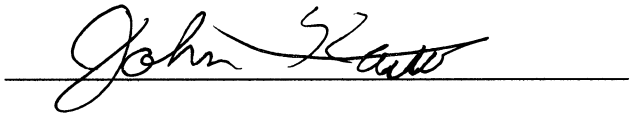
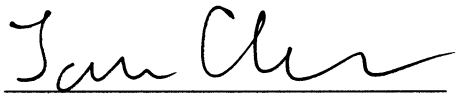
Approvals

This thesis is submitted in partial fulfillment of the requirements for the degree of
Doctor Of Philosophy
Computer Science

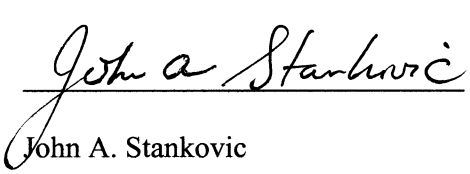


John E. Karro

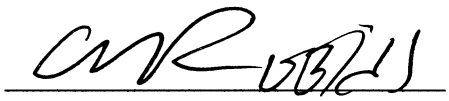
Approved:



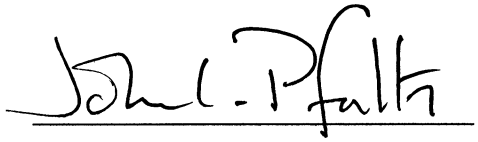
James P. Cohoon (Advisor)



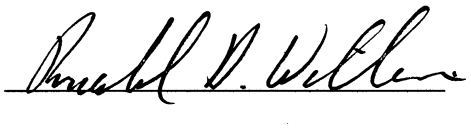
John A. Stankovic



Gabriel Robins



John L. Pfaltz (Chair)



Ronald D. Williams
(Minor Representative)

Accepted by the School of Engineering and Applied Science

Richard W. Miksad (Dean)

August, 2000

Algorithmic and Theoretical Problems Related to the Physical Design of Three Dimensional Field Programmable Gate Arrays

A Thesis

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

Computer Science

by

John E. Karro

August 2000

Abstract

Field Programmable Gate Arrays (FPGAs) have become an increasingly useful and important architecture in hardware design. As a flexible alternative to custom integrated chips, FPGA-implemented designs can be produced quickly and cheaply. However, this flexibility comes at a significant performance penalty. To help address this issue, we propose a family of three-dimensional FPGA architectures, with increased speed and smaller size as compared to existing 2D FPGAs. We implemented the first suite of tools for creating circuit designs for the new proposed architecture, and used these tools to demonstrate the efficacy of 3D FPGAs (e.g., 3D FPGA circuit mappings seem superior to those mapped to 2D ones). We explored several issues arising in the design of both 2D and 3D- FPGAs, and implemented two useful tools: (1) Spiffy, which performs placement and global routing simultaneously for 2D and 3D FPGAs, and (2) Gambit, which is the first tool to perform placement, global routing and detailed routing simultaneously, and which demonstrates the usefulness of conflict graphs. These tools yield superior solutions within reasonable runtimes, and employ a "template smoothing" technique which significantly improves the results at a modest runtime cost. Our results indicate that 3D FPGAs are a viable future architecture.

This thesis is dedicated to

Laura Tabacca,

David and Ellie Karro,

and

Jack and Nancy Karro

Acknowledgments

The first person to thank is my advisor, Dr. James Cohoon. Without his support and guidance this research could never have been completed. I also thank John Pfaltz for his advice and support. The encouragement I received from these two men were invaluable in the completion of a successful graduate school career and final dissertation. I thank Gabriel Robins, John Stankovic and Ronald Williams for serving on my committee and helping me with this work and my research.

I am grateful to a number of people for help in proofreading this document, though none more so than Laura Tabacca. Her careful and considerable efforts at editing considerably improved the writing style of this thesis, and her ability to spot error and mistakes provided me with great peace of mind. In addition, I think my father, Sean McCulloch, Rashmi Srinivasa, Michael Nahas and Gabriel Ferrer for proofreading sections of this dissertation, and Kimberly Hanks, David Coppit, and again Rashmi Srinivasa for their help in the preparation of my presentation.

I thank Joseph Ganley, Sean McCulloch, Michael Alexander, Gabriel Robins, Michael Berkellar and Matthew Saltzman for various programs and code crucial to the completion of the research.

During my doctoral studies, I have received financial support from the Virginia Aerospace Consortium, from the National Science Foundation under grants CDA 9634333, CCR 9224789, MIP 9107717, DUE 9554715, and DUE 9653413, and from the Department of Computer Science at the University of Virginia. I thank all of these organizations for their

support.

And last, I wish to thank my fellow graduate students and friends who have stood by me, supported me, made the completion of this thesis possible, and made my stay at the University of Virginia worthwhile. Among those I wish to thank are Laura Tabacca, TongTong (Theresa) Zhang, Angela Kissenger, Tara Heberling, Steven Worrell, Jay Worral, Gregory Lucado, Rich Baker, Jamie Hale, Maire Mahanes, Melissa Southwell, Jennifer Hammond, Danielle Culpepper, Cathy York, Purnima Dhavan, Louise Barbatos, and numerous others. Thank you for your friendship.

Contents

List of Figures

List of Tables

Symbols and Notation

- $\alpha(e, \kappa)$: The number of switch-blocks corresponding to edge e that do not use a net of color κ.

  Defined: Page 115

- β: The congestion vector used in route-selection.

  Defined: Page 63

- $\overline{\beta}$: The average value of the elements of β.

  Defined: Page 63

- , (S): The neighborhood of S in the simulated annealing algorithm.

  Defined: Page 60

- δ: A function from $N(e) \times S(e)$ to $\mathbb{R}$ reflecting the distance between a net and a switch block.

  Defined: Page 65

- $\eta(\nu)$: A function mapping ν to some layout-graph.

  Defined: Page 68

- ϑ: Used to denote an arbitrary switch block.

Defined: Page 115

- ι: Used to denote an arbitrary channel on the chip.

  Defined: Page 53

- κ: Used to denote an arbitrary color.

  Defined: Page 110

- $\Lambda(V', s)$: The number of minimum rectilinear Steiner arboresences in Ψ for node-set $V' \subseteq V$ with source s.

  Defined: Page 55

- $\Lambda(\Psi)$: The maximum value of $\Lambda(V', s)$ over all subsets $V' \subseteq V$ and $s \in V'$.

  Defined: Page 55

- $\mu(\vartheta)$: The set of colors currently assigned to block ϑ.

  Defined: Page 115

- ν: Used to denote an arbitrary net.

  Defined: Page 47

- ρ: The maximum number of nets that will be mapped to a switch block during the virtual-terminal assignment.

  Defined: Page 65

- σ_β: The standard deviation of the elements of β.

  Defined: Page 63

- ς: Used to denote the set of smoothing-partitions that form a smoothing-templates.

Defined: Page 88

- ς_i: The set of all smoothing-portions whose center Spiffy-portion is labeled i in Figure 6.5.

  Defined: Page 91

- $\tau(\nu)$: The thumbnail assigned to net ν.

  Defined: Page 63

- $\upsilon(G)$: An upper-bound on the chromatic number of G.

  Defined: Page 109

- $\Phi(S)$: The "score" of solution S during the simulated annealing.

  Defined: Page 60

- $\phi(G)$: The number of colors currently assigned to graph G.

  Defined: Page 114

- φ: The set of nodes in the conflict-graph.

  Defined: Page 106

- $\chi(G)$: The chromatic number of G.

  Defined: Page 109

- Ψ: The partition graph representing the partition placed on C.

  Defined: Page 47

- ω_0: The cut-off probability for the simulated annealing.

  Defined: Page 62

- b: Number of logic blocks contained on chip.

  Defined: Page 24

  Range: In commercial FPGA, $b > 100$

- C: The chip to which the circuit is being mapped.

  Defined: Page 47

- $d_{\max}$: The maximum degree of the graph being discussed.

  Defined: Page 109

- E: The edges of the partition graph.

  Defined: Page 47

  Range: In practice, $|E| \leq 20$.

- e: Used to denote an arbitrary edge in E.

  Defined: Page 63

- $f(n)$: The runtime of the integer program.

  Defined: Page 70

- G: The conflict-graph for the circuit,

  Defined: Page 106

- $g(n)$: The runtime for a graph-coloring heuristic.

  Defined: Page 119

- $M(e)$: A mapping from $N(e)$ to $S(e)$ reflecting the assignment of thumbnail edges to switch blocks.

Defined: Page 65

- $N(e)$: The set of all nets using edge e.

  Defined: Page 64

- n: The number of nets in the circuit mapping.

  Defined: Page 55

- P: Used to denote a smoothing-portion.

  Defined: Page 87

- R: The set of edges in the conflict-graph.

  Defined: Page 106

- r, p and q: The dimensions of the partition placed on the chip, with r corresponding to the number of levels.

  Defined: Page 55

  Range: In practice, each has a value of 2 or 3.

- S: Used to denote an arbitrary member of the solution space during the simulated annealing algorithm.

  Defined: Page 60

- $S(e)$ The set of all switch blocks on the partition line corresponding to edge e.

  Defined: Page 64

- s: Used to denote the source node in a rooted tree.

  Defined: Page 47

- t: The temperature in the simulated annealing algorithm.

  Defined: Page 60

- t_i: The initial temperature of the simulated annealing.

  Defined: Page 62

- V: The nodes of partition graph.

  Defined: Page 47

- v: The number of nodes in the partition graph $|V|$

  Defined: Page 53

  Range: In practice $4 \leq v \leq 12$.

- w: The channel-width of the FPGA.

  Defined: Page 105

- x, y and z: The dimensions of the FGPA in terms of logic blocks.

  Defined: Page 55

  Range: In practice, $4 \leq x, y \leq 25$ and $1 \leq z \leq 4$.

1

Introduction

From conception to physical realization, the process of creating a computer chip involves many difficult tasks. Given a function to be implemented as a digital chip, the job creating it is complex, involving difficult decisions at many different levels. A chip architecture must be selected – a choice between a number chip designs, each having various advantages and disadvantages. A functional representation of the task must be formulated, then translated into a hardware representation and physically mapped to the chip. This physical mapping must be verified, and the chip itself must then be constructed and tested. Only after this significant effort do we have an operational computer chip, be it is as complicated as a CPU or as simple as a toaster-oven timer.

Each of these tasks can be challenging and mistakes can be costly, both in terms of time and money. It is believed that many of the subtasks involved in the creation process are impossible to solve optimally in a reasonable amount of time, requiring shortcuts in the form of solution estimations. Yet, using these estimates reduces the quality of the final result, which leads to slower and larger chips. Further, as technology progresses, the circuit designs are becoming more and more complicated – making good estimates harder and harder to achieve in a reasonable amount of time.

As noted above, one important step in the creation of a chip is the selection of its

hardware architecture: the configuration of the physical components. Different architectures permit different circuit designs and impose different restrictions on the designer. The *custom design* architecture is at one end of the spectrum. Using this architecture results in the fastest and most space-efficient chips, as the only limitations are those imposed by physical laws. However, creating them is a time-consuming and expensive process.

Field Programmable Gate Arrays, referred to as FPGAs, are at the other end of the spectrum. FPGAs impose the most restrictions on circuit design. These limitations result in chips that are slower and larger than their custom design counterparts. However, FPGAs are far easier to design. The process of implementing a circuit on an FPGA is typically much faster and more cost-effective than implementing the same circuit on a custom design chip.

Once the architecture has been chosen and the functional design created, the design needs to be *mapped* to the architecture. The functions need to be translated into components that can be placed on a chip. In addition, physical locations on the chip need to be chosen for each component, and wire paths need to be selected for connecting the components. The restrictions on this *physical design* depend on the chip architecture. On a custom design chip, components can take any shape and be placed anywhere, while wires are restricted only in that they must keep certain minimal distances from other wires. On FPGAs, components have a specific size and shape, and are limited in placement to specific areas. The possible paths that wires can take are pre-designated between the component areas. The difficulties of mapping functional designs to chips vary greatly with the type of architecture.

With very small circuits it is conceivable that the physical design can be done manually. However, the time has long since passed that this can be done with reasonably-sized circuits. Because the number of components in a typical circuit now numbering easily in the millions, it is difficult for even a fast computer to perform this task in an acceptable amount of time. The physical design of such a circuit cannot be done manually. Thus the need for *design automation*: the task of delegating the physical design problems to a computer.

Physical design has been historically divided into several tasks, including *placement*, *global routing*, and *detailed routing*. The details and problems associated with each of these depends on the architecture in question, but certain goals must be achieved with each of these tasks, regardless of the hardware selected. Before the sub-tasks are performed, we must have an *abstract circuit description*, a list of components that need to be mapped to the chip surface and a specification as to how these components are to be interconnected. How this description is produced often varies; for some architectures, such as custom-design, a method known as *floor-planning* may be used. For other architectures, such as FPGAs, *technology mapping* may be employed. Given the circuit description produced, we can perform the three sub-tasks:

- **Placement:** A size and a shape has already been determined for each component (or pre-determined by the architecture type), and in the placement phase a geometric location on the chip surface is found for each component.
- **Global Routing:** Once the locations of the components have been chosen, the general connection path between components is determined. (Note that the global routing phase is sometimes treated as a part of the detailed routing phase.)
- **Detailed Routing:** Once a global route has been selected for each connection path, exact wire segment paths are chosen to implement the global routes.

The ultimate goal is to produce a chip that is as small and as fast as possible. This goal is accomplished by working towards specific objectives in each phase that frequently require the components be placed as closely together as possible, leaving only enough room to run the routing wire, and arranged so as to minimize the length of the connection paths.

This dissertation deals with the problems associated with FPGAs and the related physical design problems of placement, global routing and detailed routing. These problems are addressed as follows. In Chapter 2, we present an overview of FPGAs: a description of their architecture and an overview of the various advantages and problems associated with

them. In Chapter 3 we propose a new type of FPGA architecture to address some of the problems. This architecture is known as a *Three Dimensional* Field Programmable Gate Array (3D-FPGA). It is our thesis that the creation of a 3D-FPGA architecture is both feasible and desirable, and in this chapter we give both a theoretical justification for this thesis.

In Chapter 4 we present a more detailed explanation of physical design for the FPGA architecture. We explain the current state of algorithms dealing with the various phases, with an eye toward applying them to the 3D-FPGA architecture. We argue for a new paradigm for achieving the goal of the three phases: the simultaneous execution of all three phases, and discuss what has been done towards such a method.

In Chapter 5, Chapter 6 and Chapter 7 we present *Spiffy* and *Gambit*: tools we have created for the simultaneous placement and routing of 3D-FPGAs. Based upon the Mondrian Tool [43], Spiffy improves Mondrian's algorithm and generalizes it to the 3D architecture, replacing a number of aspects of the algorithm with superior problem solving solutions. When paired with the graph-based router of Alexander and Robins [8, 9] or the Upstart router of McCulloch and Cohoon, we have the first tool suite to generate full circuit mappings for creating circuits mappings for 3D-FPGAs, and the research provided the motivation for the construction of the architecture at Northeastern University [60, 63]. Gambit is the first tool to incorporate the detailed routing phases into the simultaneous performance of placement and global routing. We discuss the algorithms used in Gambit, showing that it is possible to perform these phases simultaneously.

In Chapter 8 we re-address the issue of 3D-FPGAs, justifying them experimentally. In our original proposal of the architecture we made theoretical arguments as to why 3D-FPGAs are superior to standard (2D) FPGAs, but we had no way of experimentally validating our arguments without physical design tools for the architecture [4]. With the creation of Spiffy this experimental validation becomes possible. With these physical design tools, we show that a 3D-FPGA architecture does lead to superior design.

In summary, the contributions resulting from this research are:

- Spiffy: A state-of-the-art tool for the simultaneous placement and global routing of FPGA. When combined with the Upstart and compared against the results of other leading tools, we achieve a 13% improvement and channel-width and a 10% improvement in path-length, while improving runtime by 38% when run on equivalent hardware. When compared against other tools on equivalent benchmarks, Spiffy produces some of the best circuit mappings in the literature.

- Template Smoothing: A new augmentation to the Spiffy algorithm, leading to further improvements in the quality of the circuit mappings. When integrating the template smoothing technique into the Spiffy algorithm, we bring about a 6% improvement in channel-width and a 7% improvement in path-length as compared to results of using Spiffy without template-smoothing.

- Gambit: The first tool to perform simultaneous placement, global routing and detailed routing for FPGAs of either dimension. While Gambit's results are not competitive with new tools, it does serve as a proof as concept. The creation of Gambit demonstrated a viable method for integrating the placement and both routing phases, and present a promising avenue of research towards a tool superior to any currently in the literature.

- The first circuit mappings for 3D-FPGAs. While other tools have been quickly modified to produce circuit mappings for 3D-FPGAs, the Spiffy tools is the first to exploit the traits of the new architecture, producing quality results.

- The first experimental justification of 3D-FPGAs, showing the architecture is worth pursuing. Through a series of experiments made possible by the creation of Spiffy, we show that in generalizing the architecture to the third dimension, we gain can gain as much as a 22% improvement in channel-width and an 11% improvement in

net length over standard FPGA circuit mappings just by the addition of three extra levels. Such improvements are significant in the world of VLSI-CAD, and justify the further exploration and creation of 3D-FPGAs.

2

Field Programmable Gate Arrays

In order to understand the motivations behind the three-dimensional field programmable gate array, it is necessary that the reader have a proper understanding of the standard FPGA architecture. To achieve this understanding requires that we provide a more thorough background in the general area of chip design. In this chapter, we explain some of the issues concerned with the physical creation of computer chips. We discuss the choices that must be made, and the trade-offs implicit in those choices. An examination of the available options will help motivate the need for FPGAs, and lead us to a detailed discussion of the FPGA architecture.

2.1 Overview

The goal of chip design is to create a chip for a given purpose. The intended function of the chip can vary, ranging from relatively simple tasks (e.g. a clock timer or an auto-transmission regulator) to much more complex jobs (e.g. an arithmetic logic unit or CPU). Regardless of the task, the chip must be constructed with certain objectives in mind. Some of the more important objectives include:

- **Maximizing Speed:** The chip should function as quickly and efficiently as possible. This requirement is more important in some applications than in others (e.g. speed in a microwave controller is less crucial than speed of an automatic targeting system). However, it is safe to assume that we wish to maximize the speed.

- **Minimizing Size:** The chip should be of minimal size. The importance of size varies with task, but smaller chips are usually less expensive and faster.

- **Minimizing Automation Complexity:** The chip should lend itself to simpler construction algorithms. How difficult is it to automate the construction of the chip? Given a function, how much work can be done automatically to create the physical realization of the function, how long will the process take, and what will be the quality of the final result? If a circuit requires millions of components, we clearly cannot derive the physical layout of those components by hand – nor can we generally wait the weeks it would take for the automation software to find an optimal solution.

- **Minimizing Production Costs:** The cost of constructing the chip should be minimal. Implicit in this is the requirement that we should be able to build the chip quickly. In a competitive market, the longer a chip's construction process takes, the less the chip is worth. The goals of minimizing production costs and design time are frequently in conflict with the goal of producing small and efficient chips.

One further goal implicit in any chip design is that the chip must function correctly, an objective that is difficult to verify. Other objectives also appear in the literature, e.g., power minimization. These objectives are beyond the scope of this dissertation. We concentrate on the goals of speed and size minimization, as well as minimizing automation complexity and production costs.

Thus, given an *abstract description* of a circuit – generally a logical or functional description of the chip's intended function – the challenge is to create a physical realization of that description in such a way as to achieve our stated goals. One of the first problems

is that of choosing a *layout environment* – an architectural structure that defines how the physical components can be placed. Different layouts lend themselves to different goals – by picking a specific environment, designers can use its characteristics to their advantage in terms of a certain goal, but doing so may be at the cost of other objectives.

2.2 Layout Environments

While there are many different types of layout environments, they can be classified into three design styles: *full custom*, *semi-custom*, and *universal* [84]. Given a set of *modules* – physical components that implement logic – and a specification as to how these modules are to be interconnected, the design style dictates a set of rules concerning the shape and placement of the modules and wires.

2.2.1 Full Custom Environments

With a full custom layout style, designers have great flexibility in how they shape and place the modules. The assignment must obey certain physical constraints – for example, a minimum distance between wires must be maintained to prevent electro-magnetic interference. But for the most part the designers may shape and size the modules as needed, place them where they want, and run the connecting wires along any path they want. In Figure 2.1 we see an example of such a circuit. Notice the variation in module shape and size, and the freedom in wire path.

With the flexibility of the custom design layout style, the designer has the freedom to efficiently pack modules and shorten wires – leading to smaller and faster chips. However, this flexibility comes at a cost: full custom chips are the most difficult to create. The freedom to place components results in complex design problems. Therefore the automation complexity can be quite high, and designing custom design chips is expensive. The physical construction of full custom chips is also quite difficult, and is further complicated by the need to construct any new chip from scratch. As a result, full custom design can involve large

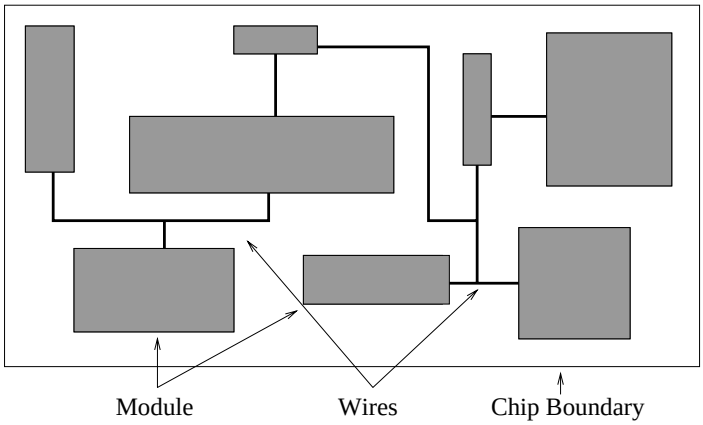


Figure 2.1: A full custom layout.

startup costs, and the entire process is time consuming. If the function being implemented is dependent on maximizing speed and minimizing size, implementing it in a custom design layout may be worthwhile. If not, it is often beneficial to sacrifice the flexibility in favor of a layout environment that can be produced at lesser cost.

2.2.2 Semi-Custom Environments

If the intended application of the computer chip allows for some sacrifice of speed or size, it is worth considering the use of a semi-custom environment. Where the full custom chip environment provides extensive flexibility, a semi-custom chip both limits the designer to the use of modules of specific shapes and sizes (typically chosen from a library) and restricts the module placement to specified locations. As a result, the designer has less options in terms of packing the modules together. However, it also results in less complex automation problems then does the full custom environment.

One class of semi-custom design chips are *cell-based* environments [73]. In a cell-based environment, designers are generally provided with a library of modules, or cells, from which they can select from to implement the various circuit functions. Each cell in the library is of

a designated shape and size, performs a designated operation, and can be placed in restricted portions of the chip (after the cells have been designed in a full custom environment). The designer must choose the operations required to implement the given function, pack the cells within the restricted area, and run the wires (which are also restricted to certain areas of the chip) between those cells requiring interconnections.

In Figure 2.2 we see an example of a standard-cell layout in which the modules are restricted to rows, hence the name *row-based architecture*. Note that the cells are of uniform height, but their width varies with function. Connecting wires are allowed to run through the *routing channels* between rows and through *feed-through cells* laid out in the cell rows. As the wires are restricted to these channels, this structure is sometimes referred to as a *channel-based* architecture.

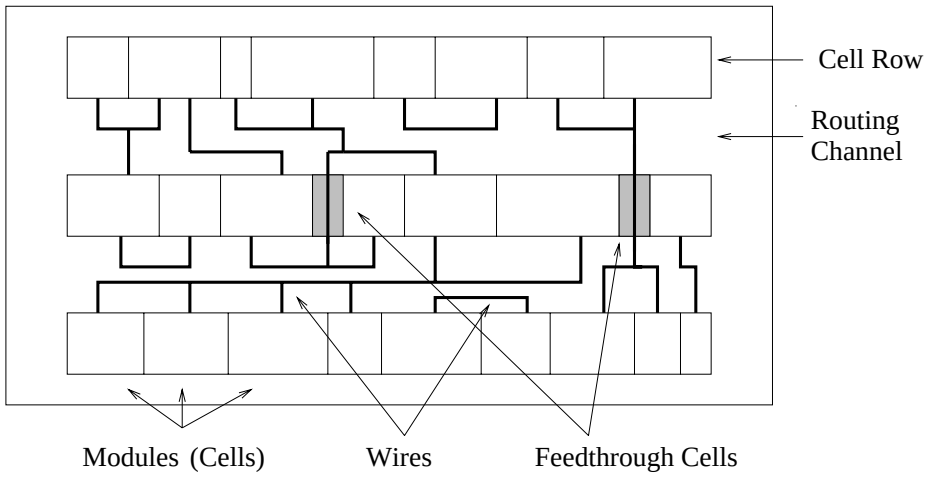


Figure 2.2: A standard cell layout.

The automation complexity of such a layout, while still quite high, is considerably less than that of the full custom design chips. As each cell has a pre-specified size and is restricted in placement options, there are fewer degrees of freedom in the solution. Hence, there is a smaller solution space to search.

