.

[55] Mingjie Lin, John Wawrzynek, and Abbas ElGamal. Exploring FPGA Routing Architecture Stochastically. *IEEE TCAD*, pages 1509–1522, 2010.

[56] Jinan Lou, Thakur Shashidhar, Shankar Krishnamoorthy, and Henry Sheng. Estimating Routing Congestion Using Probabilistic Analysis. *IEEE TCAD*, 21:32–41, 2002.

[57] Parivallal Kannan and Dinesh Bhatia. Interconnect Estimation for FPGAs. *IEEE TCAD*, 25:1523–1534, 2006.

[58] K Fujiyoshi, Y Kajitani, and H Niitsu. Design of Optimum Totally-Perfect Connection-Blocks of FPGA. In *ISCAS*, pages 221–224, 1994.

[59] Guy Lemieux, Paul Leventis, and David Lewis. Generating Highly-Routable Sparse Crossbars for PLDs. In *FPGA*, pages 155–164, 2000.

[60] Charles Colbourn. *The Combinatorics of Network Reliability.* Oxford University Press, 1987.

[61] Leslie Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM J. Comput.*, 8(3), 1979.

[62] J. G. Shanthikumar. Reliability of Systems with Consecutive Minimal Cutsets. R-36:546–550, 1987.

[63] E.F. Moore and C.E. Shannon. Reliable Circuits Using Less Reliable Relays. *Journal of the Franklin Institute*, pages 281–297, October 1956.

[64] Shant Chandrakar, Dinesh Gaitonde, and Trevor Bauer. Enhancements in UltraScale CLB Architecture. In *FPGA*, pages 108–116, 2015.

[65] Lattice Semiconductor. ECP5 and ECP5-5G Family Data Sheet, 2016.

[66] Charles Chiasson and Vaughn Betz. COFFE: Fully-Automated Transistor Sizing for FPGAs. In *FPT*, pages 34–41, 2013.

[67] ITRS 2011. Interconnect Report.

[68] Verilog-to-Routing Project. VTR Documentation. `https://vtr.readthedocs.io/en/latest`, 2016.

[69] T. Kong. A Novel Net Weighting Algorithm for Timing-Driven Placement. In *ICCAD*, pages 172–176. IEEE, 2002.