In cell-based environments there are still a number of significant challenges. Once the set of cells has been picked, it is necessary to divide them among the rows and place them in some order on each row. In order to maximize the speed of the chip, we want to place them in such a way as to minimize the length of the connecting wires. At the same time, we want to minimize the size of the chips by minimizing the number of wires running through routing channels – thus minimizing the needed width of each channel (the *channel-width*). We are still subject to physical laws, e.g., wires must maintain a given distance from each other and may not cross.

2.2.3 Universal Environment

Given an arbitrary function, designers may take the extra time to implement it in a full-custom environment, or they may sacrifice speed and size in order to implement it in a semi-custom environment. Regardless of which option is picked, the chip still has to be constructed according to specifications of the functional layout. Even for semi-custom environments this is no trivial task and introduces some delay between the completion of the layout design and the physical completion of the circuit. Further, it is only after the completion of the circuit that the layout can be tested. Any corrections will require the construction of a new chip, which leads to additional time delay.

Universal environments are pre-constructed – without knowledge of the circuit to be laid out. Using programmable logic and routing resources, the components are placed in a uniform pattern on the chip and can be easily mass produced. A designer need only pick an appropriate device and program the resources in such a way as to implement the correct logic and connections. Thus there is no delay between completion of the layout design and construction of the chip. The chip has already been fabricated, which makes for a very efficient design cycle. In addition, mass production of the architecture leads to considerably lower production costs. However, this efficiency comes at a cost in terms of speed and size. Designers have little flexibility in the positioning of the components and

can only work with a pre-specified configuration. Therefore they have no freedom to tailor the chip to their specifications. Thus universal environments are usually the cheapest to build, but they result in slower and larger chips.

Although the design of universal environments is less complex than that of semi-custom chips, it is still high. Each cell must be programmed into one of a limited selection of pre-existing modules. Further, routing options are even more limited than with semi-custom design, as all possible options have been specified. Thus the designer still faces the challenge of assigning cells to locations in such a way as to minimize the length of connections. While the channel-width of the architecture is predetermined in the construction, the designer may be able to select among architectures of different widths, hence, different sizes. The challenge is to map the logic to the cells so as to minimize the required channel width – or *congestion*. This minimization enables the chip to be as small as possible.

One example of universal circuitry is the FPGA architecture, which is discussed in detail in the next section. Generally consisting of an array of logic modules with specified connection paths, FPGAs have become hugely popular in industry because of their efficient design cycle. While clearly not appropriate for applications very dependent on speed or size, they are very good for applications where these requirements are of lesser importance – as well as for prototyping circuit designs before laying them out in a more complex environment.

2.3 Field Programmable Gate Arrays

As mentioned above, FPGAs provide the advantages of the universal layout environment, and some have the additional advantage of being *reprogrammable*. The hardware can actually be reconfigured for a new use or to correct a current implementation. While still slower and larger than full-custom or semi-custom chips, low startup cost, low financial risk, and quick turn-around time usually more than make up for these problems in many applications.

FPGAs were first introduced commercially by Xilinx in 1984 [26, 106] and have since been heavily used in industry. FPGAs come in a variety of styles from a number of producers; discussions of FPGA technologies can be found in several surveys [21, 22, 74]. In this dissertation we concentrate on one particular FPGA technology: the *symmetric architecture* produced by Xilinx and others. It is the dominant version on the market. While many of the results in this paper could be generalized to include more styles of FPGAs, the symmetric architecture provides a good frame of reference in which to discuss our results.

2.3.1 The Symmetric FPGA Structure

In Figure 2.3, we see the structure of a symmetric FPGA and the components from which it is composed. We first explain the function of each component of an FPGA:

- **Logic Block**: Logic blocks perform the logic functions of the circuit. They are implemented as a programmable lookup table and can be programmed with any boolean function of up to a constant number of inputs and outputs.

- **Pins**: Pins carry the input and output signals to and from logic blocks.

- **Connection Wires**: Connection wires carry signals between components of the FPGA.

- **Connection Blocks**: Connection blocks are a programmable resource used to connect pins to channel wires. At the designer's option, a connection block can usually be programmed to connect any incident pin to any incident channel wires.

- **Switch Blocks**: Switch blocks are a programmable resource used to connect incident channels wires – thus giving the designer flexibility in signal paths. Ideally, switch blocks would provide the capacity to connect any subset of incident edges. Unfortunately, that kind of flexibility requires switch blocks too large and complicated to be practice. In commercial FPGAs only a subset of the connections is possible.

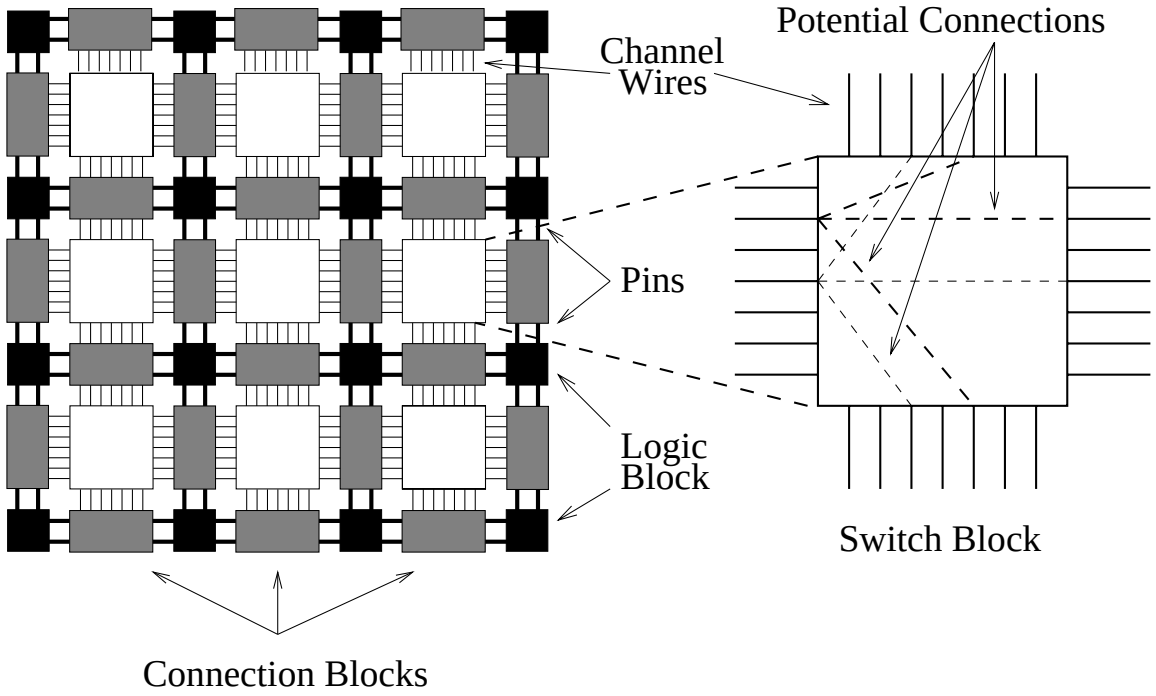


Figure 2.3: A 2D-FPGA symmetrical architecture with a channel-width of seven.

In addition to these components, it is necessary to define several more terms to facilitate discussion of the architecture:

- **Channels**: A channel is the area between two switch blocks in which the set of connecting channel wires are contained.

- **Channel-Width**: channel-width is the maximum number of wires running through each channel at any point. Note that as FPGAs are prefabricated, they are generally constructed such that the channel-width is uniform over all channels. Thus when paired with a given number of logic blocks, the channel-width defines the size of the chip.

- **Net**: Once logic blocks have been programmed with logic functions, certain pins need to be interconnected to provide input and output signals. Each set of two or more pins requiring an interconnection is known as a net. A pin can be a member of at most one net. Because a net can involve more than two pins, an interconnection is typically not a simple path but rather a tree.

- **Congestion**: The congestion of a channel is the maximum number of nets routed through the channel at any point. Since each channel wire segment can carry only one signal, the channel-width cannot be less then the congestion of any channel.

After programming the logic blocks with functions, and the switch blocks and connection blocks to correctly transmit signals and connect the nets, we need a way to quantify our results in terms of size and speed. With respect to chip size, our objective is to minimize channel-width, thus minimizing the size of the FPGA which will be used to implement the circuit. Given an abstract functional description to map to an FPGA, we generally have no control over the number of logic blocks required – hence that aspect of the FPGA size determined for us. However, we do have some control over the congestion. If we can prevent the congestion from exceeding some value n, then we can use an FPGA with channel-width n. Since channel-width determines size, we can minimize the size of the FPGA by minimizing congestion.

In maximizing speed, we need to minimize connection path lengths. The flexibility of switch blocks comes at a cost: signals are slow to travel through them. Hence we want to minimize the number of switch-blocks through which a signal must travel. Since the number of wires over which the signal travels directly determine the number of switch blocks through which it travels, we need only examine the nets' connecting trees to judge the quality of the route in terms of speed.

There are several ways to measure the size of an interconnection tree. Frequently, researchers look at the size of the total segment length that makes up the trees (taking either an average or the maximum value) and use this to rate their results [27, 44, 61].

However, it is more accurate to find the source of each net (the unique pin on each net from which the signal originates) and to consider the length of the maximum source-to-sink path in the tree (either the average or maximum value over all nets). This second metric is said to be *performance driven*, and is also frequently measured in the literature [42, 65, 76]. In this dissertation, we consider both tree-length and source-to-sink path length, and rate our tools with these metrics as appropriate.

2.3.2 Improvement of Speed and Size

As has already been emphasized, while FPGAs can be cost-effective they do not measure up to other layout styles with respect to size and speed. The designer cannot efficiently pack cells together, and the programmable hardware takes up space and introduces propagation delay. As a result, researchers have proposed numerous improvements, and are continuously experimenting with new modifications to improve the architecture [20, 100].

One option for improving size is to modify the structure of the logic blocks. By increasing the number of inputs per logic block, we increase both the size and functionality of the blocks, hence decreasing the number of blocks required. This trade off has been examined and experimented with, and it has been found that a block with four inputs provides a good balance [79, 90].

Other studies have looked at the problems of modifying the structures of switch blocks [30, 49, 78, 94]. Decreasing the flexibility of each switch block by reducing the number of options for connecting channel wires will decrease its size. But it will also decrease the routing flexibility of the FPGA by making it much more difficult to route a net in a given channel-width, generally resulting in a given circuit requiring a larger chip. This implies a limit to the reductions in size that can be made in this manner, as there will be a point at which the benefits of further reductions in size will be outweighed by the costs of further decreases in switch block flexibility.

In terms of speed, "long wires" are frequently provided – bypassing switch blocks, thus

eliminating the induced propagation delay. Others have proposed "hard-wiring" sets of logic blocks together, thus eliminating the need for switch blocks within these sets [32, 42]. The concept of a *hierarchical FPGA* has been proposed to eliminate the need for switch blocks when moving between sections of the chip [1]. All of these proposals help improve performance – but at the cost of raising the automation complexity by a substantial degree.

In the spirit of these proposals, we have developed an architectural variation with the aim of improving both size and speed without raising the design complexity or decreasing flexibility by any significant degree. In Chapter 3 we discuss our idea of a *three dimensional* FPGA, explaining the concept and providing a theoretical justification as to why this variation is worthwhile.

3

Three-Dimensional Field Programmable Gate Arrays

While the improvements discussed in Section 2.3 modify specific parts of the FPGA architecture, here we look at a more general approach to improving the performance and logic density of FPGAs. Instead of changing any one part of the FPGA, we propose changing the layout of the logic blocks. By configuring the architecture as a three-dimensional structure, we can obtain a significant improvement in performance and logic density without sacrificing design complexity or flexibility. While we were the first to propose the idea [4, 5], a number of investigations have expanded upon on our results [60, 63, 68, 87, 107]. These investigators have all concentrated on the physical challenges of constructing such a device, whereas we are concerned with the challenges of mapping circuits to the proposed model and the advantages of doing so.

In this chapter we propose and such a structure and argue for its use. Section 3.1 outlines the model of the new architecture, and Section 3.2 argues that the construction of such a chip is a feasible goal. In Section 3.3 we present a theoretical justification for the proposed architecture. We leave the experimental justification to Chapter 8.

3.1 Physical Description

Recalling the symmetric-structure of an FPGA as diagramed in Figure 2.3, it is easy to generalize the concept to the third dimension. By stacking several of these FPGAs and providing connections between vertically adjacent switch blocks, as shown in Figure 3.1, we obtain a three-dimensional architecture. Conceptually, the generalized switch-block configuration is such that each interior block has an upper and lower neighbor in addition to the standard four planar neighbors. The conceptual generalization is depicted in Figure 3.2.

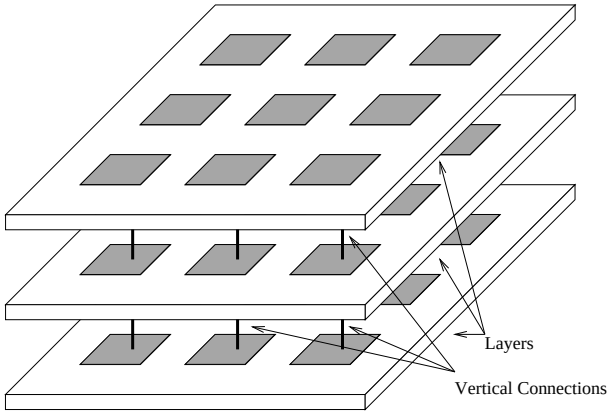


Figure 3.1: Stacking three FPGAs to form a three layer 3D-FPGA

Certain component structures of the 3D-FPGA require modification, while others remain the same as in the 2D architecture. Logic block structure is not changed; each pin on the logic block still links the logic block to a connection block within the same level. Similarly, the pin and connection block structures stay the same. However, the switch block structure must allow for vertical connections. The channels of a 3D-FPGA can be classified into two categories: planar channels and vertical channels. Planar channels run between switch blocks and connection blocks within the plane, as they do in standard FPGAs. Vertical channels run between vertically adjacent switch blocks.

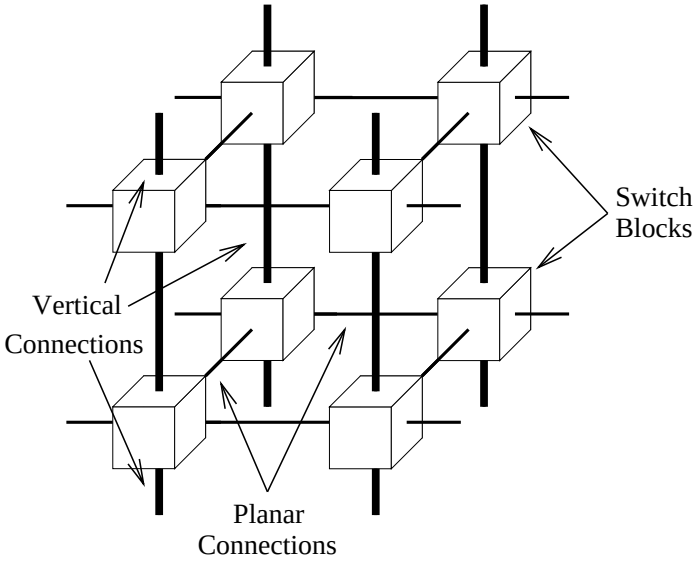


Figure 3.2: Conceptual model of the 3D-FPGA switch block configuration

In modeling the 3D-FPGA, a number of details must be addressed. For example, 2D-FPGAs are normally built with a uniform channel width. However, there is no reason to assume that the vertical channels of a 3D-FPGA must have the same channel width as the planar channels. Nor is it a given that every two vertically adjacent switch blocks must have a vertical channel between them. If the construction cost of a vertical interconnection is high, it may be worth providing fewer of them.

The structure of the switch block must also be considered. The switch block of a 2D-FPGA does not always provide the option of connecting any arbitrary set of incident channel wires, but only of certain subsets. This condition is still the case in the 3D architecture. Restricting the flexibility of the switch blocks increases the complexity of mapping circuits to the FPGA and decreases the flexibility of the structure. This has led to investigations of the proper balance between switch block size and routing flexibility for the 2D architecture [30, 94]. This research needs to be generalized to the 3D architecture.

3.2 Construction

Although this dissertation is primarily concerned with the theoretical advantages of the proposed architecture, it would be incomplete without consideration of the physical problems in its construction. There would be minimal interest in this discussion if construction of a 3D-FPGA is infeasible. While we argue below that a 3D FPGA can be built, this is not immediately obvious. Heat dissipation can be a problem, as the surface-to-volume ratio is greatly diminished. Similarly, vertical interconnections are currently difficult to implement.

Fortunately, the 3D-FPGA is not the first three-dimensional chip architecture to be proposed. Multi-chip modules, or MCMs, have been successfully implemented in a three-dimensional version known as an MVM-V [25]. Researchers have dealt with the same issues and constructed MVM-V devices.

MCM-Vs have been used to implement a 3D-FPGA. This was done at the University of Sheffield in the TriMorph project in 1996. While we are unaware of any literature that has been formally published on TriMorph, results were promising. However, TriMorph deviated from our model in one significant way: logic blocks could only be placed on the surface of the structure, as shown in Figure 3.3. While this provides some advantages over the 2D architecture, it does not compete with our proposal.

Our initial proposal of a 3D-FPGA motivated research at Northeastern University [60, 63, 68], where researchers succeeded in constructing a 3D-FPGA in line with our model. Instead of using the symmetric architecture, they used a structure known as the Triptych architecture [16, 47]. They produced an actual 3D chip with interior logic blocks which acts much the same way as our model. The stacking of chips was done using a technology developed at their lab [107], allowing them to place vertical metal interconnects, known as *interlayer vias*, anywhere on the chip. Technically, vias do not run vertically between switch blocks; instead switch blocks exist as planar entities with vertical connections attached to selected output pins, as depicted in Figure 3.4. This is an unimportant distinction, and the switch block configuration can easily be modeled as we have proposed.

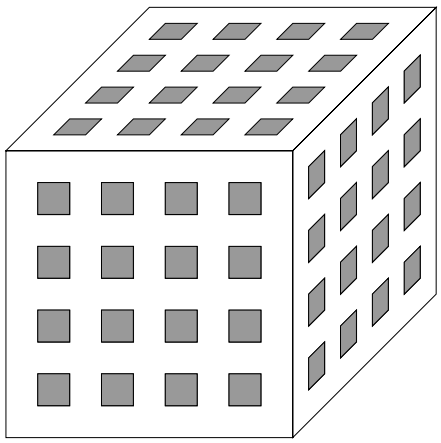


Figure 3.3: A TriMorph FPGA – all logic components are on the surface.

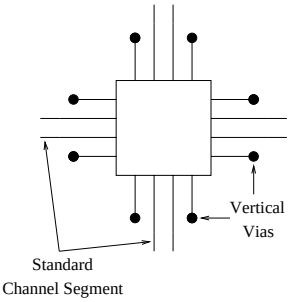


Figure 3.4: A TriMorph switch block. The switch block is identical to a 2D switch block, but several pins connect to interlayer vias – thus providing the vertical interconnections.

The research done at Northeastern University was motivated by our research, but does not overlap our work in any significant way. They have provided the answer to the one crucial question we were not equipped to address: Can 3D-FPGAs be created? While they did address the problems of mapping circuits to the architecture, it was an issue of only secondary concern for them, with no competitive results.

We can now continue with our motivation of the proposed architecture and the related design automation problems, knowing that 3D-FPGAs are a feasible technology.

3.3 Improvements over Standard FPGAs

While we have argued that the construction of a 3D-FPGA is feasible, we have not yet discussed the advantages of the proposed architecture. As discussed in Section 2.3, the major drawback to standard FPGAs is their speed and logic density. The technology that makes them flexible slows them down and consumes space that could otherwise be used for logic resources. By moving to the three dimensional architecture we can improve both performance and logic density, without raising the design complexity or losing any of the flexibility that make FPGAs worthwhile.

3.3.1 Speed

FPGAs suffer from major interconnect delay. In addition to the standard delay due to wire lengths, the technology that allows switch blocks their flexibility also slows down signals that transverse the structure. In total, interconnect delay can account for 70% of the clock cycle period [21, 99]. One way to deal with the delay is to place related pins closer together – thus reducing the connection distance. The restriction of placement to discrete points on the plane clearly limits our ability to do this. By allowing use of the third dimension, we increase our flexibility to minimize connection lengths.

Suppose we have a circuit requiring b logic blocks, and we have the option of configuring them in two or three dimensions, as depicted in Figure 3.5. Consider the average distance between points. With the two dimensional configuration, the average distance between any two points is $\frac{2}{3}\sqrt{b}$, while in the three-dimensional configuration the distance is $\sqrt[3]{b}$. In practice, for $b \geq 12$, the average distance between points is shorter in the three-dimensional configuration. An average shorter distance should lead to expected shorter net lengths, which means less interconnection delay. As any real circuit is unlikely to have a value of b less than 100, 3D-FPGAs are superior to 2D-FPGAs in this respect.

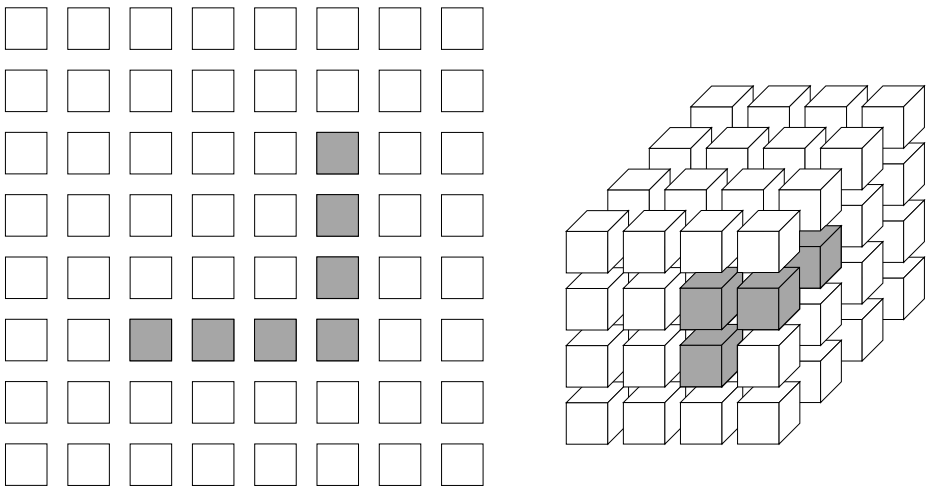


Figure 3.5: 2D vs 3D configuration of 64 blocks, with a path of average length highlighted for each structure.

3.3.2 Logic Density

The technology that allows FPGAs their flexibility comes at a second cost: size. By incorporating the potential for many different connections, we force an increase in the size of switch blocks – taking up area that could otherwise have been used for logic. Thus, there is less logic per chip, which means circuits require a larger area. By making use of the third dimension, we can reclaim some of this space.

Recall our goal of minimizing congestion. As discussed in Section 2.3, FPGAs are prefabricated with a uniform channel width. The smaller the width of the channel, the smaller the chip will be. When mapping a circuit to the architecture, we need to minimize the congestion, thus minimizing the required channel width. In the standard configuration, the limitation of four channels incident to any switch block restricts our ability to do this. In the three-dimensional architecture we increase this number, and thus increase our options. The average number of shortest paths between points increases, giving us more ability to route around congested areas without sacrificing net-length. As a result, we are able to

decrease congestion, thus reducing required channel width. Further, the amount of area dedicated to routing can be decreased, thus improving the logic density.

3.3.3 Other Advantages

While improved performance and logic density are the primary advantages of the 3D-FPGA architecture, it is worth noting other advantages of the structure. Unlike many of the proposed variations discussed in Section 2.3, the architecture does not increase design complexity or decrease flexibility. For example, the proposed addition of long wires to bypass switch blocks does improve the performance of some circuits, but it also increases the design complexity [32, 42]. The hierarchical FPGAs are faster but much more restrictive – some circuits cannot be mapped to them in an efficient way [1]. 3D-FPGAs are free from both problems; the mapping problem is no more complex than that of a standard FPGA, nor should any circuit suffer from being mapped to the 3D architecture.

3D-FPGAs reduce the need to partition large circuits between multiple chips, which results in gains in both speed and power consumption [60]. With the standard architecture, it is generally cost-effective to split large circuits up and provide inter-chip connections. However, these connections introduce huge delay, and it is a difficult (NP-complete) problem to partition the circuit in such a way as to minimize the number of connections required. It was shown at Northeastern that by moving to the multi-layered architecture, we alleviate this problem [60, 63]. As interlayer connections are considerably faster than inter-chip connections, we get considerable speedup. In addition, the reduced number of I/O pins and the long planar interconnections between FPGAs result in significantly less power consumption.

3.4 Further Remarks Concerning 3D-FPGAs

In this chapter, we proposed a model for the 3D-FPGA, showed that the concept is feasible, and discussed the potential advantages of the architecture. However, the discussion was

only speculative – it is still left to show that these advantages hold in practice. To do this, we must produce actual circuit mappings for the proposed architecture. Thus we must create actual design automation tools for 3D-FPGAs.

In the following chapter, we temporarily leave the discussion of 3D-FPGAs and move to the physical design of the standard FPGA architecture. In Chapter 5 we present a new design automation tool. For purposes of explanation and quality comparisons these tools are discussed in terms of the standard FPGA architecture. However, the tools can be easily generalized to the 3D architecture; all implementations are for the 3D-FPGA architecture with a 2D-FPGA treated as a degenerate case. In Chapter 8 we show the efficiency of applying our tools to the new structure and show that in practice our assertions do hold: 3D-FPGAs are superior to the standard architecture.

4

Design Automation for FPGAs

We have explained the basic structure of the standard and three-dimensional FPGA, and can now discuss our major concern: electronic design automation. How do we automate the process of implementing the circuit on an FPGA? In the following chapter we first discuss each of the traditional phases in the automation process, and the relevant work that has been done in the area. In Section 4.3 we discuss an inherent flaw in the standard methodology, and an alternative method to correct the problem.

Note that for ease of explanation, all concepts in this chapter are explained in terms of the standard symmetric FGPA architecture. The 3D-FGPA architecture is, for the moment, ignored. However, the concepts presented here easily generalize to the 3D architecture, as is discussed in Chapter 8.

4.1 Overview

Given a boolean description of some function, there are several tasks to be performed in order to implement the function on an FPGA. Logic blocks must be programmed, and pins on the logic blocks must be reserved for specific nets. Signal paths must be designated to connect nets, while switch blocks and connection blocks must then be programmed to implement these paths. All of this must be done with our objectives in mind: minimizing

the number of logic blocks, path lengths, and congestion. Computing a quality mapping of even the smallest circuits is difficult, and is a task that needs to be automated.

Traditionally, design automation is divided into four phases: technology mapping, placement, global routing and detailed routing [21], as depicted in Figure 4.1. In the technology mapping phase, we choose the set of functions to be programmed into the logic blocks. In the placement phase, we map each of these functions to a specific logic block on the FPGA. In the global routing phase we loosely specify each net's connection tree, assigned the net to channels, but not to exact wire-segments within each channel. In detailed routing we choose wire-segments with each channel for each net, subject to the restrictions imposed by the switch blocks.

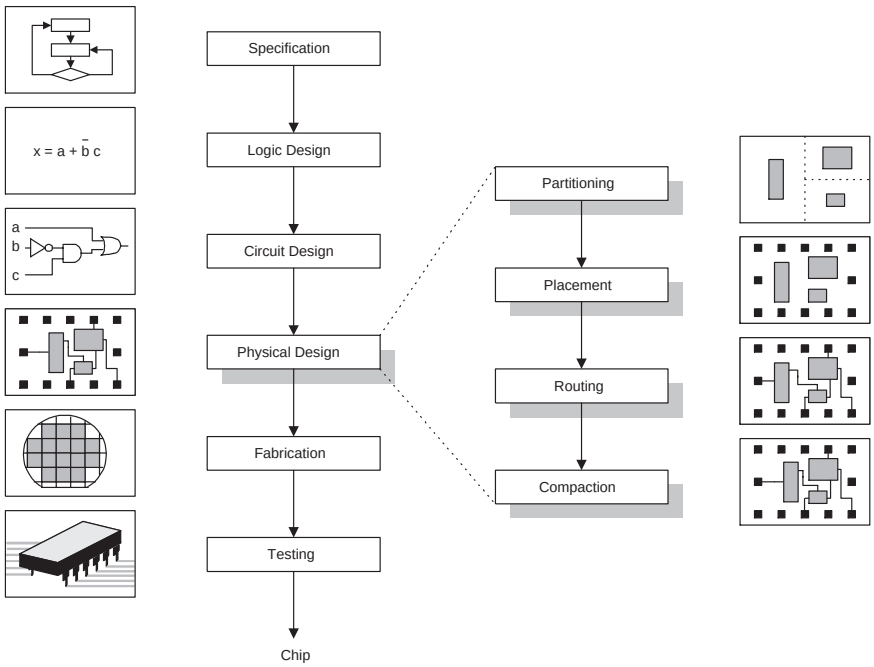


Figure 4.1: The stages of chip design and electronic design automation [64].

Traditionally each of these steps are performed sequentially: the output of one phase

becomes the input of the next. We do not believe that this is the best approach, and discuss an alternative in Section 4.3.

4.2 The Phases of Design Automation

Before we discuss our alternative approach, we need to explain the design automation stages, and the history of research relating to each stage, in more detail. In this section we present an in depth discussion of each stage, concentrating on those phases directly relevant to our research.

4.2.1 Technology Mapping

While the research in this dissertation does not concern the technology mapping phase directly, it does rely on the results of technology mapping tools. Hence any discussion would be incomplete without mention of this phase. The first serious technology mapping tool, Chortle, was developed at the University of Toronto in 1990 [40]. Chortle begins with a directed acyclic graph representation of a boolean network modeling the circuit, and groups interior nodes (each representing a simple function) such that each grouping can be implemented on one logic block. It must do this in such a way as to ensure that for each function group the number of inputs and outputs does not exceed the number of inputs and outputs allowed on a logic block. Given this constraint the algorithm attempts to minimize the total number of groups, thus minimizing the number of logic blocks used. Chortle makes use of a dynamic-programming algorithm, grouping and splitting nodes as needed during a search of the tree.

Since Chortle a number of more sophisticated algorithms, employing more sophisticated optimization functions, have been published in the literature [33, 34, 35, 36, 81]. By grouping functions in such a way as to lend themselves to a more efficient placement and a better routing, and allowing the mapping to take advantage of certain aspects of the architecture, the final circuit design can be much improved.

4.2.2 Placement

As our research deals directly with issues concerning the placement phase, we discuss this stage of the design automation process in more depth. Upon completion of the technology mapping phase, we have a list of functions to be mapped to logic blocks and a list of interconnections. In general terms, the goal is to map these functions to logic blocks in such a way as to minimize the tree lengths and channel width resulting from the final routing. Part of the difficulty inherent in placement is predicting the quality of the route that would result from a particular placement without overly complicating the placement phase.

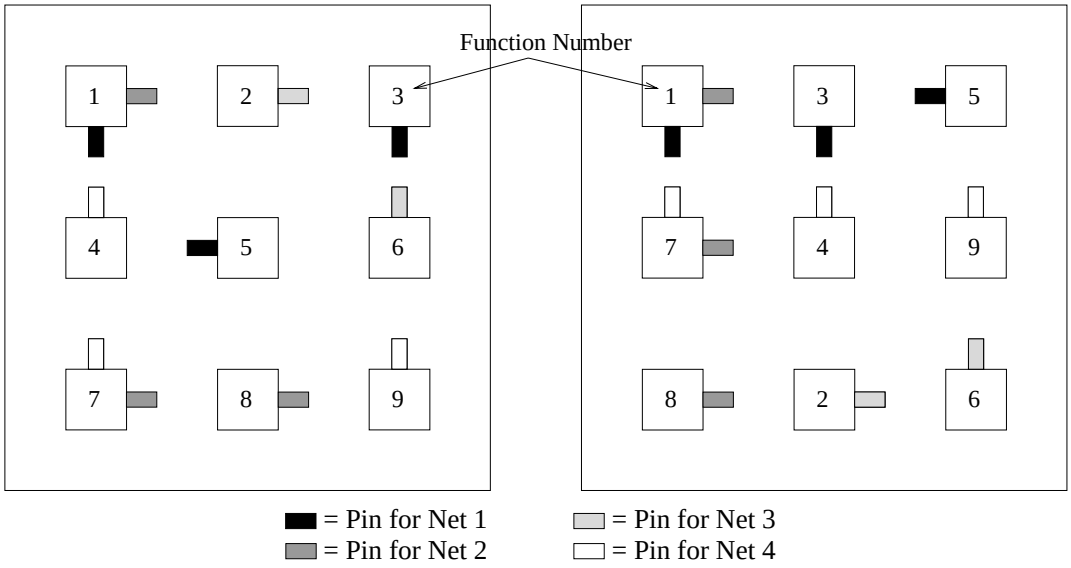


Figure 4.2: Two possible placements for a set of nine function-groupings on a 3x3 FPGA.

In Figure 4.2 we see two possible placements on a small FPGA. The first placement depicts a mapping of the nine functions to logic blocks without regard to the optimization criteria. Note that the mapping of a function to a block determines exactly which pins on that block must be used for which nets. In the second placement we have re-mapped the

functions in such a way as to minimize the lengths of the connecting trees. But in doing so, we made no attempt to minimize the number of nets per channel that will result from the final routing. In fact, our assignment does not even assure that the routings are possible given the switch block constraints.

Traditionally, placement research has been split into two approaches: *local search* strategies, such as simulated annealing an genetic algorithms, and *partitioning-based placement*. Local search strategies were for a long time considered superior to the partitioning-based placement. However, with the release of tools such as GORDIAN [56], the partitioning-based methods became competitive.

The first partitioning-based placement tools used the technique of *min-cut bisection* [18, 19, 46, 58]. With this technique the chip area is cut into two sections, and the function-groups partitioned between the two sections in such a way as to minimize the number of nets split between partitions. The process is then applied recursively to each partition, until the area is small enough that the placement can be done by some other method, as shown in Figure 4.3. Dunlop and Kernighan improved this technique with the addition of *terminal propagation* [38]. In terminal propagation, *virtual terminals* are designated for each net along the border of the partition regions to better reflect connections between regions – as will be detailed in the next section.

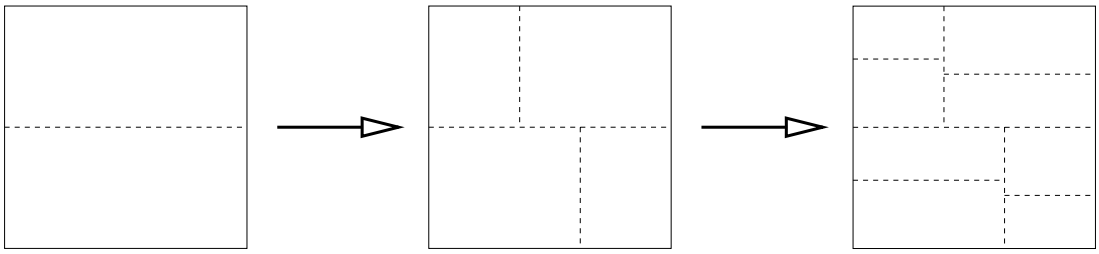


Figure 4.3: A recursive min-cut partitioning of a chip area.

These basic min-cut algorithms allow only placement, with routing to be done later. Suaris and Kedem generalize the min-cut approach to a *quadrisection*, in which the vertical

and horizontal cuts of a region are performed simultaneously [91]. Thus the partition is done with a "two-dimensional" view, allowing for a better placement in the plane. Combining this technique with terminal propagation, it is also possible to develop a course global route for each net as the blocks are placed in the quadrants, leading to the basic technique for simultaneous placement and global routing [92].

The quadrisection technique was generalized to larger grids by Mayrhofer and Lauther [67], who also introduced the notion of using minimum Steiner trees over the grid to define their optimization function. Bapat and Cohoon then developed the technique into a methodology known as *sharp partitioning* [12], leading to a full tool for the simultaneous placement and global routing of standard cell devices. Ganley implemented the method for FPGAs, leading to the *Mondrian* tool [7, 43] – the predecessor for the Spiffy tool discussed in Chapter 5.

While we do not address local search placement strategies directly, we compare the results of our tools against tools making use of these strategies. Most local search strategies are based on simulated annealing. One of the original tools for placement was Timberwolf, which used simulated annealing and a bounding-box metric to generate the placement, as well as a global route [86]. More recently, Betz and Rose produced VPR, which attempts to place blocks in such a way as to minimize a function of the bounding boxes of the nets [15], and is also capable of independently computing a route for the placement it produced. In a later paper VPR has been further improved, and geared towards timing-driven placement [66].

4.2.3 Global Routing

In global routing, the goal is to take the placement and compute a general route for each net. That is, to pick the channels each net will use without choosing the exact wire segments. The objectives depend on the application. Generally the user wants to optimize over two criteria. Congestion should be minimized by minimizing the maximum number of nets

assigned to any one channel. And propagation delay should be minimized by minimizing the size of the trees. Which of these criteria takes priority depends on the application, and is sometimes superseded by more specialized criteria, e.g., minimizing the length of certain critical nets, or minimizing clock skew.

In Figure 4.4, we see a placement on a small FPGA, and two possible global routes of that placement. Note that both global routes are optimized in terms of tree length, but the second route induces a channel width of two, while the first has no more then one net assigned to any channel.

There has been debate on whether global routing is a necessary step in the design of an FPGA [62]. Some tools skip the step [3, 8, 9, 42, 61]. However, there has been a fair amount published in the literature on global routing. As mentioned in the last section, the Mondrian tool performs placement and global routing simultaneously. The Maple-opt tool of Togawa, Sato and Ohtsuki was originally design to perform placement and global routing simultaneously, and was later expanded to include technology mapping [95, 96, 97, 98]. Other tools work independently on global routing, generally using graph-based techniques based on standard shortest-path algorithms [14, 15, 27, 29, 93].

4.2.4 Detailed Routing

Once the global router has finished its job (if used), a detailed router assigns each net to channel segments within the various channels used by the net. Only one net can be assigned to a particular wire segment within a given channel. It is here that the structure of the switch blocks must be considered. As switch blocks are limited in their allowed connections, it is the detailed router's job to ensure that a net is assigned to wires such that it can be connected through the switch blocks in a legal manner.

Depending on the quality of the placement and global route, it is possible that the router will not be able to create a legal route for the circuit. As neither the placement nor global routing tool typically take switch block structure into account, the tools may assign the nets

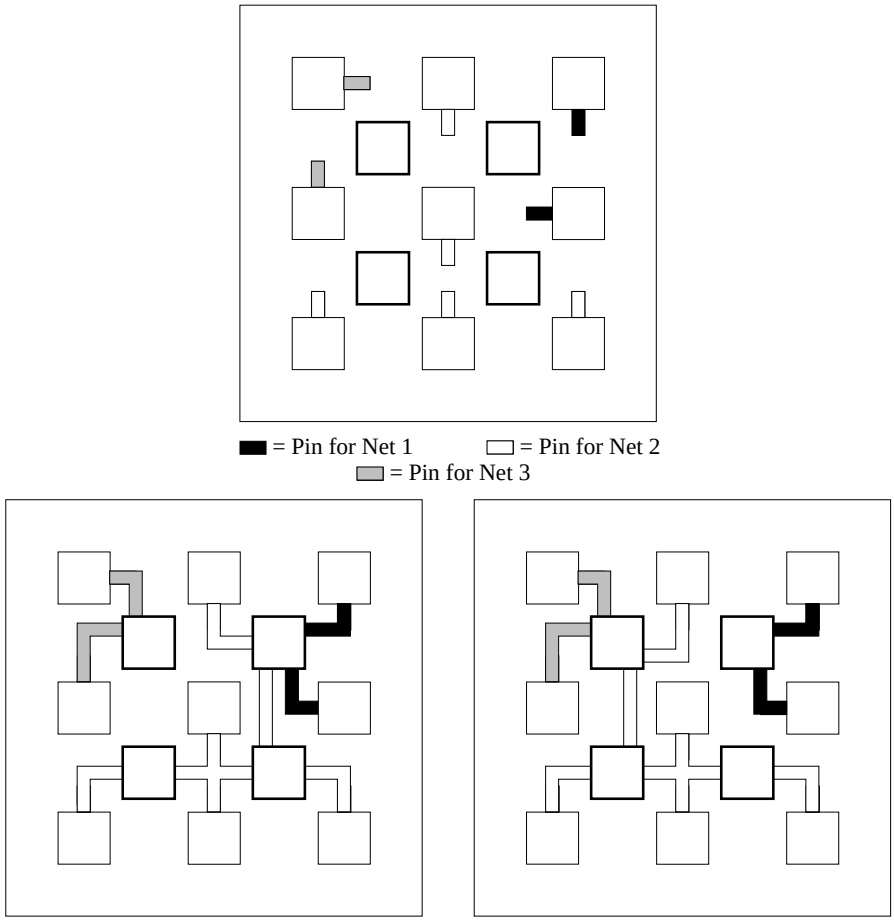


Figure 4.4: A placement on an FPGA of 3x3 logic blocks, and two possible global routings of that placement. The left routing will require a channel width of two, while the right a channel width of only one.

in such a way as to be impossible to connect all of them through the switch blocks. This problem can sometimes be alleviated by increasing the channel width, performing a "rip-up-and-reroute," or changing the global route of the nets [8, 9]. Frequently the detailed router is also left to incorporate special characteristics of the architecture, such as long wires, thus possibly improving on the global routes. So while the primary objective of a detailed router is to compute a feasible detailed route for each net, it still should optimize congestion and

net-length within the solution space of legal routings.

In Figure 4.5 we see an example of a global route, and a detailed route computed from it. Note that if the designer had prioritized channel width over net length in the detailed routing stage, the largest net could have been rerouted, sacrificing size to reduce congestion.

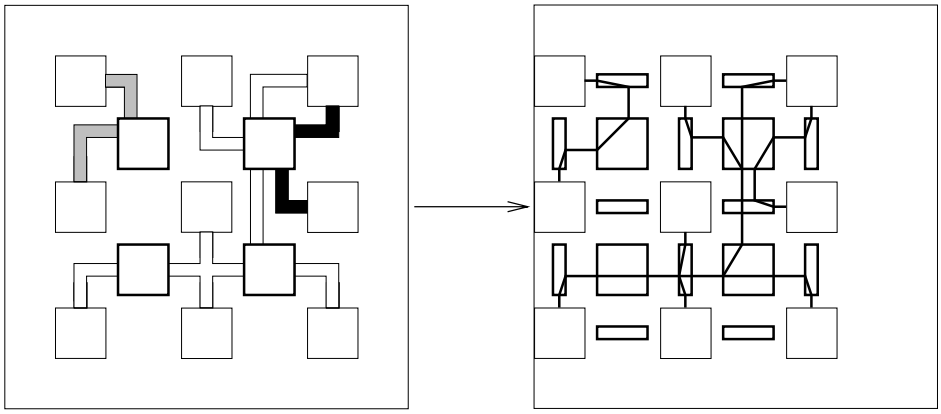


Figure 4.5: A global route for some circuit, and a detailed route computed from that global route.

While it is a goal of our research to perform detailed routing simultaneously with placement and global routing, creating a pure detailed router was beyond the scope of our research. While there are a number of routers in the literature [23, 24, 101], we used two routers produced at the University of Virgina: the router developed by Alexander and Robins [8, 9], and McCulloch's and Cohoon's *Upstart* routing tool. Each of these are graph based (hence trivially applicable to 3D-FPGAs as well as the standard architecture), and work net-by-net. For each net, the router attempts to route the net within its specified global route, then begins to expand the possible paths if this fails. Details of that expansion vary between the tools.

4.3 Sequential versus Simultaneous Phases

Traditionally, the four phases are done sequentially; the output of one phase becomes the input of the next [88]. The reasons behind this boil down to simplicity: it is considerably more complex to deal with the problems together then it is to solve each independently. However, as the quality of the result of any stage is directly dependent on the output of the preceding stages, there is some loss in not allowing the later stages to influence of the earlier stages. In this section, we discuss the idea of performing the stages simultaneously, and review the relevant research.

4.3.1 Motivation

While performing the stages sequentially simplifies the mapping process as a whole, it comes at a cost. In splitting the tasks up, we are implicitly relegating certain decisions to later stages. For example, switch block structure is generally considered only in the detailed routing phase – thus simplifying the earlier stages. However, as the global routing stage ignores the constraints imposed by the switch block structure, the global route that is produced will frequently be infeasible, requiring the detailed router to redo some of the global router's work. Similarly, placement algorithms generally measure the quality of their placement based on the spread of the nets – trying to get the pins of each net as close together as possible. Because the placement metric often does not take channel width into consideration, the global router will frequently have a more difficult time minimizing congestion then it would have with a slightly different placement.

Consider the example in Figure 4.6. Initially we see two possible placements of a given net – the top placement differing by the bottom in that we have switched the lower rows of blocks. Note that if we are taking only net-lengths into consideration, these placements are considered equivalent. The length of each connecting tree will remain the same. Thus the placer has no reason to pick one placement over another.

Next we see a global route for each placement. At this stage, we can see the top route

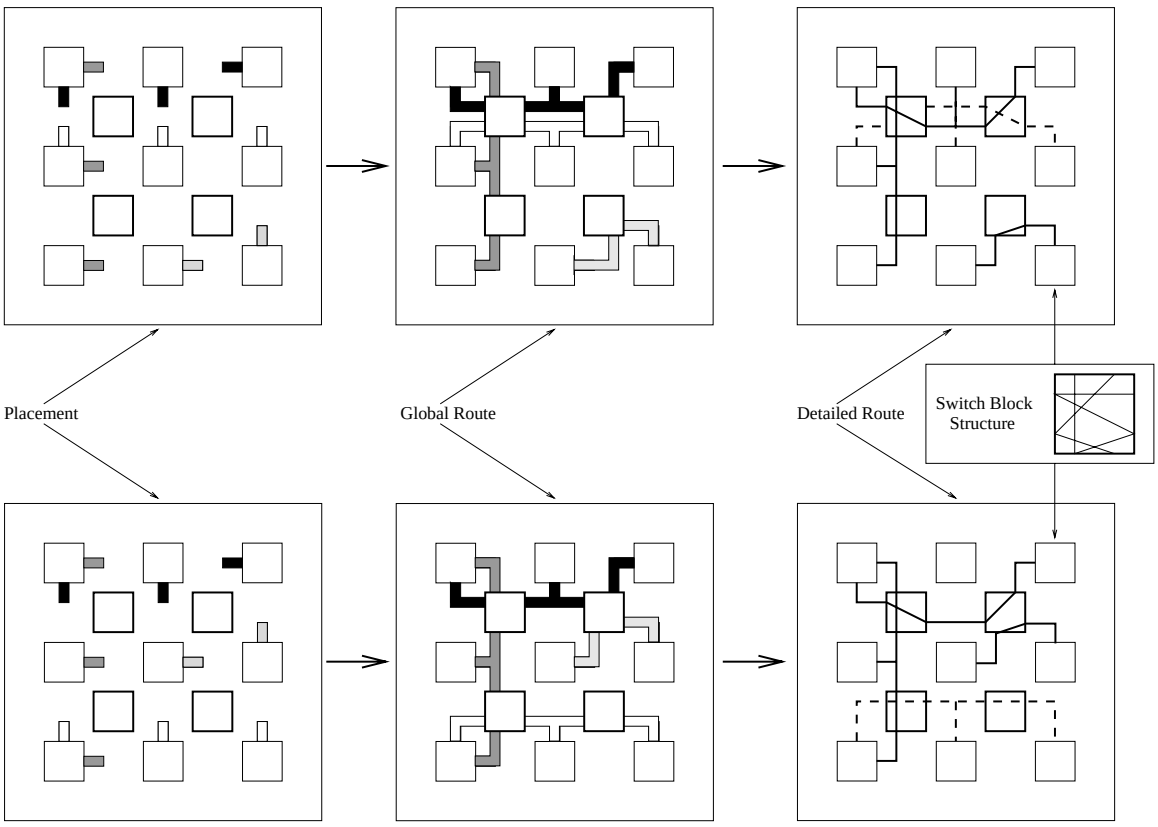


Figure 4.6: Two possible placements of equivalent quality, and the global and detailed routes that follow. Note that the top global route is of lesser quality, and did not lead to a feasible global route given the structure of the switch block shown below.

is inferior to the bottom – it requires a channel width of two, where the other requires a channel width of one. The global router will produce a route for the placement it is given; it does not have the option to choose between these two routes, or to modify the placement. In a strictly sequential system, the placer can decide which of the placements will be used *without considering channel width.* Thus there is no way to guarantee that the superior placement is used.

Now assume that the switch blocks for the FPGA in question are structured as shown, with the interior lines designating potential pin connections, and we see that the one place-

ment cannot be routed. The upper left block does not have the capacity to carry two signals horizontally. Yet because the placement phase does not consider the switch block structure, it does not know to avoid that placement. For the same reason the global router cannot spot the problem. Thus we do not identify the problem until the detailed routing phase, costing us a significant amount of time without providing any solution other than to start from scratch.

Clearly it would benefit the final result if constraints such as switch block structure and objectives such as channel width minimization were addressed in earlier stages. One approach towards this goal is to perform the stages simultaneously and interactively, so decision made in the "later" stages can affect decision made in the earlier stages without necessitating the duplication of the earlier stages work.

4.3.2 Work on Simultaneous Phases

The first method in which two phases were combined came from the introduction of terminal propagation [38]. This, coupled with the quadrisection technique [92] and generalized to a larger grid [67] led to the sharp partitioning as discussed in Section 4.2 [12].

In Figure 4.7 we see a chip area that has been quadrisected, with the pins of a two-terminal net assigned to quadrants one and three. When we recursively apply the quadrisection to each quadrant, there is no indication where each of the nodes should be placed within a quadrant. Hence there is no reason they might not be assigned to opposite ends of the chip – inducing the worst connection length possible. However, when we assign virtual terminals along the partition lines – each virtual terminal being treated as a node in every partition it touches – the partition algorithm will be penalized for placing terminals in partitions not containing the virtual terminals.

Thus we have integrated a portion of the global routing phase with the placement phase. By assigning the virtual terminals, a part of the global route is developing and is able to influence future placement decisions. Bapat and Cohoon refined this methodology with

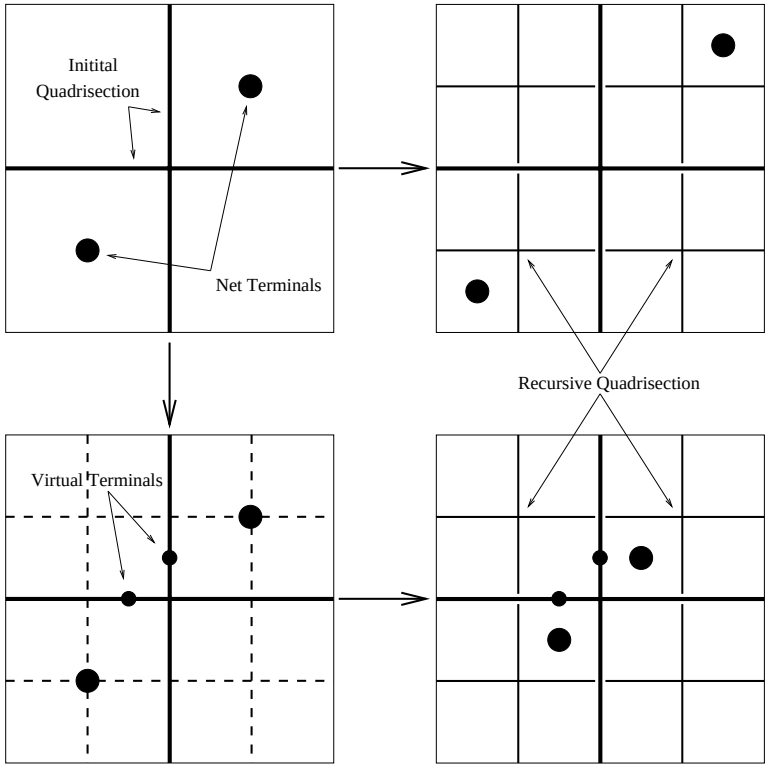


Figure 4.7: A quadrisection min-cut quadrisection of one net, followed by a possible partition without terminal propagation, and one in which virtual terminals are assigned.

their introduction of sharp placement [12], shown in Figure 4.8. Here we see how a two-terminal net might be partitioned within the three-by-three grid. In the second stage we pick the route the net will take between partitions, thus giving us a coarse global route, and a selection of virtual nodes to propagate the information to future recursions. In step three we apply the algorithm recursively, treating virtual nodes as actual nodes. Thus it can be seen how the assignment of virtual nodes leads to an overall global route.

There have been several other projects with the aim of integrating the phases into one simultaneous phase. Schlag, Kong and Chen make a first attempt with their tool for routability-driven technology mapping [85], a tool that does not perform the phases

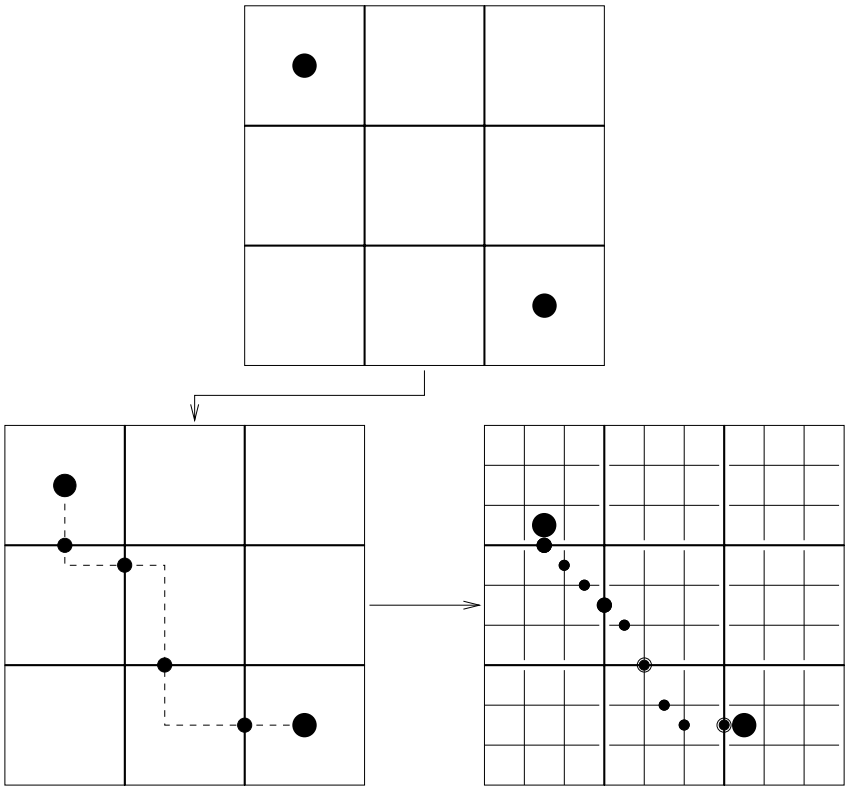


Figure 4.8: A sharp-partition of one net, followed by the selection of a global route between partitions and the assignment of virtual nodes, then the algorithm to one level of recursion.

simultaneously, but does take routing constraints into consideration during the technology mapping phase. A first attempt to actually perform technology mapping and placement together was proposed by Chen, Tsay, Hwang, Wu and Lin [31]. The Maple tool created by Togawa, Sato and Ohsuki originally combined placement and global routing, but later improved on the algorithm by Chen et. al. to combine technology mapping as well in the tool Maple-opt [96, 97, 98]. Nag and Rutenbar have also published several results for simultaneous placement and routing [69, 70, 71], as have Nakatake, Sakanushi, Kajitani and Kawakita [72].

Less has been done with the integration of detailed routing. Frequently global routing

is not performed, delegating the problems of global routing to the detailed router [3, 8, 9, 61]. The VPR tool of Betz and Rose does explicitly perform the two simultaneously [15]. However, there is no tool in the literature that performs detailed routing and placement simultaneously. The tool Gambit presented in Chapter 7 is the first.

4.4 Summary

We have now explained the problem and objectives of electronic design automation, explored the relevant work done in the area, and explained our goal: to create a tool that performs the stages of design automation simultaneously. While complicated in nature, research has indicated that such tools will lead to considerably better circuits designs then does the sequential methodology.

In the next chapter we follow up on the work of Bapat, Cohoon and Ganley [12, 7, 43] to create Spiffy: a tool for the simultaneous placement and routing of three-dimensional FPGAs.

5

Spiffy: A Tool for the Simultaneous Placement and Global Routing of FPGAs

In this chapter we present Spiffy, a tool for the simultaneous placement and global routing of FPGAs. Spiffy is also the first tool to perform these tasks for 3D-FPGAs. In Section 5.1 we present the necessary background and a brief overview of the methodology. Section 5.2 introduces the concept of a *thumbnail* – the underlying basis for Spiffy's routing capabilities. In Section 5.3 we discuss the data structures that contribute to the success of Spiffy, in Section 5.4 we present the Spiffy algorithm in detail, and in Section 5.5 we give an asymptotic analysis. We present our experimental results in Section 5.6.

5.1 Overview and Background

In Section 4.3 we outlined the sharp methodology of Bapat and Cohoon, using a recursive application of sharp partitioning in combination with terminal propagation to simultaneously develop a placement and global route [12], which was implemented specifically for FPGAs in Ganley's Mondrian system [43]. Spiffy is based on the Mondrian system, though the specifics have been extended and improved. As a result, Spiffy is a superior tool to Mondrian, producing superior results in significantly less time.

Like all placement tools, Spiffy begins with the output of the technology mapping stage: a list of functions which must be mapped to logic blocks, and the list of net interconnections. The tool's output is a placement and global routing for the circuit. Spiffy employs a divide-and-conquer algorithm that works by creating an initial placement and loose-global route, then refining the current results in each recursion. Within each recursion of Spiffy there are three major activities:

- *Partitioning:* In a given recursion, Spiffy partitions the area being routed into a grid and assigns logic elements to grid-partitions in such a way as to minimize the "spread" of the nets, i.e., this distance between terminals of a given net. Each grid-partition will then serve as a sub-problem in a future recursion. There is no theoretical limit on the size of the grid, but in practice it is computationally infeasible to use a grid more refined than a 3×3 partition.

- *Route Selection:* Having placed each logic element into exactly one partition, Spiffy now decides the path that the nets will take between partitions. The objective is to minimize the congestion by spreading the paths evenly across the partitions, while still using minimum source-to-sink paths for each net.

- *Virtual Terminal Assignment:* While we have assigned each net a route between partitions, we do not know exactly where a given route will cross a given partition-line. Spiffy assigns virtual terminals for each net along these partition lines, specifying the switch block locations at which the nets will cross the lines. Virtual terminals enable the algorithm to consider each partition independently.

In our implementation, the recursion terminates when it reaches a partition containing one logic block, at which point nets are routed around the block to complete the overall routes.

These activities are depicted in Figure 5.1. In Figure 5.1(a) we see an FPGA with a 3×3 grid placed on it, each logic element lying in exactly one grid partition, and each partition line cutting through a series of switch blocks and connection blocks. In Figure

5.1(b) logic elements have been assigned to partitions. The net shown in the diagram has been split between six partitions. In Figure 5.1(c) we see the inter-partition route chosen for that net, and in Figure 5.1(d) the virtual terminals along each partition line have been chosen. In Figure 5.1(e) we see a solution for each partition, the union of which gives our overall solution in Figure 5.1(f).

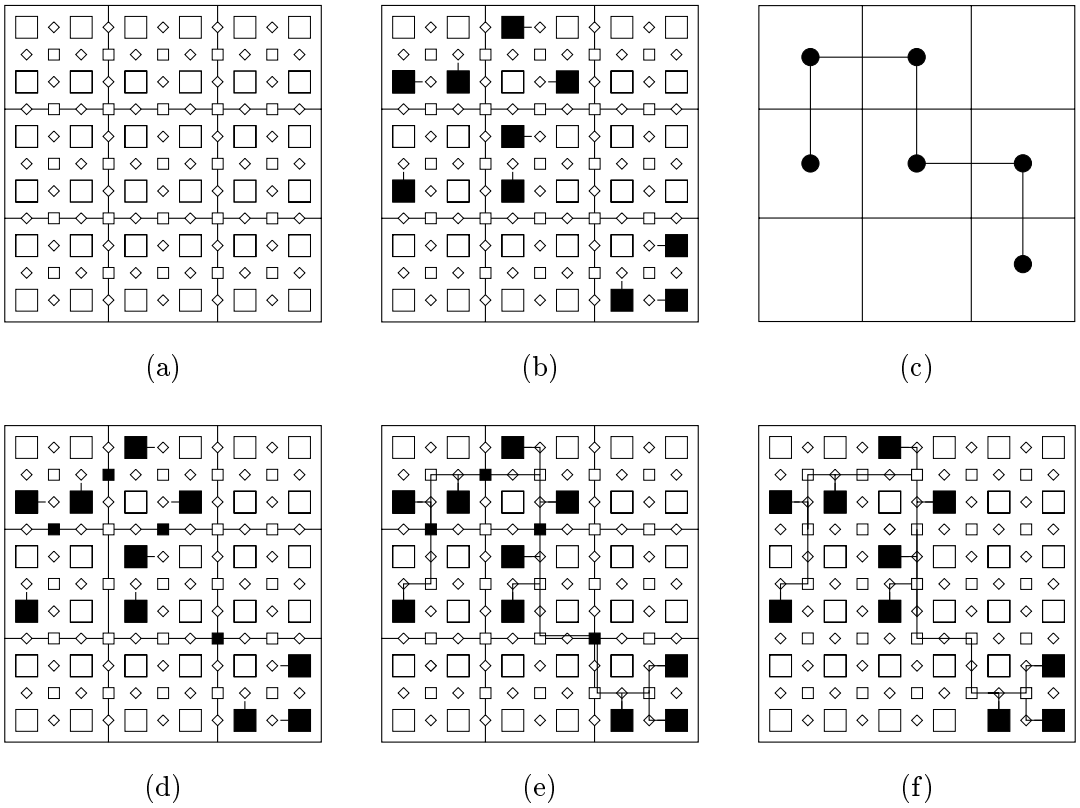


Figure 5.1: An illustration of the steps of Spiffy on a single net. (a) The FPGA is partitioned into a three-by-three grid, with each logic bock lying in exactly one partition, and each partition-line consisting of a series of switch blocks and connection blocks. (b) The logic blocks are partitioned between regions. (c) A route between regions is picked for the net. (d) Switch blocks along the global route are assigned as virtual terminals. (e) Each partition is solved recursively. (f) The union of the global routes is the final solution for this net.

In Section 5.4, we discuss the algorithm in more detail. However, preceding that discussion we first examine the graph constructs and data structures that contribute to its efficiency.

5.2 Thumbnails

Our graph construct discussion begins with *thumbnails*. A thumbnail is related to the concept of a *minimum rectilinear Steiner arboresence*.

Definition 5.1 *Let G be a rectilinear graph, N be some subset of the nodes of G, and $s \in N$ be the node that is designated the source node of N. A* **minimum rectilinear Steiner arboresence** *is a connected subgraph of G containing all nodes in N such that the source-to-sink path length of the tree is minimized for every path, and the length of the tree is of minimal length over all such trees.*

Note that any nodes outside of N used in the tree are referred to as *Steiner points*.

Figure 5.2 illustrates a minimum Steiner arboresence for a sample graph and point set. Using the gray node as the source and the black nodes as sinks, we first see three different minimum Steiner arboresences for the point set – each containing two Steiner points in addition to its terminal points. Note that the fourth tree, while smaller than the other three, is *not* a minimum Steiner arboresence as it does not contain a path of minimal length for every source/sink pair. The concept of Steiner arboresences has been long studied [50], and the problem of finding them was recently shown to be NP-complete [89].

Next, for a partitioned chip we define the partition graph of a chip:

Definition 5.2 *Given a chip area partitioned in a grid pattern, the* **partition graph** *of that chip is the graph containing one node for each partition, with two nodes being adjacent if and only if the corresponding partitions share a common boundary.*

We depict a sample partition graph in Figure 5.3. Note that as the partitions are in a grid pattern, the partition graph must be rectilinear.

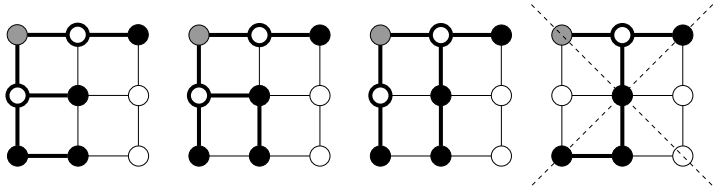


Figure 5.2: Three minimum Steiner arboresences for a given point set, with the gray node representing the source. Note that while the fourth graph is smaller, it does not contain a minimum-length path for every source/sink pair and hence is not a minimum arboresence.

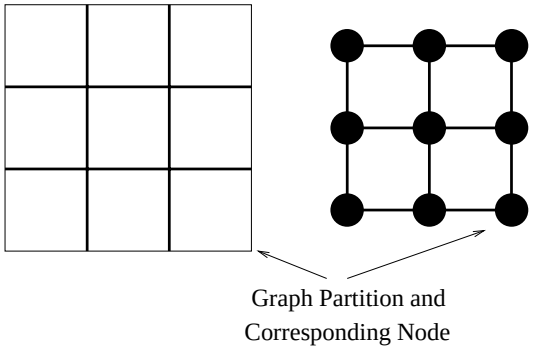


Figure 5.3: A three-by-three partition of a graph and its corresponding partition graph.

Having defined minimum rectilinear Steiner arboresences and the partition graph, we can now define the concept of a thumbnail:

Definition 5.3 *Let C be a chip with a partition grid, a placement of a circuit on the chip, and some net ν. Let $\Psi = (V, E)$ be the (rectilinear) partition graph of the chip, let $V' \subset V$ be the set of all nodes such that ν has a pin occupying the corresponding partitions, and let s be the node corresponding to the partition occupied by ν's source pin. A* **thumbnail** *of net ν is a selected minimum rectilinear Steiner arboresence of Ψ with respect to V' and source node s.*

In Figure 5.4 we see a sample net placement and four possible thumbnails for that net. Thumbnail evaluations will form the basis of Spiffy. They allow us to model both the spread of the nets as they are placed in partitions and the route a net takes between partitions.

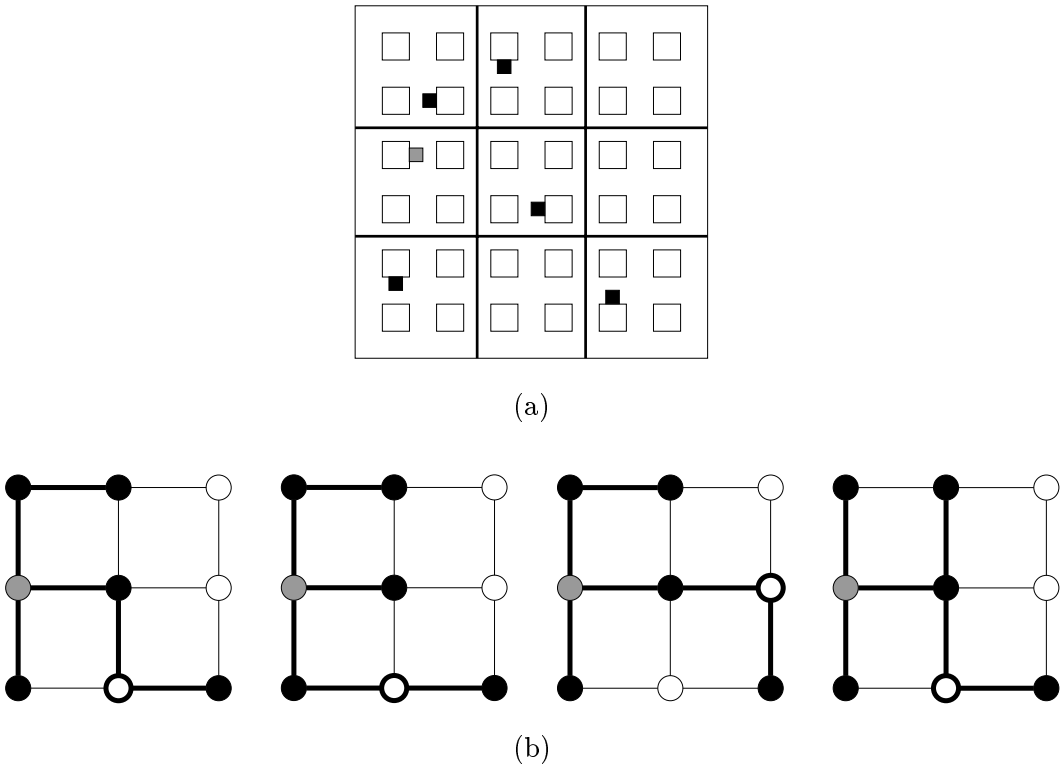


Figure 5.4: A placement of a net and four thumbnails for that placement.

Note that in using Steiner arboresences, we are orienting the tool towards minimizing source-to-sink paths, thus creating a placement-driven router. If we were interested only in minimizing net-length, we could instead use minimum Steiner trees, i.e., the smallest tree connecting a given point set. In the example of Figure 5.2, only the fourth graph would be

a minimum Steiner tree of the point set. While our experiments are done mostly in terms of the arboresences, Spiffy is configured to handle either model.

5.3 Data Structures

Spiffy was developed in an object-oriented style. In particular, a number of its structures are built on top of the Standard Template Library [10]. We discuss the data structures necessary for the representation of our problem, followed by the other data structures needed in the operation of our algorithm.

5.3.1 FPGA Structural Representation

The FPGA is represented by several persistent objects. These objects are created in the initialization of the program as specified by the input.

- **Chip:** A chip is a three-dimensional array of logic elements or blocks. Given a location, the block at that location can be returned in constant time. Blocks on the chip may change location in constant time.
- **Block:** A block is a set of pins partitioned into six directions, or sides, numbered clockwise around the block. The ith pin of the block can be returned in constant time, as can the ith pin of a given side, e.g., the 5th pin of the north side. Each block's location on the chip can also be returned in constant time. Note that a block can represent a logic block, switch block or connection block of an FPGA; which can be determined from its location in constant time.
- **Pin:** A pin exists on exactly one block and has at most one net mapped to it, identified by the net's index. The location of the pin's block can be found in constant time, as can the identity of the pin's net. A net can be added to a pin, removed from a pin or changed on a pin in time logarithmic in the number of pins on the affected pin's block.

- **Net:** A net consists of three sets:

  Terminal Pins: The set of all logic block pins used by the net. The set can be returned in time linear in the size of the set. After its initial creation, pins are not added or removed, though they may move location on the chip in constant time.

  Steiner Pins: The set of all switch block and connection block pins used by the net. The list can be returned in time linear in its size. Pins may be added or removed in time logarithmic in the size of the list.

  Channels: The set of channels to which the net has been assigned. The list of channels can be returned in time linear in its size, and channels may be added or removed in time logarithmic in its size.

  In addition, the net contains a source pin, a terminal pin designated by the input, and a source tree: a directed acyclic graph such that the root represents the source pin, the set of leaves represents the remainder of the terminal pins, and the interior nodes represent the Steiner pins on the paths from the source to the terminals. Given any node of a source tree, the node's parent and set of children may be returned in constant time, and the node may be moved to any other position in the tree in constant time.

At the termination of the algorithm, the structure of the source tree directly reflects the global route of its net. If we begin at the source node of a net's source tree and trace some path to a sink, the switch block pins represented by the source tree's interior nodes form a path from the source pin to the sink node's terminal pin. The switch block pins are discarded, but the switch blocks to which they correspond define the global route. We see this construction depicted in Figure 5.5, with a net on the left and its corresponding source tree on the right. For simplicity, we have combined all nodes corresponding to the same switch block.

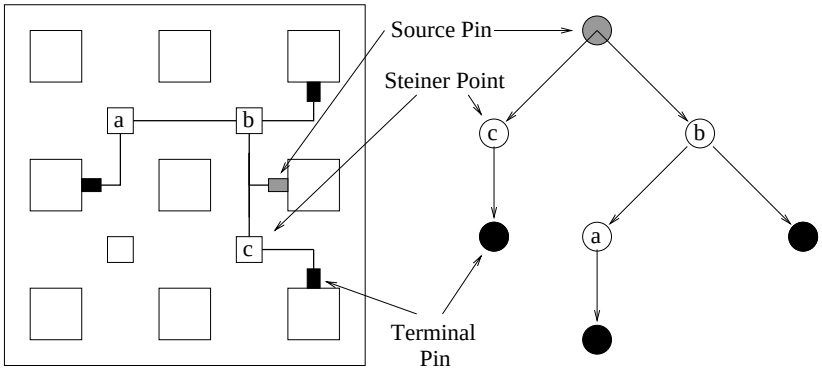


Figure 5.5: The global route of the net, and the corresponding source tree. For simplicity, we have combined all nodes representing pins of the same switch block into one switch block.

Problem 5.1 *Given a chip with blocks, pins and nets initialized as indicated by the input, rearrange the blocks and create a global route for each net based on this arrangement such that:*

- *The maximum number of nets using any one channel is minimized.*
- *The maximum source-to-sink path length averaged over each net is minimized.*

5.3.2 Other Data Structures

While the above data structures are sufficient to solve the problem, additional structures and classes are necessary for the efficient performance of the algorithm.

- **Chip Portions:** A chip portion object records the boundaries of a cubic portion of the chip. Each chip portion contains the following information:

  The boundaries of the portion in each dimension. These are accessible in constant time.

  The source pin of each net for that portion, i.e., the pin where the net enters the portion, or the source pin of the net if it is in the portion. The pin is accessible

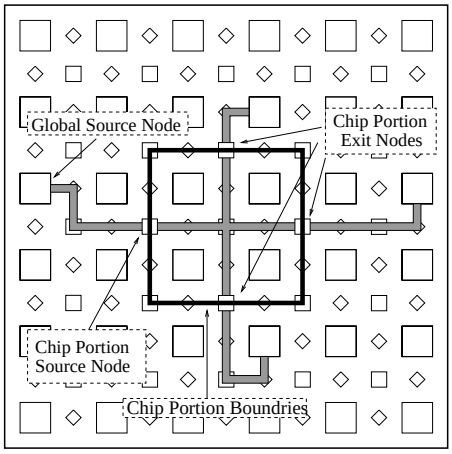


Figure 5.6: An illustration of an FPGA wit one chip portion boundary highlighted, and one net that is routed through the portion. The associated object would contain information defining these boundaries with respect to he chip, as well as net net's source node and exit nodes.

for a given net in constant time, as is the corresponding node in the net's source tree.

The exit pins of each net, i.e., the pins where the net leaves the portion. Exit pins differ from the entrance pins because we are viewing the route as a directed tree, with the net's global source acting as the root. The list of exit pins is accessible in time linear in its size.

A sample chip portion is depicted in Figure 5.6. Spiffy is designed to operate on a chip portion object, thus simplifying recursion. Its initial input will be the chip portion whose boundaries are comprised of the chip's boundaries. When operating on any portion, it uses the partition grid to define new sub-portions, placing each of these in a partition queue (or stack). This when it has completed it task with respect to one portion, it takes the next portion from the queue, thus emulating a recursion call. Note that with the queue we are actually performing a breath-first search, while a stack structure is equivalent to a depth-first search.

- **Channel Matrix**: This object indicates which nets have been mapped to each channel. The object is implemented as an array of sets. Given two adjacent blocks with connecting channel ι , we can add a net to ι or remove a net from ι in time logarithmic in the number of nets currently mapped to ι. We can determine the number of nets mapped to ι in constant time.

- **Connecting Map**: For a given net and any two blocks, the connection map determines whether the two blocks are connected by the net's global route. This information is necessary to prevent loops in the route. It is implemented as a standard union-find structure, with the find and union operations taking close to constant amortized time.

- **FPGA Graph:** This graph represents the route of each net on the FPGA. Each node in the graph represents one block in the route of the FPGA. Nodes are adjacent if and only if the net is routed between the intervening channel in the case of a switch block and connection block, or on a pin in the case of a logic block and a connection block. This graph is created at the end of the algorithm, and allows us to prune routes of unnecessary paths, check for disconnected net routes, and compute statistics such as tree length.

5.3.3 Steiner Tree Representation

During the course of the algorithm, it becomes necessary to calculate minimum Steiner arboresences. Because calculating arboresences is an NP-complete problem, we maintain a list of all possible arboresences for all possible point sets in our grid to retain tractability. Thus we need an efficient way to store node sets of the graph and the arboresence structures.

Since the partition graph is never modified, we can create a representation by numbering each node and each edge. Let v be the number of nodes in the partition graph Ψ. By numbering the nodes from 0 to $v-1$, we can encode any node set with a binary number

such that the ith bit is 1 if and only if node i is in the set. Access to the nodes is then logarithmic in the number of nodes. By numbering the edges, we can specify edge sets – and subgraphs – in a similar manner. This leads us to our final data structures:

- **Steiner Sizes:** An array of integers such that the ith element is the size of any minimum Steiner arboresence corresponding to the node set encoded by i.
- **Steiner Trees:** An array of arrays of arrays of integers such that the jth element of the ith element is an array of the subgraph encoding of the minimum Steiner arboresences corresponding to the point set i with node j as the source.
- **Incident Edges:** An array of integers such that the ith integer is an encoding of the edges incident to node i.
- **Incident Nodes:** An array of integer pairs such that the ith pair consists of the two nodes to which edge i is incident.

The structures are initialized from a pre-computed file and allow us to do several things:

- Given a point set and a source node, we can compute the size of the minimum Steiner arboresences for that point set in constant time given that the number of nodes fits in a single word. Given the size of our graphs this assumption is valid, and is made in the following definitions as well.
- We can examine all minimum Steiner arboresences of a point set in time linear in the number of such arboresences and determine the number of such arboresences in constant time.
- Given a particular arboresence, we can conduct a breadth-first or depth-first search on the tree in time linear in its size.

Note that while we have been assuming that we wish to minimize source-sink paths as discussed in Section 2.3, the structures are flexible enough to minimize tree size if that

metric is favored. In that case, we modify the structures to list all minimal Steiner trees instead of arboresences.

5.4 The Spiffy Algorithm

We now describe Spiffy in detail. In this section we will present each phase of the algorithm, and describe our approach to the problems posed by those phases.

5.4.1 Notation and Problem Size

In describing and analyzing the Spiffy algorithm, we will use the following notation:

- n denotes the number of nets for our problem instance.
- b denotes the number of blocks on the chip.
- x, y and z denote the dimensions of the chip.

  $$b = x \cdot y \cdot z.$$

- p, q and r denote the dimensions of the partition grid.
- $\Psi = (V, E)$ denotes the partition graph, where V is the node set (with $v = |V|$) and E is the edge set.

  $$v = p \cdot q \cdot r$$

  $$|E| = x \cdot (y-1) \cdot z + (x-1) \cdot y \cdot z + x \cdot y \cdot (z-1)$$

- $\Lambda(V', s)$ denotes the number of minimum rectilinear Steiner arboresences in G on the point set $V' \subseteq V$, with source $s \in V'$.
- $\Lambda(\Psi)$ denotes the maximum possible value of $\Lambda(V', s)$ over all node source / sink pairs in Ψ.

In terms of asymptotic analysis, the size of the partition grid is *not* considered an aspect of the problem size. Due to the large number of thumbnails that must be stored on disk, it is impractical to use grids larger than 9 partitions. Hence we can assume that p, q and r as constant with respect to the problem size, and perform our analysis in terms of the two factors which do grow: the number of nets (n) and the number of blocks (b).

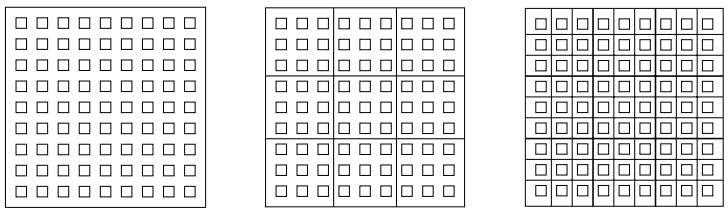
5.4.2 Initialization

Initialization of the data structures is a simple task. The arboresence-related data structures are read directly from a pre-computed file. The chip, blocks, pins and nets are also read directly from the input files produced by a technology mapping tool. Each block object must be created, each pin object must be created, and each net object must be created with its proper terminals. In initializing the structures, we create an arbitrary placement. Each logic element has been assigned to a block, and it is now Spiffy's job to rearrange the block positions.

We also pre-calculate all partitions to which the algorithm will be applied, define them each with a chip portion object, and order these as we wish to address them. The first chip portion consists of the entire chip, while the next v portions consist of the v sub-portions into which the chip is divided. The following v^2 portions consist of the next level of sub-partitions, and so on until partitions of size one are reached, as diagramed in Figure 5.7.

Note that though these chip portions are being pre-calculated (a concession made for ease of the *template smoothing* technique introduced in Chapter 6), we transverse them in a breath-first search. As long as each portion is examined after its parent, we are not bound to a breath-first search. We experimented with a depth-first search, but the breath-first search gives better results and is more amenable to the template smoothing technique.

When initializing the chip structure, we need to run through the list of blocks once – requiring $O(b)$ time. In constricting the net structure, we run through the list of nets,



(a) (b) (c)

Figure 5.7: The ordering of the chip portions for a 3x3 grid on a 9x9 FPGA. (a) The first partition – the entire chip. (b) The next set of partitions – the first level of recursion. (c) The next 81 partitions, which are the base cases of the recursion.

adding terminals as needed. As adding a terminal to a net requires time logarithmic in the number of terminals already in the net, the entire process requires a worst-case time bound of $O(n \log b)$, but an average-case time bound of $O(n)$.

Initialization of the arboresence-related data structures depends on the size of the partition graph, but this is not considered part of the input size. Therefore, in terms of an asymptotic analysis we can consider this to be a constant time operation.

Pre-computing the chip portions is the most asymptotically expensive operation of the initialization stage. While each portion can be initialized in constant time, the number of portions increases exponentially with the depth, for a total of $\sum_{i=1}^{\log_v b} v^i$ partitions. With each of these partitions taking constant time, the total time required is $\frac{v^{\log_v b+1}-1}{v-1} = O(b)$. Thus we have our complexity analysis for the initialization phase:

Complexity Analysis 5.1 *The initialization phase of the Spiffy algorithm runs in*

$$O(b) + O(n \log b) + O(n)$$

worse case time and

$$O(b) + O(n)$$

average case time.

5.4.3 Partitioning

5.4.3.1 Problem Definition

Given the chip portion under consideration and a partition of that portion, our first objective is to rearrange the blocks within the portion in order to group the pins of any net within a minimal number of partitions, thus keeping the interconnect distance of the net small. It is not always possible to do this; some nets must be split. We prefer to keep these split nets in adjacent partitions when possible. We illustrate this circumstance in Figure 5.8, where an arbitrary placement of two nets is shown, and two possible block rearrangements. While both placements minimize the number of nets that is split, the right placement places the split net in adjacent partitions.

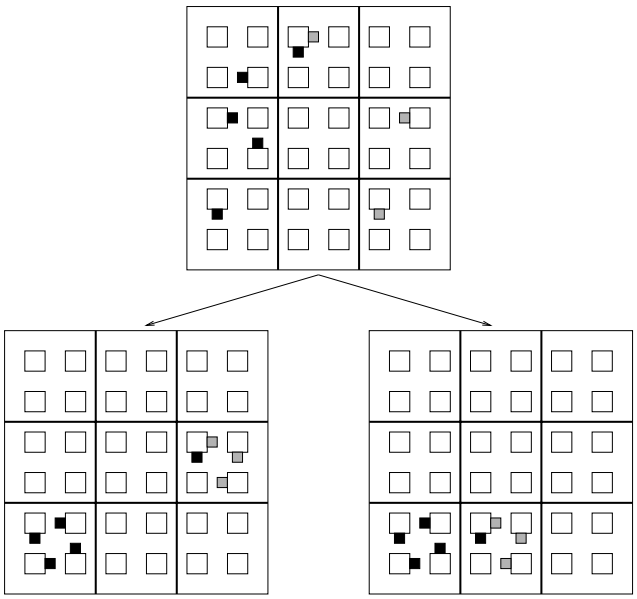


Figure 5.8: An initial placement of two nets, and two possible rearrangements. The left picture is superior to the initial placement because the nets are grouped together as much as possible. The right picture is even better because it keeps the split net in adjacent partitions.

It is at this point that we use the thumbnails defined in Section 5.2. Consider the

thumbnails in terms of our placement goals. If a net can be placed within one partition, its only thumbnail has a length of zero. If it must be split, then the closer its assigned partitions are, the smaller its thumbnails will be. Thus thumbnails can be used to model the quality of our placement and formalize our partition problem.

Problem 5.2 *Given a portion of chip with a gird partition and a set of logic blocks in the portion, the objective of the* **partition problem** *is to assign, or rearrange, the logic blocks to partitions such that the sum of the thumbnail lengths over all the nets is minimized.*

Consider again Figure 5.8. Calculating the thumbnail sizes for each placement is straightforward. In the first, each net has a thumbnail size of three, for a total score of six. In the lower left placement, the split net still has a thumbnail size of three, while the other net has a size of zero, for a total score of three. In the third placement, the split net has been reduced to its minimum possible size, 1, while the other net remains 0. This placement has a total score of 1 – an optimal partition score for this example.

Finding an optimal placement is NP-complete [45]. Even with the Steiner Tree structure allowing us to find the size of the thumbnail in constant time, we cannot find the optimal solution for this problem in polynomial time (unless $P = NP$). Thus we use a probabilistic search method to find a good solution: simulated annealing.

5.4.3.2 Problem Solution

Simulated annealing, First introduced by Kirkpatrick, Galatt and Vecci [55], is a probabilistic search method modeled on the annealing of crystals. The algorithm works by starting with an arbitrary solution and randomly searching the neighborhood for improved variations. An initial "temperature" is computed, dictating the probability that the algorithm will accept a solution inferior in quality to the solution currently under consideration. (A better solution is always accepted.) As the temperature is decreased over the course of the algorithm, the probability of accepting inferior solutions decreases until the algorithm converges to a stable answer. Thus the algorithm searches for local optima, but also hill-climbs

Let S be a random solution.

Randomly sample b^2 different solutions.

Set t to the standard deviation of the sampled scores.

Let S_B be the set of those solutions.

While the first stopping criterion is not met:

  Repeat until second stopping criterion is met:

    Choose a random neighbor S' of S.

    Let $\Delta = \Phi(S') - \Phi(S)$.

    If $(\Delta < 0)$ or $(\text{Rand}(0,1) < e^{-\Delta/t})$

      $S \leftarrow S'$

    If $(\Phi(S) < \Phi(S_B))$

      $S_B \leftarrow S$

  Decrease t by a fixed percentage

Figure 5.9: The simulated annealing algorithm for the partition phase. The function Rand(0,1) returns a random value between 0 and 1, and $\Phi(S)$ returns the score of solution S – the sum of the thumbnails.

in order to avoid being trapped in any one local optimum. As the temperature decreases the amount of hill-climbing tends to decrease.

The idea behind simulated annealing is based on the concept of the neighborhood of a solution. Given a solution S , the neighborhood of S, denoted , (S) , is the set of solutions that can be reached from S by a single move. For our problem, a move consists of the swapping of two logic blocks in different regions. Swapping the blocks can be done in constant time, and calculating the new score requires time linear in the number of nets mapped to the blocks being switched. As the standard architecture allows no more then eight nets per logic block, this is a constant time operation.

In Figure 5.9 we see the basic simulated annealing algorithm (though the two loop-halting criteria must still be addressed). The number of the iterations of the inner loop is a function of the number of solutions the algorithm has tried or accepted. It will stop when $O((b+n)^2)$ solutions have been tried or $O(n+b)$ solutions have been accepted during the loop. The outer loop watches for solution convergence, halting when no significant improvement occurs for a pre-specified number of iterations.

5.4.3.3 Asymptotic Analysis

Given the probabilistic nature of the simulated annealing technique, it is difficult to give a precise run-time bound on the algorithm. However, we can derive a loose worst-case bound.

The initialization of the algorithm is simple: we require $O(b^3)$ time to compute the initial temperature. Within the inner loop there are three operations: switching two logic blocks, calculating the new score, and recording the new solution if it is the best seen so far. As any technology has a constant number of pins per logic blocks, we can take that number as a constant. Therefore these first two operations require constant time, and the third requires $O(b)$ time. Thus one iteration of the inner loop requires $O(b)$ time. The inner is executed until there have been $O(n+b)$ solutions accepted or $O((n+b)^2)$ solutions examined, leading to a worst case of $O((n+b)^2)$ iterations. Hence one iteration of the outer loop requires $O(b^3+bn^2)$ time in the worst case.

The outer loop is more difficult to optimize, as it does not halt at any pre-specified time. Instead, it tends to halt when the temperature decreases to a sufficiently small level. The initial temperature is based on the standard deviation of a random sampling of solution-space members. Given a partition graph of size v, no net can have a thumbnail of size greater than $v-1$. Hence the largest possible score of a solution is $(v-1)n$, while the smallest is 0. Thus the maximum standard deviation, which the maximum possible starting temperature, is $\sigma = \frac{1}{2}n(v-1) = O(nv)$.

In every iteration of the outside loop the temperature is reduced by a specified percent,

until the probability of accepting poorer quality solutions is sufficiently small, which will eventually leave the algorithm in a local optima. The probability of accepting a new solution of lesser quality is $e^{\frac{-d}{t}}$, where d is the difference in solution quality – hence $d \geq 1$. Thus for any given t, the probability of regressing to any poorer solution is at most $e^{\frac{-1}{t}}$. Assume we begin at temperature t_i , we choose to halt when this probability is no greater than some fixed value ω_0 , and assume the temperature is decreasing to some fixed percentage c every iteration ($0 < c < 1$). Thus on the kth iteration the temperature will be $t_i c^k$, and the probability of regression will be $e^{-\frac{1}{t_i c^k}}$. In the worst case, $t_i = O(nv)$, and we will require $O(\log nv)$ iteration to reach the cutoff probability ω_0.

Complexity Analysis 5.2 *The worst case run-time of the Spiffy partition stage is*

$$O(b^3) + O((b^3 + bn^2 + b^3) \log nv).$$

Note that while this is the tightest bound we can compute, it is still a very-loose worst case bound, with an average case bound that is likely to be much smaller.

5.4.4 Route Selection

Once the blocks have been placed in partitions, we must consider the nets that have been split between partitions and choose inter-partition routes for them. Here the exact configuration of the thumbnails becomes important.

5.4.4.1 Problem Definition

Given a placement of the blocks, we must pick an inter-partition route for each split net such that:

- Each source-to-sink path of the net is minimized.
- No one region is over-congested because a disproportionately large number of nets cross over one boundary.

The first objective is easily modeled with thumbnails as a thumbnail of the net contains the shortest source-to-sink paths possible for the net. Hence we need to pick one thumbnail for each net, which could be done in $O(n)$ time except for the second condition.

Assume we assign each net ν a thumbnail, denoted $\tau(\nu)$, and from this assignment we create the *congestion vector* β such that

$$\beta[e] = |\{\nu \,:\, e \in \tau(\nu)\}|.$$

In other words, $\beta[e]$ counts the number of times edge e is used. To satisfy the second condition, we try to keep the values of $\beta[e]$ roughly equal. To do this, we want to minimize their variance

$$\sigma_\beta^2 = \frac{\sum_{e \in E}(b[e] - \overline{b})^2}{|E| - 1},$$

where E is the set of edges in the partition graph and $\overline{\beta}$ is the average of the elements of β. From this, we can formally define our problem:

Problem 5.3 *Given a placement on a partitioned chip, for each net ν choose a thumbnail $\tau(\nu)$ such that the variance of the corresponding congestion vector is minimized.*

5.4.4.2 Problem Solution

If the number of partitions v is considered to be parameter of the problem, the problem would is than NP-complete [57, 75]. However, as we are computationally confined to using small partition graphs, we can consider v and $|E|$ to be constant. Given this, the problem can be solved in polynomial time by a dynamic programming algorithm, but this algorithm requires $O(2^{|E|}n^{|E|+1})$ [43]. As we are dealing with values of $|E|$ of no less than 12, this is an $O(n^{13})$ algorithm, which unacceptable in practice.

As the exact algorithm is too slow, we use a first-fit greedy heuristic to compute a good solution that works in two stages:

1. Sort the nets according to the number of different thumbnails each has available to it.

2. Consider the nets in sorted order, and for each net pick the template that minimizes the congestion value for the partial-solution developed to that point.

Therefore we first route those nets which have few choices and save other nets to compensate for the less avoidable congestion caused earlier in the algorithm.

5.4.4.3 Asymptotic Analysis

We begin by sorting the nets by the number of thumbnails that are available to each. This number that is bounded by the constant $\Lambda(G)$, so we can use a simple bucket sort with an $O(n)$ bound. Following this, we once again perform at most $\Lambda(G)$ constant operations on each net.

Complexity Analysis 5.3 *The worst-case and average-case run time of the route selection stage of the algorithm is* $O(n)$.

5.4.5 Virtual Terminal Assignment

At this point we have partitioned each net and assigned each an inter-partition route. For each net, we must assign a virtual terminal along any partition-lines the net's routes cross without overloading any switch blocks. Once this is done, each partition can be considered independently, and the union of the solutions result in a full global route.

5.4.5.1 Problem Definition

For any edge e in the partition graph, let $N(e)$ be the set of all nets that use a thumbnail contain edge e, and let $S(e)$ be the set of all switch blocks on the edges of the partitions corresponding to the two nodes incident to e. In assigning virtual nodes we are mapping each element of $N(e)$ to an element of $S(e)$ in order to:

1. Minimize the maximum number of nets assigned to any one switch block.

2. Minimize the sum of the distances of each net from the nearest switch blocks to which they are assigned.

The first of these goals, which we prioritize, can be formulated by the requirement to minimize $\max_{s \in S(e)} |\{\nu \in N(e) : \nu \text{ is mapped to } s\}|$. The second objective requires a distance function $\delta : (N(e) \times S(e)) \rightarrow \mathbb{R}$ such that $\delta(\nu, s)$ is the sum of the distances from s to the nearest terminal of ν on each side of the partition line.

Having defined these, we can now formulate the problem:

Problem 5.4 *Given a placement of each net and a thumbnail assignment for each net, find a mapping $M_e : N(e) \rightarrow S(e)$ for each edge e in the partition graph such that the value:*

$$n \cdot c \cdot \max_{e \in E}\{\max_{s \in S(e)}\{|\{\nu \in N(e) : M_e(\nu) = s\}|\}\} + \sum_{e \in E}\sum_{\nu \in N(e)} \delta(\nu, M_e(\nu))$$

where c is some number larger than twice the size of the largest path on the chip.

The first part of the function reflects the largest number of nets assigned to any one switch block. The second term reflects the distance of the switch block to each net terminal. The first sum is multiplied by $n \cdot k$ so that it will dominate the second term, thus prioritizing width over distance.

5.4.5.2 Problem Solution

It is simple to determine the optimal value of the first term of the function. Given an edge e, the pigeon hole principle tells us that we must assign at least $\lceil \frac{N(e)}{S(e)} \rceil$ nets to some switch block in $S(e)$. Nor do we ever have to assign more than that number of nets to any switch block. So the optimal value of $\max_{e \in E}\{\max_{s \in S(e)}\{|\{n \in N(e) : M_e(n) = s\}|\}\}$ is equal to $\max_{e \in E}\lceil \frac{N(e)}{S(e)} \rceil$. We can easily compute this number, which we denoted ρ .

We solve the remainder of the problem by translating it into a *minimum-cost perfect matching* problem, first solved by Hopcraft and Karp [48]. Though this problem could be solved for the entire chip at once, by using the ρ value and solving it edge-by-edge, we

can compute a more accurate δ function for the later edges that is updated to reflect the mappings made for the earlier edges.

For each edge e, we create a complete bipartite graph as follows. For each element in $N(e)$ we create a node. These nodes together form the first partition, which has size $|N(e)|$. For each element in $S(e)$ we create ρ nodes, forming the second partition of nodes, which has size $|S(e)|\rho$. Then for every net $n \in N(e)$ and every switch block $s \in S(e)$, we add an edge between n's node and each of s's node with weight $\delta(n, s)$. Following this construction, we use the Hopcraft/Karp algorithm, as encoded by Saltzman [82], to pick exactly one edge adjacent to each net-node such that the edge-weights are minimized. Thus we have induced a mapping optimizing the objective function.

5.4.5.3 Problem Analysis

The Hopcraft algorithm, which runs in $O(|N(e)|^{\frac{5}{2}})$ time, must be run for each edge e. For each edge we must create the δ function, which requires $O(\frac{b}{v}S(b))$ time to check each block in each partition against each switch block. In the worst case, $|N(e)| \leq n$ and $S(b) \leq b$.

Complexity Analysis 5.4 *The virtual-node assignment phase has a worst-case runtime of* $O(b^2 + n^{\frac{5}{2}})$.

5.4.6 Source Computation

The last task before the recursion is to update the source tree for each net with respect to the new partitions. Recall from Section 5.3 that each net contains a source tree, defining the paths from its source to it sinks. As we have now added new pins to each net, we need to insert them into the source trees. Further, we have created v new partitions, each of which needs to be provided with a list of each net's source and exits with respect to the portion in question.

The solution to this is straightforward. However, it requires the support of the provided data structures to perform efficiently. As the chip portion to which we have been applying

the algorithm has each node's source and sink pins recorded, we can perform this operation in constant time for each net. For a given net, we select the source node for that net and begin a search of the net's global routing tree. The tree indicates which portion lines are now crossed, and which switch blocks have been selected. We can create and add the nodes into the source tree in constant time, while also updating each partition. The search of the tree requires $O(v)$ time, hence the final stage of our analysis:

Complexity Analysis 5.5 *Updating the source-trees and chip portion objects requires $O(n)$ time.*

Having finished this step, we now have enough information for each new portion to treat it independently, and know that the union of the solutions will provide a global-route.

5.4.7 Base Case

The base case consists of a chip portion containing exactly one logic block. In such a case, the portion will contain that logic block, the four neighboring connection blocks, and the four neighboring switch blocks. This configuration is shown in Figure 5.10, with the graph representation of the layout which is used to formulate our problem.

5.4.7.1 Problem Definition

When we reach a portion containing exactly one switch block, that block may contain nets mapped to its logic pins. Further, the four switch blocks surrounding the logic block may serve as virtual terminals for some nets. For each net, the set of blocks assigned to that net must be connected. In Figure 5.11 we see the selection of nodes for some net, and three possible ways to connect the graph. Each of these is a potential *layout-graph*, and denote In terms of minimizing net length, the first route is clearly the best. However, we need to select a routing-graph for each net without over-using any channel. As minimizing channel width takes priority over minimizing the route length, the presence of other nets could force the use of one of the longer routes.

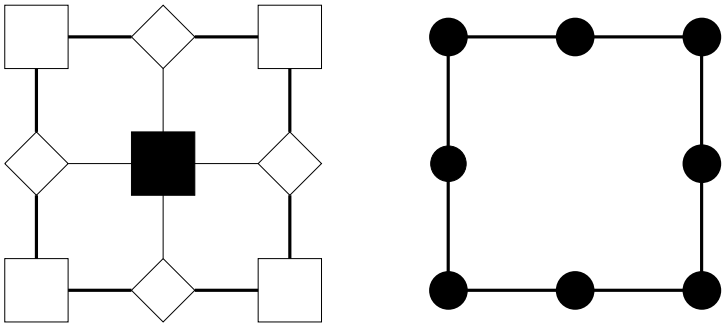


Figure 5.10: The layout of a base-case chip portion and a graph representation of that layout. In the left diagram we see the blocks surrounding the logic block in question. In the right diagram we see a graph representation of the block structure. Each node represents one block, and two nodes are adjacent if the corresponding blocks are connected by a wire.

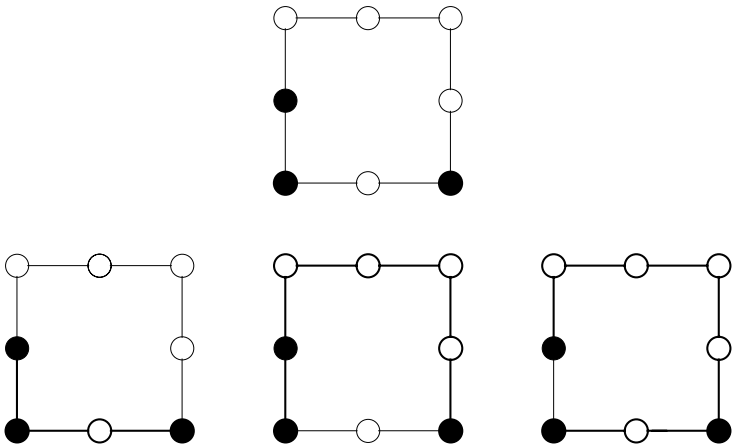


Figure 5.11: A sample base-case layout graph, and three possible routings for the graph.

Problem 5.5 *Given the placement of nets in a single chip portion, assign each net ν a layout-graph $\eta(\nu)$ such that:*

1. *The value* $\max_e |\{\nu : \eta(\nu) \text{ uses edge } e\}|$ *is minimized.*

2. *The total number of edges is minimized.*

With (1) taking priority over (2).

5.4.7.2 Problem Solution

The problem is NP-complete [41, 43]. However, as this problem is the base case of the recursion, the problem instance is small. We know that $b = 1$, and we can reasonably assume the number of nets will be a small percentage of the original number. Thus we have chosen to use an exact algorithm. We use the integer programming technique formulated by Ganley [43] to get the best possible solution, using the tool LP_SOLVE tool produced by Berkelaar [13], and have found that in practice the algorithm works quickly.

5.4.7.3 Problem Analysis

A meaningful analysis of the problem solution for the base-case is quite difficult. Because we are using integer programming, we know the worst case run-time is exponential in n, but we do not have a bound on n. While there is no theoretical reason the base case portion could not contain all nets on the chip, in practice the number of nets it does contain is *always* a very small percentage of the original number.

5.5 Asymptotic Analysis of the Spiffy Algorithm

We have now presented the six stages to the Spiffy algorithm, each with the tightest analysis known:

1. Initialization: $O(b) + O(n \log b) + O(n)$.
2. Partitioning: $O(b^3) + O((b^3 + bn^2) \log n)$.
3. Route Selection: $O(n)$.
4. Virtual Terminal Assignment: $O(b^2 + n^{\frac{5}{2}})$.

5. Source-Tree Update: $O(n)$.

6. Base-Case: $f(n)$ (Unknown) .

Thus we can calculate the time complexity of the recursion phase of Spiffy:

$$\begin{aligned} T(n,b) &= vT(\frac{b}{v},n) + O((bn^2 + nb^2 + b^3)\log nv + \Lambda(G) + \frac{1}{v}b^2 + O(n^{\frac{5}{2}}) \\ &= bf(n) + O(bn^2 \log b \log vn + \frac{bv^2(b^2-1)\log nv}{(v^2-1)} + \\ &\quad \frac{\Lambda(G)(b-1) + b(bv-v) + n^{\frac{5}{2}}}{v-1}) \end{aligned}$$

Complexity Analysis 5.6 *Taking the size of the partition graph as a constant and the net-size and chip-sizes as input parameters, the Spiffy Algorithm has a worst-case runtime bound of:*

$$O(bf(n) + (bn^2 \log b + nb^2 + b^3)\log n + b^2 + n^{\frac{5}{2}}b)$$

where $f(n)$ is the runtime bound of the base-case.

In addition to the integer programming technique used for the base-case, Ganley also presents a $O(n^3)$ heuristic. If we were to use this, we would then be able to bound Spiffy at $O(bn^3 + b^3 \log n + bn^2 \log b \log n)$. However, linear programming has proved to work quickly, and also produces solutions of considerably higher quality.

We reiterate that this a very loose worst-case bound. While the best bound we could prove for the simulated annealing was $O(b^3 + bn^2 \log n)$, an average-case or amortized analysis would likely be much better – leading to a much better bound over-all. Further, we have assumed that the full number of nets will be considered in each recursion, while in truth the number under consideration will be considerably reduced in each recursion. Thus the average case bound should be significantly lower than the worst case bound presented here.

5.6 Experimental Results

Now that we have explained the Spiffy algorithm, we show that the implementation of Spiffy produces quality results. In this section we discuss the results of using Spiffy on standard benchmarks, and compare them against the results of other tools. As there are no public results for 3D-FPGAs, we limit ourselves to benchmarks for the 2D architecture to demonstrate the quality of our tool.

5.6.1 Benchmarks

For our comparisons we have used a set of benchmarks created at the University of Toronto that have been the basis for the testing of a number of tools [3, 6, 7, 8, 9, 14, 15, 21, 23, 24, 40, 43, 53, 62, 66, 78]. They are designed for the two 2D-FPGA symmetric architectures, the Xilinx 3000 and 4000. The characteristics of each benchmark can be seen in Table 5.1.

Circuit Name	FPGA Size	Number of Nets
busc	13×12	151
dma	18×16	213
bnre	22×21	352
dfsm	23×22	420
z03	27×26	608
9symml	11×10	79
term1	10×9	88
apex7	12×10	115
alu2	15×13	153
too_large	14×14	186
example2	14×12	205
vda	17×16	225
alu4	19×17	255
k2	22×20	404

Table 5.1: Statistics on the benchmark circuits used to test Spiffy. The first five benchmarks were originally designed for the Xilinx 3000-series circuits and the last nine for the 4000-series circuits.

5.6.2 Detailed Routing

When testing placement tool or global routing tool on an FPGA, the quality of the results can only be judged after the detailed routing has been performed. Hence we need a detailed router to process the output of Spiffy. In the course of our testing, we made use of two such tools: one created by Alexander and Robins [8, 9], and the Upstart tool created by McCulloch and Cohoon at the University of Virginia. Each router is graph based, working by sequentially routing the nets along the global routes, and recomputing those routes when necessary.

5.6.3 Spiffy vs Mondrian

In rating the quality of Spiffy, compare it to its predecessor, Mondrain. This is a natural comparison, as Mondrian is based on the same principles as Spiffy, was a state-of-the-art tool at the time of its implementation, and still produces some of the best results in the literature. In the next two sections we discuss the differences between Spiffy and Mondrian, and present the experimental results from running each tool on a set of standard benchmarks.

5.6.3.1 Differences

Though Spiffy is based on the same general principles as the Mondrian algorithm of Cohoon and Ganley [43], there are significant technical differences that make Spiffy a more efficient, effective and flexible tool. We will subsequently compare these results to support this claim, but we first discuss the difference, explaining why Spiffy is the better tool.

Both tools work on the principles of geometric partitioning: place a partition on the chip, place and route blocks between partitions, assign virtual nodes and recursively apply the algorithm to each partition. However, the implementation details are significantly different, and the capabilities of Spiffy are superior to those of Mondrian. Among the most important differences are:

- **Object oriented design**: As discussed in Section 5.3.2, Spiffy was written in an objected oriented manner. Built on the standard template library [10] and loosely based on the FPGA data-structures created by McCulloch and Cohoon, this design provides a number of advantages:

    1. **Efficiency**: The object-oriented nature of the design leads to greater efficiency. By streamlining and improving a number of the data-structures used by Mondrian, as well as experimenting with various alternatives and additions to the original data structures, we were able to significantly decrease the runtime of the algorithm.
    2. **Flexibility**: Because of the design, there is considerable flexibility in the operation of the algorithm, allowing easy modification and experimentation. Having finished the basic Spiffy algorithm, we were able to experiment with a significant number of variations on the program that were not easily introduced into Mondrian. Among the most important of these were the introduction of template smoothing and the creation of the Gambit tool, as discussed in Chapters 6 and 7. With the basic design of Spiffy, the implementation of these new ideas was quick and straightforward.
    3. **Generlizability**: Spiffy can be readily modified to operate on other architecture variations. As the FPGA is represented by an object, the object need only be modified to reflect the characteristics of the new architecture. So long as the interface methods are present in the object description, Spiffy will continue to function and produce chip mappings. Architecture variations could be as simple as a modified FPGA architecture, adding or modifying certain characteristics, or as complicated as switching to a different channel-based semi-custom design architecture.
    4. **Readability**: Spiffy is more readable then Mondrian, allowing researchers interested in the area great ease in experimenting with variations on the basic

algorithm. Because of the modularization, a researcher can isolate and experiment with one aspect of the algorithm (e.g. replacing the simulated annealing with an alternative solution method), or experiment with more general changes in the methodology.

- **Performance Driven Capabilities**: While Mondrian was based on Steiner trees, then later adopted for arboresences, Spiffy is designed specifically for arboresences. As the maximum source-to-sink path length of a net is a more precise estimation of performance, the arboresence measurement is a superior model. Because Spiffy was designed for them, Spiffy does a better job at minimizing these path-lengths. Where Mondrian treated the arboresences as Steiner trees, Spiffy keeps a detailed list of source information throughout the refinements, and is designed to minimize source-to-sink lengths in ways Mondrian could not. Further, Spiffy is able to maintain an exact measurement of the source-to-sink path-lengths of each net, where Mondrian could only estimate these lengths.

- **Steiner Tree Encodings**: As discussed in Section 5.3.3, we exploited the regular properties of the partition graph to encode the arboresences into an efficient data structure. As a result, we have reduced the cost of the Steiner-tree operations by an order of magnitude and greatly improved the runtime of the Spiffy algorithm.

- **Block Portion Objects**: Where Mondrian decomposes the chip in the course of the normal refinements, Spiffy pre-computes these chip portions and creates an object for each. With these objects available during the course of the execution, Spiffy has access to more information and has opportunities to prepare these objects for the algorithm's execution. As a result, both runtime and the quality of the final results are improved.

- **Improvements in the Simulated Annealing Algorithm**: As the simulated annealing algorithm requires that the solution be randomly modified a considerable

number of times, it is vital that we be able to make the modification quickly. By introducing several new data structures and the tree encodings addressed above, these modifications can be made much faster than in the Mondrian algorithm, causing a considerable improvement in the execution time.

- **Improvements in Route Selection and Virtual Terminal Assignment**: A number of minor improvements have been made to the route selection and virtual terminal assignment algorithms. While no one of the changes is significant to merit discussion, the cumulative effect of these changes is considerable.

- **Improvements in Base-Case Solution**: A number of constraints were added to the integer programming, restricting the solution space and thus improving the runtime of the algorithm. Further, Spiffy treats the base-case independently (and after) the recursion, in a carefully selected order to further improve the results.

5.6.3.2 Comparison of Results

Our tests begin with a comparison to Mondrian, the predecessor to Spiffy. In Table 5.2 we see that with a few exceptions, Spiffy runs significantly faster than Mondrian. As both tools involve probabilistic search, these numbers are in a sense a random sampling of the true speedup of Spiffy, hence the confidence intervals indicated for each benchmark. Taking the midpoint of each interval as the point-estimation for the runtime over each benchmark, we see an average runtime improvement of 37.7%.

We now consider the quality of the results. As noted above, we cannot directly judge the quality of either tool's output, we are obliged to pair each tool with a detailed router. In the research done by Ganley [43], the output of Mondrian was routed by a the tool created by Alexander and Robins [8, 9]. We originally paired Spiffy with this router, but found that this combination resulted in mappings superior in net-length but inferior in congestion [54, 53]. The reasons for this loss of quality are due of the lack of a post-processing heuristic

Circuit	Mondrian	Spiffy	Improvement
busc	77.5	106.3	-37.2%
dma	196.8	137.9	29.9%
bnre	654.4	230.0	64.4%
dfsm	528.6	289.1	45.3%
z03	1042.3	523.0	49.8%
9symml	56.1	27.7	50.6%
term1	30.9	30.2	2.3%
apex7	48.0	56.5	-17.7%
alu2	154.4	74.5	51.8%
too_large	216.5	102.3	52.8%
example2	139.7	161.5	-15.6%
vda	1019.3	280.2	72.5%
alu4	1105.8	153.2	86.2%
k2	12198.8	872.3	92.8%
Average			37.7%

Table 5.2: A comparison of the run-times of Mondrian and Spiffy. All tests were run on a 50MHZ Sparc 20 using SunOS 4.1.3. Time is given in seconds.

employed by Mondrian, and because the Alexander and Robins tool tends to ignore much of the global routing information produced by Spiffy.

When we paired Spiffy with Upstart, the results improved significantly. In Table 5.3, we see the channel widths of those routings produced by the Mondrian tool and Alexander and Robins router pairing, as compared to those produced by the Spiffy and Upstart pairing. Each result in the table is an average for of thirty runs, with the corresponding 95% confidence interval given. Over the nine benchmarks tested, Spiffy was able to improve the channel with by an average of 13.2%.

Using the same methodology, Table 5.4 gives us the comparison of Mondrian and Spiffy (with their respective routers) when measuring net-length. When running the tools with an arboresence-based model, thus optimizing source-to-sink paths, Spiffy improves the total wire-length by 10.2%. While the more interesting statistic for this case would be the average source-to-sink path length, this information was not provided for Mondrian, making

Name	Mondrian	Spiffy	%Δ
9symml	11.4	8.86 ± 0.18	22.3% ± 1.6%
term1	7.0	7.50 ± 0.22	−7.1% ± 3.1%
apex7	9.1	7.53 ± 0.22	17.3% ± 2.4%
alu2	12.2	9.85 ± 0.19	19.3% ± 1.6%
too_large	11.9	10.05 ± 0.20	15.6% ± 1.7%
example2	9.3	9.47 ± 0.33	−1.8% ± 5.6%
vda	15.0	12.03 ± 0.23	20.7% ± 3.6%
alu4	14.5	11.67 ± 0.19	19.5% ± 1.3%
Average			13.2%

Table 5.3: A comparison of channel widths of Mondrian results (routed by the Alexander/Robins tool) against Spiffy results (routed by the Upstart tool) for the Xilinx 4000 benchmarks. All Mondrian results are taken from Ganley's thesis [43]. All Spiffy results are an average of 60 runs, stated in terms of 95% confidence intervals.

comparison impossible.

Name	Mondrian	Spiffy	% Gain
9symml	21.8	20.53 ± 0.38	5.8% ± 1.7%
term1	14.0	13.60 ± 0.46	2.9% ± 3.3%
apex7	18.2	15.06 ± 0.31	17.3% ± 1.7%
alu2	24.4	22.72 ± 0.37	6.9% ± 3.7%
too_large	23.8	20.87 ± 0.27	12.3% ± 1.1%
example2	18.6	14.95 ± 0.58	19.6% ± 2.3%
vda	30.0	27.20 ± 0.38	9.3% ± 0.8%
alu4	29.0	26.84 ± 0.16	7.5% ± 0.5%
average			10.2%

Table 5.4: A comparison of net wire lengths of Mondrian results (routed by the Alexander/Robins tool) against Spiffy results (routed by the Upstart tool) for the Xilinx 4000 benchmarks. All Mondrian results are taken from Ganley's thesis [43]. All Spiffy results are an average of 60 runs, stated in terms of 95% confidence intervals.

5.6.4 Comparisons Against Other Tools

Now that we have established Spiffy's superiority over Mondrian, the start-of-the-art tool in the literature at the time of its publication in 1995 [7], we need to compare Spiffy against tools produced since that time.

In Table 5.5 we compare the FPGA placement tool Altor [80] against Spiffy in terms of channel width. For the global routing, Altor is paired with a verity of tools, including LocusRoute [77], GBP [104], TRACER [59], VPR [15], and the Alexander/Robins routing tool [9]. For those global routers that did not compute a detailed route as well, the tools CGE [23], SEGA [61] and VPR are used to complete the mapping. In all cases, Spiffy performs better – with a 15.6% over the next best tool suite in the terms of the total number of tracks used.

Placement Tool	Altor							Spiffy
G. Routing Tool	LocusRoute		GBP	OCG	Tracer	VPR	A/R	
D. Routing Tool	CGA	SEGA				SEGA		Upstart
9symml	9	9	9	9	6	7	8	8
alu2	12	10	11	9	9	8	9	9
alu4	15	13	14	12	11	10	11	10
apex7	13	13	11	10	8	10	10	6
example2	18	17	13	12	10	10	11	8
term1	10	9	10	9	7	8	8	6
too_large	13	11	12	11	9	10	10	9
vda	14	14	13	11	11	12	12	10
Total	104	96	93	83	71	75	76	66

Table 5.5: Comparison of channel widths resulting from the Altor placement tool [80] paired with various route tools, compared against the Spiffy/Upstart suite. All scores shown are the best out of 60 trials. Tools for scores other than Spiffy are taken from the paper on VPR [15].

In Table 5.6 we compare the results of the Spiffy/Upstart suite against the VPR placer. While the results do not look promising – for each benchmark Spiffy is at best able to tie VPR's results – the comparison is questionable. In the experiments run by Betz and

Rose [15] the placement was calculated using a different technology mapping then used in our experiments. As a result, the nets in their placement are smaller, naturally leading to smaller channel widths.

Placement Tool	Altor	VPR	VPR	Spiffy
G. Routing Tool	VPR			
D. Routing Tool		SEGA		Upstart
9symml	6	6	5	8
alu2	8	6	6	9
alu4	9	8	7	10
apex7	8	5	4	6
example2	10	5	5	8
term1	6	5	5	6
too_large	10	7	6	9
vda	10	8	8	10
Total	65	52	46	66

Table 5.6: Comparison of channel width using VPR for various stages, compared against the Spiffy/Upstart suite. All scores shown are the best out of all trials, with scores for VPR benchmarks taken from the paper on that tool [15]. Note that as VPR uses a different technology mapping for each circuit, it was provided with an advantage in computing a good mapping.

In terms of net-length, we have shown that Spiffy improves circuit mappings over Mondrian. However, it is not standard practice for papers to report estimations of this metric. Hence we have no results to compare against.

5.6.5 Changing the Partition Size

To this point, all experiments have been run using a $1 \times 3 \times 3$ partition. It is reasonable to assume that by changing this partition size, the quality of our results should vary. However, it is computationally infeasible to work with a partition such that $v > 12$; the structure containing the Steiner trees is too large to hold in memory. Thus we experimented with three different partitions: $1 \times 2 \times 2$, $1 \times 3 \times 2$ and $1 \times 4 \times 3$. In Tables 5.7, 5.8 and 5.9 we see the results of using these partitions. It is no surprise that the quality of the mappings

suffer from the use of the smaller partitions, or that the runtime suffers with the use of the larger partitions. However, it is odd that the channel-width suffers from the use of the larger partition, but this is likely a factor of statistical sampling error. It likely we would see more dramatic improvements in the use of larger partitions, but such experiments are not feasible.

Circuit	$1 \times 2 \times 2$		$1 \times 3 \times 2$		$1 \times 4 \times 3$	
	Width	%Δ	Width	%Δ	Width	%Δ
busc	8.37 ± 0.33	-4.58%	8.50 ± 0.31	-6.25%	7.80 ± 0.25	2.50%
dma	10.33 ± 0.41	-17.87%	9.50 ± 0.34	-8.36%	8.43 ± 0.23	3.80%
bnre	13.80 ± 0.42	-28.17%	13.10 ± 0.48	-21.67%	10.60 ± 0.29	1.55%
dfsm	13.50 ± 0.45	-31.92%	12.53 ± 0.45	-22.48%	9.63 ± 0.33	5.86%
9symml	9.17 ± 0.26	-3.38%	9.10 ± 0.23	-2.63%	8.63 ± 0.25	2.63%
term1	8.03 ± 0.37	-7.11%	8.10 ± 0.45	-8.00%	7.53 ± 0.38	-0.44%
apex7	7.70 ± 0.33	-2.21%	7.93 ± 0.40	-5.31%	7.13 ± 0.27	5.31%
alu2	10.27 ± 0.38	-4.23%	10.17 ± 0.30	-3.22%	9.57 ± 0.29	2.88%
too_large	10.43 ± 0.31	-3.81%	10.57 ± 0.27	-5.14%	10.17 ± 0.42	-1.16%
example2	9.93 ± 0.29	-4.93%	9.80 ± 0.38	-3.52%	9.10 ± 0.40	3.87%
vda	13.27 ± 0.48	-10.25%	13.20 ± 0.35	-9.70%	11.57 ± 0.38	3.88%
alu4	13.17 ± 0.43	-12.86%	13.00 ± 0.40	-11.43%	11.80 ± 0.49	-1.14%
Average		-10.94		-8.98		2.46

Table 5.7: The channel-width resulting from the use of different partitions, and the percentage improvement over the $1 \times 3 \times 3$ partition. Scores are averaged over 30 runs, with all intervals being at the 95% level of confidence.

5.7 Summary and Conclusion

Based on the Mondrian tool, Spiffy employs a divide-and-conquer algorithm designed to simultaneously generate a placement and a global routing for FPGAs. In this chapter, we presented the details of the tool and the experimental results produced by running the tool on a standard set of benchmarks. Confining ourselves to 2D-FPGAs, we found that Spiffy runs considerably faster than its counterpart, and when used in conjunction with the Upstart router produced better results. In comparing it to other tool suite we find that

Circuit	$1 \times 2 \times 2$		$1 \times 3 \times 2$		$1 \times 4 \times 3$	
	Length	%Δ	Length	%Δ	Length	%Δ
busc	11.62 ± 0.29	-6.43%	11.26 ± 0.23	-3.12%	10.36 ± 0.17	5.08%
dma	17.40 ± 0.38	-18.66%	16.19 ± 0.24	-10.39%	14.45 ± 0.20	1.49%
bnre	21.37 ± 0.37	-31.92%	18.95 ± 0.26	-16.97%	15.71 ± 0.16	3.01%
dfsm	19.49 ± 0.26	-35.54%	17.13 ± 0.31	-19.08%	13.50 ± 0.16	6.13%
9symml	13.56 ± 0.49	-6.53%	13.22 ± 0.46	-3.86%	12.80 ± 0.39	-0.57%
term1	10.80 ± 0.42	-5.33%	11.30 ± 0.60	-10.15%	10.62 ± 0.55	-3.52%
apex7	11.25 ± 0.40	-5.75%	11.40 ± 0.51	-7.19%	10.36 ± 0.31	2.64%
alu2	15.56 ± 0.23	-7.43%	15.27 ± 0.20	-5.46%	13.98 ± 0.15	3.48%
too_large	16.12 ± 0.35	-10.92%	15.77 ± 0.32	-8.49%	14.44 ± 0.33	0.66%
example2	12.03 ± 0.21	-7.03%	11.97 ± 0.58	-6.55%	10.61 ± 0.41	5.56%
vda	22.20 ± 0.78	-17.64%	20.69 ± 0.29	-9.65%	18.28 ± 0.20	3.14%
alu4	20.53 ± 0.29	-17.52%	19.39 ± 0.26	-11.00%	17.09 ± 0.21	2.14%
Average		-14.22		-9.33		2.44

Table 5.8: The average path length resulting from the use of different partitions, and the percentage improvement over the $1 \times 3 \times 3$ partition. Scores are averaged over 30 runs, with all intervals being at the 95% level of confidence.

Spiffy suite improves channel width over all tools except the VPR suite. However, VPR employs a tool that can map the circuit instance to equivalent but smaller instances.

We also note that Spiffy is equipped to provide a great deal of flexibility to the user. Not only may the user specify that Spiffy replace its aboresence model with a Steiner tree model, there is also the option to control a number of parameters. The temperature decrease and termination criteria for the simulated annealing may be specified by the user, as may the cost of transversing vertical interconnections. Further, the use may specify that the simulated anneal heuristic be replaced with an exact branch-and-bound algorithm for all chip portions of a given sizes.

In the next chapter we discuss *template smoothing* – a new technique added to the Spiffy methodology to further improve results.

Circuit	$1 \times 2 \times 2$		$1 \times 3 \times 2$		$1 \times 4 \times 3$	
	Time	%Δ	Time	%Δ	Time	%Δ
busc	7.30 ± 0.78	15.28%	8.30 ± 0.80	3.68%	8.47 ± 0.99	1.74%
dma	12.37 ± 0.50	-5.40%	12.60 ± 0.65	-7.39%	13.80 ± 0.70	-17.61%
bnre	28.07 ± 0.87	-30.85%	24.13 ± 0.54	-12.51%	22.53 ± 0.62	-5.05%
dfsm	35.60 ± 0.80	-25.72%	30.53 ± 0.71	-7.83%	30.50 ± 0.77	-7.71%
9symml	2.07 ± 0.09	0.00%	2.07 ± 0.09	0.00%	2.37 ± 0.21	-14.52%
term1	2.07 ± 0.09	7.46%	2.03 ± 0.18	8.96%	2.50 ± 0.29	-11.94%
apex7	3.97 ± 0.43	0.42%	3.30 ± 0.44	17.15%	4.73 ± 0.55	-18.83%
alu2	6.70 ± 0.36	-15.85%	6.50 ± 0.25	-12.39%	6.20 ± 0.27	-7.20%
too_large	7.87 ± 0.27	2.88%	7.77 ± 0.34	4.12%	9.17 ± 0.56	-13.17%
example2	9.43 ± 1.01	10.44%	8.93 ± 1.17	15.19%	12.00 ± 1.39	-13.92%
vda	15.57 ± 0.59	19.90%	16.90 ± 1.36	13.04%	19.30 ± 1.72	0.69%
alu4	14.63 ± 0.25	-14.32%	14.63 ± 0.42	-14.32%	13.53 ± 0.38	-5.73%
Average		-2.98		0.64		-9.44

Table 5.9: The runtime resulting from the use of different partitions, and the percentage improvement over the $1 \times 3 \times 3$ partition. Scores are averaged over 30 runs, with all intervals being at the 95% level of confidence.

6

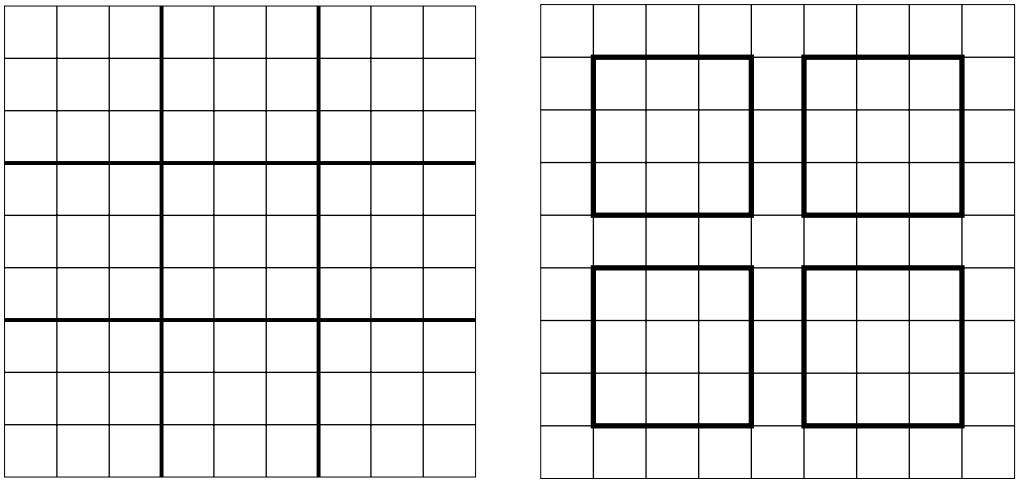
Template Smoothing for Spiffy

In this chapter we present the technique of *template smoothing*, a novel augmentation of the Spiffy algorithm that can be used to significantly improve circuit-mapping quality. In Section 6.1 we provide an overview of the template smoothing technique and motivation for the technique. In Section 6.2 we explain how the technique is integrated into the Spiffy algorithm. In Section 6.3 we present our experimental results.

6.1 Overview

Consider again the steps in the Spiffy algorithm: we divide the chip area into partitions, simultaneously place and route the nets with respect to these partitions, and recursively apply the algorithm to each partition. However, there is no reason we need to apply the algorithm only to those partitions defined by the recursion; it could easily be applied to different regions of the FPGA – regions that do not necessarily "fit" within those regions normally created.

In Figure 6.1 illustrates this idea. In Figure 6.1(a) we see the area of a chip partitioned to two levels of recursion. First the chip is partitioned with a 3×3 grid, then each partition is partitioned with a 3×3 grid. In Figure 6.1(b) we have recombined sets of the second-level partitions to create larger areas, which we refer to as *smoothing-portions*.



(a) (b)

Figure 6.1: (a) An FPGA area partitioned to two levels of recursion with a 3×3 partition grid. (b) Four 3×3 partitions of regions of the chip overlapping the first level partitioned regions, but corresponding to regions defined by the second level of recursion.

Figure 6.2 depicts an example of template smoothing. In Figure 6.2(a) we see a segment of a chip and two levels of a 3×3 partition placed on the chip area. Figure 6.2(b) shows a net assigned to this segment after the completion of Spiffy, and in Figure 6.2(c) the outline of a smoothing-portion is highlighted. In Figure 6.2(d) all information within this smoothing-portion is erased. The algorithm is applied to the smoothing-portion, depicted in Figure 6.2(e). As a result, the final routing tree has been rearranged, leaving unneeded branches. These extra branches must be pruned, giving us our final solution in Figure 6.2(f). Taking the gray block as the source for the net, we see that this final solution has decreased the maximum source-to-sink path.

Definition 6.1 *Consider an application of the Spiffy algorithm at level l of the recursion, and let A denote the portion of the chip to which Spiffy is being applied at that point. We*

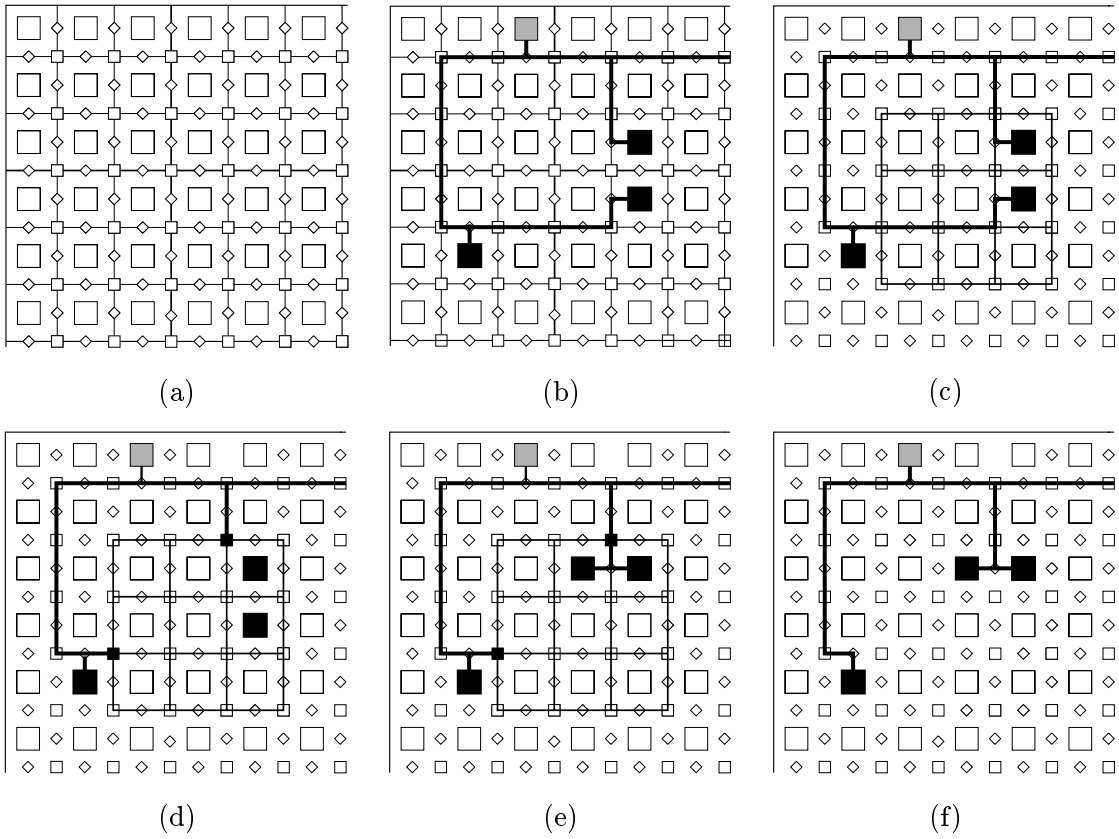
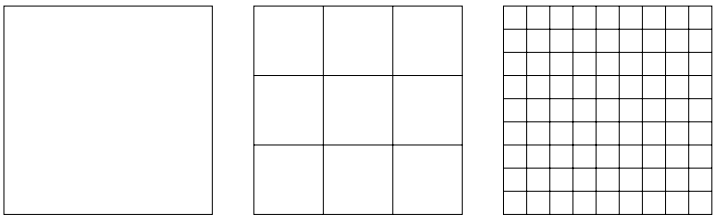


Figure 6.2: An illustration of the potential improvement of template smoothing. In (a) we see a section of the chip area and the partition of that area through two recursions. In (b) the placement and route induced for a specific net is illustrated. In (c) we mark the new portion to be routed, and in (d) we remove all information from that portion. In (e) we have applied the tool to the new portion, and in (f) we see the new resulting placement and route.

refer to A as a **level** l **Spiffy-portion**.

Recall from Chapter 5 that $v = r \cdot p \cdot q$ is the number of partitions in the partition grid. In Figure 6.3 we see an illustration of the first three levels of Spiffy-portions for a 3×3 partition. The chip itself is the only level 0 Spiffy-portion. When then partitioned into an

$r \times p \times q$ grid, each of these new v partitions is a level 1 Spiffy-portion, and each of the v^2 sub-partition of the level 1 Spiffy-portions is a level 2 Spiffy-portion.



(a) (b) (c)

Figure 6.3: An illustration of the first three levels of spiffy-portions in a 3×3 partition. (a) shows the only level 0 Spiffy-portion. (b) shows the 9 level 1 Spiffy-portions. And (c) shows the 81 level 2 Spiffy-portions.

In defining our smoothing-portions, we do not introduce partitions that would not be considered during the course of the normal Spiffy algorithm. Introducing such portions results in both the assignment of virtual terminals in unexpected places and the omission of virtual terminals along the edges of some of the Spiffy-portions, making the recursion impossible. Hence we create smoothing portions out of smaller Spiffy-portions.

Definition 6.2 *A* **level** l **smoothing-portion** *is a region of the block consisting of v level $l+1$ Spiffy-portions, to which the Spiffy algorithm is applied outside the course of the normal recursion calls. The partition grid placed on the smoothing-portion is defined by the level $l+1$ Spiffy-portions.*

In Figure 6.4 we see an chip divided into its level 2 Spiffy-portions (using a 3×3 partition), an example of a legal level 1 smoothing-portion, and an example of an illegal smoothing portion. Notice that the level 1 smoothing-portion consists of 3×3 level 2 Spiffy-portions, thus avoiding introducing new partition lines. In the third case, the marked region is not composed of 3×3 Spiffy portions. Using this marked area would both introduce

virtual terminals into the interior of the level 2 Spiffy-portions and possibly leave their edges without virtual terminals, ruining our ability to treat each Spiffy-portion recursively.

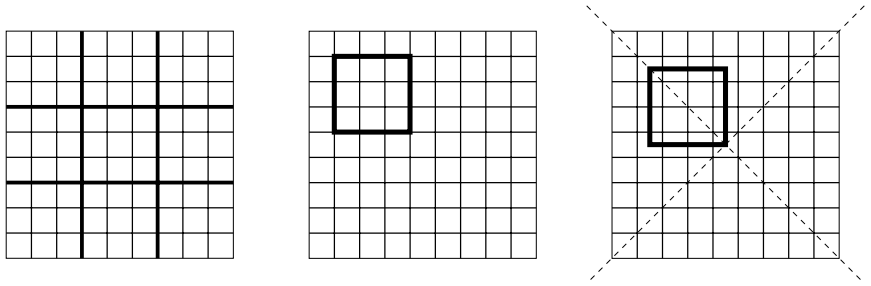


Figure 6.4: An example of a 3×3 Spiffy partition to two levels of recursion, a legal smoothing-portion, and an illegal smoothing-portion. The third example is an illegal smoothing-portion as it is not built on Spiffy-portions.

Given a level l smoothing-portion P, we can apply the Spiffy algorithm to P once we know its edges have been assigned the proper virtual nodes, something we can be assured of after we have applied the algorithm to each level l Spiffy-portions intersecting P. When the Spiffy algorithm has been applied to P at the ith level, we can recursively continue this application to each of its partitions. However, since each of these partitions is a Spiffy-portion, the algorithm is applied to them in the course of the normal Spiffy algorithm. Thus there is no need to follow these recursions.

Given this, we can now define the portion order in which the algorithm is to be applied. We begin with a standard breadth-first search: start with the unique level 0 Spiffy-portion (the chip), follow this with each level 1 Spiffy-portion, and continue with each level 2 Spiffy portion. At this point we are able to apply the algorithm to each level 1 smoothing-portion, after which we can work with the level 3 Spiffy-portions. In short, we use a breadth-first search for the Spiffy-algorithms, interspersed with the smoothing-portions after the completion of specified levels of the breadth-first search.

We still need to address exactly which of the potential smoothing-portions should be used. We expect that examining more smoothing-portions will results in higher quality

results, but will require a longer runtime of the tool. We discuss this tradeoff in Section 6.3, presenting the experimental results of different strategies.

6.2 Implementation

Given the data structures presented in Section 5.3, integration of template-smoothing into the Spiffy tool is straightforward. In order perform template smoothing, two additional tasks must be performed:

1. Smoothing-portions must be created and added into the portion list to reflect the order in which they should be routed.

2. When applying the algorithm to a smoothing-portion, all interior switch blocks of that portion must be cleared of all virtual node assignments, and all data structures must be updated accordingly.

Integrating the smoothing-portions into the list is a simple task. We create all smoothing-portions from the smaller Spiffy-portions, and add them to the list as we reach the end of each recursion level in our breath-first search of the Spiffy-portions. As each smoothing-portion can be created in $O(1)$ time, the amount of time this adds is $O(|\varsigma|v)$, where $S\varsigma$ is the set of smoothing-portions to be used.

The second item is the most computationally complex. Given a smoothing-partition P, we need to remove the virtual nodes assigned to each block within the partition. This removal is complicated by switch blocks on the edge of the partitions. Those blocks that are used by a net to enter a partition must be retained, while those used by a net within the partition must be discarded. We solve this problem by "locking" nets to a switch block side in any virtual node assignment, thus allowing us to judge if a given virtual node serves as an entry point for a given partition. However, the operations of locking a net, unlocking a net, or checking to see if a net is locked on a give switch block side is logarithmic in the

number of nets locked to the side. So the runtime is $O(\log n)$ in the worst case, where n is the number of nets.

In order to clear the virtual nodes within a smoothing-portion, we check every block of the smoothing-portion. For each of those blocks we check every net assigned to the block, and for each of those net assignments we determine if the net should be removed. If it is to be removed, we must also remove it from the net's Steiner point list and modify the net's source tree as needed. Further, if the smoothing-portion has two "entry" pins (as the example net did in Figure 6.2(c)), we need to reorganize the tree to eliminate one source node and make all exit nodes children of the remaining source node. For a level l smoothing-partition there are $\frac{b}{v^l}$ blocks to check, where b is the number of blocks on the chip. Checking and removing each net on a block requires $O(\log n)$ time, and any tree reorganization requires $O(1)$ time. As we argued in Chapter 5 that we can take v as a constant, this procedure requires $O(b \log n)$ time in the worst case, though the bound should be much smaller in the average case.

We can conclude that the cost of performing the algorithm on a smoothing-portion is increased by $O(b \log n)$. Since we do not make the recursive calls on these portions, the entire algorithm is increased by $O(|\varsigma| b \log n)$ – a term absorbed in the run-time bound of the Spiffy algorithm. Thus the worst-case bound of the Spiffy algorithm remains unchanged.

6.3 Experimental Results

In this section we examine the results of using template smoothing. We first ran a series of tests without the template smoothing feature. Using the statistics from these experiment as our base-line for comparisons, shown in Table 6.1, we then ran a series of tests with the template smoothing feature to judge the improvement in circuit mapping and corresponding increase in runtime. All experiments in this section are based upon a 3×3 partition gird, the same used in the testing in Chapter 5.

Circuit	Channel Width	Average Path Length	Runtime (seconds)
busc	8.00 ± 0.19	10.92 ± 0.20	8.62 ± 0.75
dma	8.77 ± 0.20	14.67 ± 0.18	11.73 ± 0.44
bnre	10.77 ± 0.23	16.20 ± 0.17	21.45 ± 0.34
dfsm	10.23 ± 0.22	14.38 ± 0.15	28.32 ± 0.74
9symml	8.87 ± 0.18	12.73 ± 0.31	2.07 ± 0.06
term1	7.50 ± 0.22	10.26 ± 0.38	2.23 ± 0.21
apex7	7.53 ± 0.22	10.64 ± 0.26	3.98 ± 0.37
alu2	9.85 ± 0.18	14.48 ± 0.20	5.78 ± 0.15
too_large	10.05 ± 0.20	14.53 ± 0.21	8.10 ± 0.41
example2	9.47 ± 0.33	11.24 ± 0.34	10.53 ± 0.97
vda	12.03 ± 0.23	18.87 ± 0.18	19.43 ± 1.32
alu4	11.67 ± 0.22	17.47 ± 0.13	12.80 ± 0.21

Table 6.1: The base-line results for comparison against the template smoothing experiments. Each statistic was taken over an average of 60 runs, with all confidence intervals computed at a 95% level of confidence. The path length of a net is defined to be the number of wires in the largest source-to-sink path of a net, and the length presented in this table is the average over all net lengths.

6.3.1 Smoothing Templates

While we have discussed the concept of a smoothing-portion, we have not yet discussed the *smoothing-templates*: the ordered multi-set of smoothing-portions to be used during the execution of the algorithm. The selection of the smoothing templates may have a significant effect on the final results, and in the following experiments we evaluate different possible templates on a 3×3 partition. In order to simply the experiments, we first need to define the idea of a *smoothing-class*.

Figure 6.5 shows all level 2 Spiffy-portions of an FPGA with respect to a 3×3 partition grid, and a numbering of all interior Spiffy-portions such that the numbers are symmetric with respect to both axis.

Definition 6.3 *A* **smoothing-class** ς_i *is the set of all smoothing-portions P such that the center Spiffy-portion of P is labeled i in Figure 6.5.*

	2	8	9	5	9	8	2	
	8	3	10	6	10	3	8	
	9	10	4	7	4	10	9	
	5	6	7	1	7	6	5	
	9	10	4	7	4	10	9	
	8	3	10	6	10	3	8	
	2	8	9	5	9	8	2	

Figure 6.5: An array of level 2 Spiffy-partitions of a chip, and an arbitrary symmetric numbering of each interior partition. This numbering allows us to characterize our smoothing-portions into smoothing-classes for ease of notation. ς_i is the set of all smoothing-partitions P such that the center Spiffy-portion of P is numbered i.

In Figure 6.6 we see two examples of smoothing-classes: ς_3 and ς_{10}. ς_3 contains all smoothing-portions consisting of nine Spiffy-portions arranged in a 3×3 grid, where the center Spiffy-portion is labeled with a 3 in Figure 6.5. ς_{10} is defined in much the same way. Note that the set of level 1 Spiffy-portions are equal to the union of the classes ς_2, ς_5 and ς_1.

When constructing a smoothing-template, it makes sense to include the reflections of any smoothing-portion along each axis. Hence by constructing the smoothing-template from smoothing-classes, we ensure that any templates used are closed under reflection. In the following sections, we characterize a template by its class indices. For example, the template $T = (3, 4, 1, 3)$ denotes that in ordering the smoothing-portions, we use those in ς_3, followed by those in ς_4, followed by those in ς_1, and finishing with those in ς_3. Note again that this is an ordered multi-set, hence the smoothing-template $(3, 4, 1, 3)$ is different than $(4, 3, 1, 3)$, which is different than $(4, 3, 1)$.

6.3.2 First Experimental Results

Our first goal in experimenting with smoothing-templates is to determine whether there is any value in their use. In Table 6.2 we see the results of using the smoothing-template

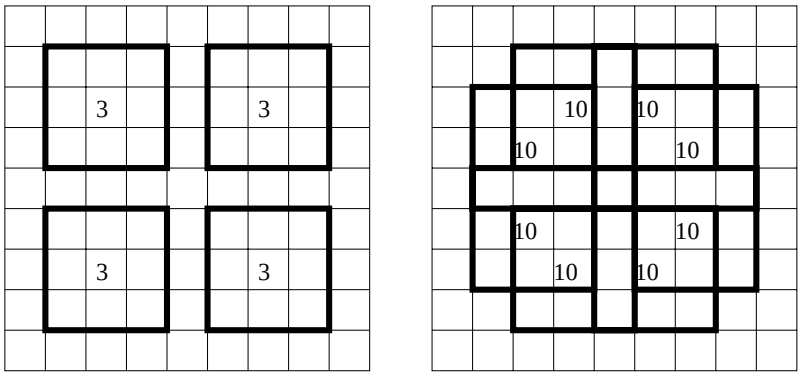


Figure 6.6: (a) The set of smoothing-portions consisting of the smoothing-class ς_3. (b) The Set of smoothing-portions consisting of the smoothing-class ς_{10}.

(8,9,3,10,6,4,7,2,5,1), a template using every smoothing-class. From these results we see that the technique does induce an improvement in circuit mappings, decreasing the channel-width by 5.97% and the path length by 7.41% over Spiffy without template smoothing. As expected, it does increase the runtime of the algorithm by a significant amount. However, as the largest benchmark still takes less than 55 seconds, the increase in circuit quality will be worth the extra run-time to many designers.

We have established that template smoothing is a technique worth pursuing. However, it is worth investigating smaller templates for designers who do not wish make the runtime-sacrifice, or are dealing with exceptionally large circuits. Thus we ran a series of experiments on different smoothing-templates.

6.3.3 Effects on Run Time

The first characteristic we look at when using a smoothing-template is the effect of its size on runtime. In Table 6.3 we see the average runtime for templates of various sizes, and

Circuit	Channel Width		Path Length		Run Time	
busc	7.87 ± 0.34	1.67%	10.33 ± 0.19	5.36%	13.17 ± 1.00	-52.80%
dma	8.47 ± 0.41	3.42%	13.52 ± 0.24	7.85%	23.73 ± 1.21	-102.27%
bnre	9.50 ± 0.38	11.76%	13.65 ± 0.23	15.71%	47.30 ± 3.62	-120.51%
dfsm	8.60 ± 0.60	15.96%	11.88 ± 0.34	17.42%	54.60 ± 4.39	-92.82%
9symml	8.80 ± 0.28	0.75%	12.12 ± 0.30	4.82%	5.23 ± 0.16	-153.23%
term1	7.17 ± 0.30	4.44%	9.65 ± 0.28	5.88%	4.70 ± 0.37	-110.45%
apex7	7.30 ± 0.30	3.10%	10.13 ± 0.26	4.72%	8.23 ± 0.63	-106.69%
alu2	9.70 ± 0.31	1.52%	13.89 ± 0.22	4.08%	13.37 ± 0.43	-131.12%
too_large	9.97 ± 0.43	0.83%	13.85 ± 0.26	4.75%	17.53 ± 0.89	-116.46%
example2	8.93 ± 0.44	5.63%	10.66 ± 0.25	5.13%	18.97 ± 1.50	-80.06%
vda	10.60 ± 0.50	11.91%	17.72 ± 0.31	6.09%	35.20 ± 5.29	-81.13%
alu4	10.43 ± 0.29	10.61%	16.22 ± 0.18	7.14%	30.68 ± 1.15	-139.68%
Average		5.97%		7.41%		-107.27%

Table 6.2: Results using the template $T = (8, 9, 3, 10, 6, 4, 7, 2, 5, 1)$, and the improvement over mappings done without template smoothing. Each average is taken over 30 runs. Each statistic represents the center of a 95% confidence with width no more than 5.39% of the value.

the corresponding percentage increase of runtime for each template size shown in Table 6.4. When we calculate the linear correlation coefficient, shown in Table 6.5, we see that in each case these coefficients are very close to 1 – indicating the percentage increase in time is very close to linear in the size of the smoothing-template. Given that they are linear, the average regression angle of the line is 63.96°, meaning that for every smoothing-portion added we can expect an increase in runtime of approximately 2.05% of the base-line runtime.

6.3.4 Effects on Routing Quality

Having determined in practice that the increase in runtime is linear in the size of the smoothing-template, we now investigate the effect of size of the smoothing-template on circuit quality. The objective depends on the needs of the user, and whether they want the fastest tool, the best mapping, or some combination of the two. In the following experiment we try a number of smoothing-templates, using the Upstart router to complete

Name	4	8	12	16	20	24
busc	8.31	8.85	9.49	9.98	10.14	11.07
dma	12.72	14.04	15.53	16.76	17.31	19.49
bnre	20.66	23.08	25.37	27.77	28.81	33.56
dfsm	27.54	30.34	33.16	35.83	36.71	41.38
9symml	2.02	2.14	2.51	2.95	3.05	3.26
term1	2.29	2.39	2.52	2.59	2.84	3.32
apex7	4.12	4.44	4.89	4.95	5.45	5.79
alu2	6.19	6.97	7.77	8.61	8.90	10.24
too_large	8.56	9.73	10.57	11.48	11.83	13.63
example2	11.31	12.09	13.02	14.11	14.81	15.17
vda	19.56	20.76	23.21	25.30	25.16	28.95
alu4	13.62	15.63	17.06	19.40	19.63	23.53

Table 6.3: The average runtime for each benchmark for different sized templates, where size is defined as the number of smoothing-portions contained in the template.

Name	4	8	12	16	20	24
busc	5.73%	12.60%	20.74%	26.97%	29.01%	40.84%
dma	10.61%	22.09%	35.04%	45.74%	50.52%	69.48%
bnre	8.97%	21.73%	33.81%	46.47%	51.95%	77.00%
dfsm	8.34%	19.35%	30.45%	40.95%	44.41%	62.79%
9symml	1.00%	7.00%	25.50%	47.50%	52.50%	63.00%
term1	19.90%	25.13%	31.94%	35.60%	48.69%	73.82%
apex7	8.14%	16.54%	28.35%	29.92%	43.04%	51.97%
alu2	12.34%	26.50%	41.02%	56.26%	61.52%	85.84%
too_large	8.35%	23.16%	33.80%	45.32%	49.75%	72.53%
example2	6.00%	13.31%	22.02%	32.24%	38.80%	42.17%
vda	4.82%	11.25%	24.38%	35.58%	34.83%	55.14%
alu4	12.47%	29.07%	40.88%	60.20%	62.10%	94.30%

Table 6.4: The increase in runtime over the base-line for different sized templates.

each mapping.

We begin by looking at each smoothing-class individually. In Table 6.6 and Table 6.7, we see the results of using only one of the smaller smoothing-classes, with actual data presented in Table 6.14 and Table 6.15 at the end of this chapter. In Table 6.8 and Table

Circuit	Linear Correlation Coefficient	Angle of Regression
busc	0.96	73.61°
dma	0.99	63.43°
bnre	0.99	60.75°
dfsm	0.99	64.71°
9symml	0.97	59.31°
term1	0.94	65.16°
apex7	0.99	69.28°
alu2	0.99	59.74°
too_large	0.98	60.90°
example2	0.99	70.09°
vda	0.96	63.06°
alu4	0.97	57.50°
Average		63.96°
σ		4.68°

Table 6.5: The linear correlation coefficient between template size and run-time increase for each benchmarks, as calculated from Figure 6.4 and additional experiments not shown.

6.9 we see the results of using one of the larger smoothing-classes. In the following tables we present other smoothing templates, with actual data present at the end of the chapter. The gain from any of these is minimal. A number of the one-class smoothing-templates actually seem to increase the channel-width, but once we increase to larger size improvements the channel-width does decrease.

Exactly which smoothing-template is best will depend on the individual circuit instance. In our experiments we tested a number of smoothing-templates, and none produced results equal to the $T = (8, 9, 3, 10, 6, 4, 7, 2, 5, 1)$ template first presented in Table 6.2.

6.4 Conclusion

In this chapter we presented template-smoothing to improve the results of the Spiffy algorithm. Having implemented the tool, we provided a characterization of the templates

Circuit	(2)	(3)	(4)	(5)	(6)	(7)
busc	1.25%	1.25%	2.50%	1.67%	3.75%	0.42%
dma	0.76%	-0.76%	1.90%	-1.90%	0.00%	1.52%
bnre	0.93%	-0.31%	1.24%	2.17%	0.00%	5.57%
dfsm	0.33%	2.61%	0.00%	4.56%	1.30%	3.58%
9symml	-3.01%	-3.38%	-0.38%	-1.88%	-1.88%	-2.26%
term1	0.89%	0.00%	-1.33%	-0.44%	-4.89%	-1.78%
apex7	-1.33%	-5.75%	0.00%	-3.10%	3.98%	0.44%
alu2	0.51%	3.55%	3.55%	3.89%	2.54%	1.18%
too_large	2.16%	1.82%	1.49%	2.82%	4.81%	0.50%
example2	5.63%	3.52%	5.63%	4.58%	3.87%	4.58%
vda	1.38%	1.11%	3.88%	2.49%	2.49%	2.49%
alu4	1.71%	-0.86%	0.00%	-0.57%	0.00%	1.71%
Average	0.93%	0.23%	1.54%	1.19%	1.33%	1.50%

Table 6.6: Percentage improvement in channel width for each smoothing template on each benchmark taken over 30 runs.%

Circuit	(2)	(3)	(4)	(5)	(6)	(7)
busc	0.91%	2.45%	2.04%	3.59%	1.56%	1.05%
dma	1.14%	0.57%	0.96%	-0.99%	0.62%	1.79%
bnre	1.43%	1.23%	0.54%	0.60%	1.78%	1.57%
dfsm	4.01%	-0.52%	0.60%	3.04%	1.94%	2.40%
9symml	-1.53%	-0.39%	0.84%	-0.05%	-2.04%	-0.29%
term1	-1.96%	-1.46%	-2.57%	1.76%	-7.27%	-1.63%
apex7	-0.57%	-2.96%	-5.03%	-4.81%	-0.75%	-3.80%
alu2	-1.54%	0.12%	-0.07%	0.12%	-0.50%	1.25%
too_large	0.45%	1.83%	-1.15%	0.49%	2.28%	0.65%
example2	5.09%	4.66%	4.51%	2.00%	2.78%	4.60%
vda	0.73%	0.30%	1.09%	-0.01%	0.68%	0.83%
alu4	-0.19%	-1.32%	0.29%	0.68%	1.43%	-0.45%
Average	0.67%	0.38%	0.17%	0.53%	0.21%	0.66%

Table 6.7: Percentage improvement in path length for each smoothing template on each benchmark taken over 30 runs.%

and presented the experimental results of using different templates. A number of these provided us with significant improvement in our circuit mapping with only a small penalty

Circuit	(8)	(9)	(10)
busc	3.33%	2.08%	1.25%
dma	0.76%	-1.90%	3.04%
bnre	2.79%	-0.31%	4.33%
dfsm	1.30%	6.84%	2.28%
9symml	0.00%	0.00%	5.26%
term1	4.00%	0.44%	-1.33%
apex7	-3.54%	2.21%	-3.54%
alu2	3.21%	2.20%	2.54%
too_large	0.83%	1.49%	4.48%
example2	3.52%	3.17%	4.93%
vda	1.66%	-0.28%	-3.05%
alu4	1.14%	4.57%	2.29%
Average	1.58%	1.71%	1.87%

Table 6.8: Percentage improvement in channel width for each smoothing template on each benchmark taken over 30 runs.%

Circuit	(8)	(9)	(10)
busc	1.25%	3.81%	3.01%
dma	2.75%	2.08%	0.99%
bnre	3.03%	3.27%	2.44%
dfsm	3.99%	4.69%	0.75%
9symml	0.51%	-3.20%	1.12%
term1	4.63%	-0.48%	-6.49%
apex7	-1.38%	0.52%	-4.90%
alu2	1.44%	0.23%	1.86%
too_large	1.14%	0.91%	-0.95%
example2	3.81%	1.31%	3.84%
vda	1.89%	1.84%	3.00%
alu4	0.12%	1.51%	-0.87%
Average	1.93%	1.38%	0.32%

Table 6.9: Percentage improvement in path length for each smoothing template on each benchmark taken over 30 runs.%

in run-time.

Circuit	(3, 6, 4, 7)	(8, 10)	(9, 10)	(8, 9)
busc	3.75%	1.25%	5.00%	3.33%
dma	-0.38%	4.18%	5.32%	3.42%
bnre	5.26%	4.33%	-0.31%	8.36%
dfsm	8.14%	7.17%	5.21%	6.19%
9symml	0.75%	-0.38%	4.14%	1.88%
term1	2.67%	4.00%	5.33%	2.22%
apex7	-1.33%	5.75%	1.77%	3.10%
alu2	4.91%	6.60%	5.58%	3.21%
too_large	3.81%	-0.50%	3.48%	3.48%
example2	2.11%	7.04%	5.99%	1.76%
vda	6.37%	-3.05%	4.43%	2.49%
alu4	2.00%	4.00%	2.29%	3.14%
Average	3.17%	3.37%	4.02%	3.55%

Table 6.10: Percentage improvement in channel width for each smoothing template on each benchmark taken over 30 runs.%

Circuit	(3, 6, 4, 7)	(8, 10)	(9, 10)	(8, 9)
busc	2.91%	3.74%	2.59%	3.05%
dma	2.05%	1.33%	0.87%	0.73%
bnre	5.56%	7.74%	6.86%	6.89%
dfsm	6.17%	5.79%	6.33%	8.17%
9symml	-0.37%	-0.65%	1.02%	2.22%
term1	0.12%	6.61%	5.54%	3.02%
apex7	-2.62%	2.28%	3.95%	0.44%
alu2	1.48%	1.99%	-0.57%	1.52%
too_large	3.56%	1.03%	3.23%	0.76%
example2	3.90%	4.89%	5.13%	1.85%
vda	3.42%	3.34%	2.51%	2.66%
alu4	1.47%	2.16%	3.05%	2.17%
Average	2.30%	3.35%	3.38%	2.79%

Table 6.11: Percentage improvement in path length for each smoothing template on each benchmark taken over 30 runs.%

Circuit	(8, 9, 10)	(10, 9, 3, 2)	(3, 10, 6, 4, 7)	(4, 3, 2, 7, 6, 5)
busc	2.50%	2.92%	2.50%	3.33%
dma	1.14%	3.42%	1.52%	3.80%
bnre	6.50%	7.12%	4.64%	8.36%
dfsm	9.77%	11.07%	0.33%	8.47%
9symml	0.75%	-1.13%	-1.50%	-0.38%
term1	1.78%	8.00%	0.89%	2.22%
apex7	1.33%	-0.89%	2.21%	-0.44%
alu2	3.55%	2.88%	3.21%	3.21%
too_large	3.81%	-0.83%	-1.16%	4.15%
example2	5.63%	7.75%	6.34%	4.93%
vda	3.88%	5.26%	8.03%	4.99%
alu4	4.57%	4.57%	0.86%	4.86%
Average	3.77%	4.18%	2.32%	3.96%

Table 6.12: Percentage improvement in channel width for each smoothing template on each benchmark taken over 30 runs.%

Circuit	(8, 9, 10)	(10, 9, 3, 2)	(3, 10, 6, 4, 7)	(4, 3, 2, 7, 6, 5)
busc	3.93%	1.42%	1.48%	4.80%
dma	4.10%	4.85%	3.72%	4.53%
bnre	9.97%	8.25%	5.34%	9.54%
dfsm	9.28%	10.56%	5.49%	10.01%
9symml	2.48%	1.58%	2.97%	4.01%
term1	2.03%	5.60%	0.31%	3.22%
apex7	0.54%	2.76%	2.47%	-1.68%
alu2	2.63%	0.62%	2.36%	2.44%
too_large	3.42%	1.30%	2.38%	3.51%
example2	6.54%	5.18%	4.08%	5.30%
vda	4.20%	4.45%	3.29%	3.63%
alu4	3.76%	3.52%	3.28%	4.53%
Average	4.41%	4.18%	3.10%	4.49%

Table 6.13: Percentage improvement in path length for each smoothing template on each benchmark taken over 30 runs.%

Circuit	None	(2)	(3)	(4)	(5)	(6)	(7)
busc	8.00	7.90	7.90	7.80	7.87	7.70	7.97
dma	8.77	8.70	8.83	8.60	8.93	8.77	8.63
bnre	10.77	10.67	10.80	10.63	10.53	10.77	10.17
dfsm	10.23	10.20	9.97	10.23	9.77	10.10	9.87
9symml	8.87	9.13	9.17	8.90	9.03	9.03	9.07
term1	7.50	7.43	7.50	7.60	7.53	7.87	7.63
apex7	7.53	7.63	7.97	7.53	7.77	7.23	7.50
alu2	9.85	9.80	9.50	9.50	9.47	9.60	9.73
too_large	10.05	9.83	9.87	9.90	9.77	9.57	10.00
example2	9.47	8.93	9.13	8.93	9.03	9.10	9.03
vda	12.03	11.87	11.90	11.57	11.73	11.73	11.73
alu4	11.67	11.47	11.77	11.67	11.73	11.67	11.47

Table 6.14: Average channel width over 30 runs on each benchmark. Each value represents the center of a 95% confidence interval with a half-width of no greater than 7.12% of its value.

Circuit	None	(2)	(3)	(4)	(5)	(6)	(7)
busc	10.92	10.82	10.65	10.69	10.52	10.75	10.80
dma	14.67	14.50	14.58	14.53	14.81	14.58	14.40
bnre	16.20	15.97	16.00	16.11	16.10	15.91	15.94
dfsm	14.38	13.81	14.46	14.30	13.94	14.10	14.04
9symml	12.73	12.93	12.78	12.63	12.74	12.99	12.77
term1	10.26	10.46	10.41	10.52	10.08	11.00	10.42
apex7	10.64	10.70	10.95	11.17	11.15	10.72	11.04
alu2	14.48	14.71	14.47	14.49	14.47	14.56	14.30
too_large	14.54	14.47	14.27	14.70	14.46	14.20	14.44
example2	11.24	10.67	10.71	10.73	11.01	10.93	10.72
vda	18.87	18.73	18.82	18.67	18.88	18.74	18.72
alu4	17.47	17.50	17.70	17.42	17.35	17.22	17.55

Table 6.15: Average path length over 30 runs on each benchmark. Each value represents the center of a 95% confidence interval with a half-width of no greater than 5.85% of its value.

Circuit	None	(8)	(9)	(10)
busc	8.00	7.73	7.83	7.90
dma	8.77	8.70	8.93	8.50
bnre	10.77	10.47	10.80	10.30
dfsm	10.23	10.10	9.53	10.00
9symml	8.87	8.87	8.87	8.40
term1	7.50	7.20	7.47	7.60
apex7	7.53	7.80	7.37	7.80
alu2	9.85	9.53	9.63	9.60
too_large	10.05	9.97	9.90	9.60
example2	9.47	9.13	9.17	9.00
vda	12.03	11.83	12.07	12.40
alu4	11.67	11.53	11.13	11.40

Table 6.16: Average channel width over 30 runs on each benchmark. Each value represents the center of a 95% confidence interval with a half-width of no greater than 9.47% of its value.

Circuit	None	(8)	(9)	(10)
busc	10.92	10.78	10.50	10.59
dma	14.67	14.26	14.36	14.52
bnre	16.20	15.71	15.67	15.80
dfsm	14.38	13.81	13.71	14.27
9symml	12.73	12.67	13.14	12.59
term1	10.26	9.78	10.31	10.92
apex7	10.64	10.78	10.58	11.16
alu2	14.48	14.27	14.45	14.21
too_large	14.54	14.37	14.40	14.67
example2	11.24	10.81	11.09	10.81
vda	18.87	18.52	18.52	18.31
alu4	17.47	17.45	17.20	17.62

Table 6.17: Average path length over 30 runs on each benchmark. Each value represents the center of a 95% confidence interval with a half-width of no greater than 10.07% of its value.

Circuit	None	(3, 6, 4, 7)	(8, 10)	(9, 10)	(8, 9)
busc	8.00	7.70	7.90	7.60	7.73
dma	8.77	8.80	8.40	8.30	8.47
bnre	10.77	10.20	10.30	10.80	9.87
dfsm	10.23	9.40	9.50	9.70	9.60
9symml	8.87	8.80	8.90	8.50	8.70
term1	7.50	7.30	7.20	7.10	7.33
apex7	7.53	7.63	7.10	7.40	7.30
alu2	9.85	9.37	9.20	9.30	9.53
too_large	10.05	9.67	10.10	9.70	9.70
example2	9.47	9.27	8.80	8.90	9.30
vda	12.03	11.27	12.40	11.50	11.73
alu4	11.67	11.43	11.20	11.40	11.30

Table 6.18: Average channel width over 30 runs on each benchmark. Each value represents the center of a 95% confidence interval with a half-width of no greater than 10.02% of its value.

Circuit	None	(3, 6, 4, 7)	(8, 10)	(9, 10)	(8, 9)
busc	10.92	10.60	10.51	10.63	10.58
dma	14.67	14.37	14.47	14.54	14.56
bnre	16.20	15.30	14.94	15.09	15.08
dfsm	14.38	13.49	13.55	13.47	13.21
9symml	12.73	12.78	12.82	12.60	12.45
term1	10.26	10.24	9.58	9.69	9.95
apex7	10.64	10.91	10.39	10.22	10.59
alu2	14.48	14.27	14.19	14.56	14.26
too_large	14.54	14.02	14.39	14.07	14.43
example2	11.24	10.80	10.69	10.66	11.03
vda	18.87	18.23	18.24	18.40	18.37
alu4	17.47	17.21	17.09	16.93	17.09

Table 6.19: Average path length over 30 runs on each benchmark. Each value represents the center of a 95% confidence interval with a half-width of no greater than 6.69% of its value.

Circuit	None	(8, 9, 10)	(10, 9, 3, 2)	(3, 10, 6, 4, 7)	(4, 3, 2, 7, 6, 5)
busc	8.00	7.80	7.77	7.80	7.73
dma	8.77	8.67	8.47	8.63	8.43
bnre	10.77	10.07	10.00	10.27	9.87
dfsm	10.23	9.23	9.10	10.20	9.37
9symml	8.87	8.80	8.97	9.00	8.90
term1	7.50	7.37	6.90	7.43	7.33
apex7	7.53	7.43	7.60	7.37	7.57
alu2	9.85	9.50	9.57	9.53	9.53
too_large	10.05	9.67	10.13	10.17	9.63
example2	9.47	8.93	8.73	8.87	9.00
vda	12.03	11.57	11.40	11.07	11.43
alu4	11.67	11.13	11.13	11.57	11.10

Table 6.20: Average channel width over 30 runs on each benchmark. Each value represents the center of a 95% confidence interval with a half-width of no greater than 5.68% of its value.

Circuit	None	(8, 9, 10)	(10, 9, 3, 2)	(3, 10, 6, 4, 7)	(4, 3, 2, 7, 6, 5)
busc	10.92	10.49	10.76	10.75	10.39
dma	14.67	14.07	13.96	14.12	14.00
bnre	16.20	14.58	14.86	15.33	14.65
dfsm	14.38	13.05	12.86	13.59	12.94
9symml	12.73	12.42	12.53	12.35	12.22
term1	10.26	10.05	9.68	10.23	9.93
apex7	10.64	10.58	10.34	10.37	10.81
alu2	14.48	14.10	14.39	14.14	14.13
too_large	14.54	14.04	14.35	14.19	14.03
example2	11.24	10.50	10.66	10.78	10.64
vda	18.87	18.08	18.03	18.25	18.19
alu4	17.47	16.81	16.85	16.89	16.68

Table 6.21: Average path length over 30 runs on each benchmark. Each value represents the center of a 95% confidence interval with a half-width of no greater than 5.29% of its value.

7

Gambit: A Tool for the Simultaneous Placement, Global Routing and Detailed Routing of FPGAs

In Chapter 4 we discussed the idea performing the stages of design automation simultaneously, and in Chapter 5 we presented a tool that simultaneously accomplishes placement and global routing. In this chapter we present Gambit: the first tool to perform placement, global routing and detailed routing simultaneously. Built on the Spiffy methodology, we incorporate a graph structure to model the detailed routing constraints. In allowing these constraints to influence the placement and global route as they are generated, Gambit produces a full circuit mapping, including both a placement and a detailed route.

7.1 FPGA Structure Revisited

Before we can explain the algorithms used in Gambit, we must first expand our explanation of the standard FPGA structure from Section 2.3. Recall the structure of a symmetric FPGA architecture (Figure 7.1). While we discussed the layout of the blocks and wires, we have not yet discussed the structure of the switch blocks.

Ideally, switch blocks could connect any two incident wires, thus guaranteeing any global route could be realized. However, this flexibility makes them far too large and complicated.

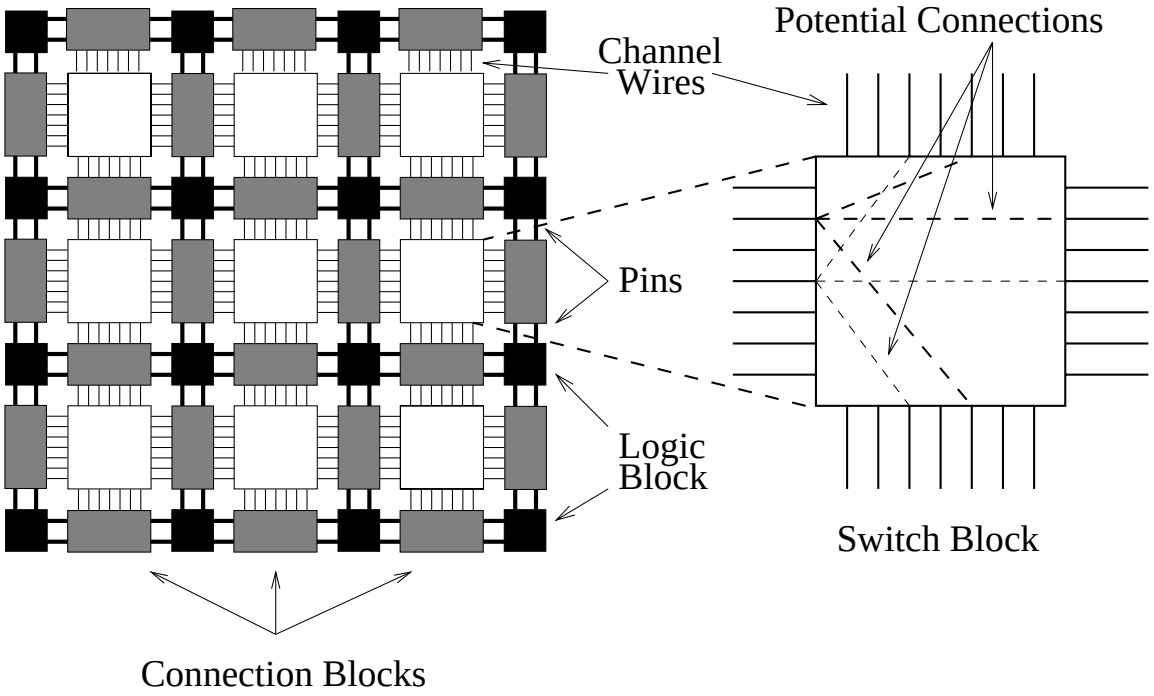


Figure 7.1: A 2D-FPGA symmetrical architecture with a channel-width of seven.

Thus switch blocks can only connect certain subsets of wires. In Figure 7.2 we see a Xilinx switch block and its *anti-symmetric* model [30]. A number of studies have been done on the minimal number of switches required to provide full routing capability, and it has been determined that this structure is a good choice [28, 30, 78, 102].

The model shown in Figure 7.2 has an important characteristic which Gambit exploits: it divides the channel-wires into equivalence classes. Consider the labeling of the wires in the diagram, and assume we extend this labeling to every channel: each channel-wire is numbered from 1 to w (the channel-width of the FPGA), starting with the left-most or top-most wire. Notice that the wires can be connected by a switch block if and only if they have the same label. As a result, if a net is routed on a wire of label i, it cannot be transferred to a wire of any other label; it must stay on wires labeled i. Thus if we think of

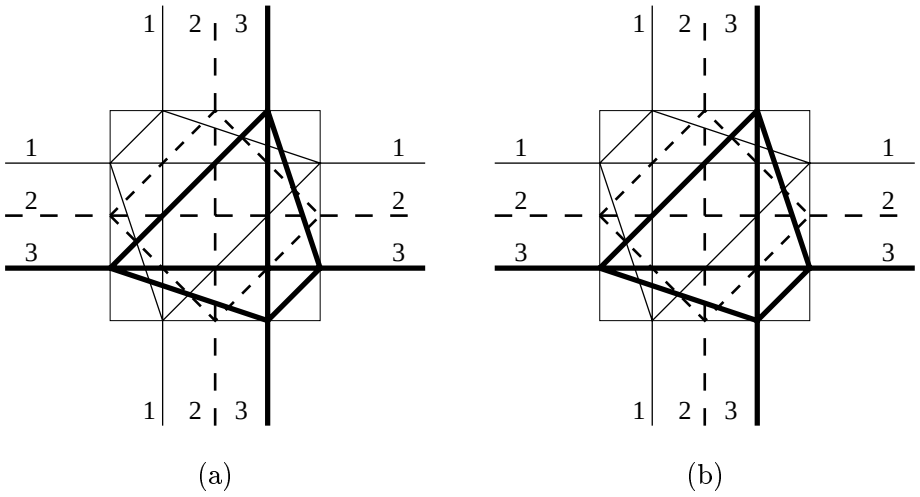


Figure 7.2: (a) A Xilinx XC4000-type switch module and (b) its switch-module model.

the wires in terms of equivalence classes, with two wires being in the same class if and only if they have the same label, we then know each net can be routed to only one equivalence class.

7.1.1 Conflict Graphs

Given a circuit design with a placement and global route already calculated, we now define a new structure:

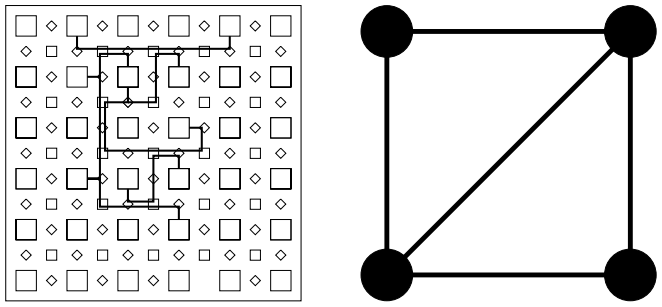
Definition 7.1 *Given a placement and global routing of some circuit, the* **Conflict Graph** $G = (\varphi, R)$*, where the set of nodes* φ *are the set of nets, while* $(n_1, n_2) \in R$ *if the global routes of* n_1 *and* n_2 *share a channel.*

Consider what it means if two nodes in G are adjacent. As the two global routes share a channel, they may not be routed to the same channel-wire within that channel, hence they

must be assigned to wires of different labels. Because we can only assign a net one label, we are mapping these nets to different equivalence classes. If we think of this mapping as a node coloring, we have our basic theorem:

Theorem 7.1 *Given the placement and global route for come circuit description and an FPGA with anti-symmetric switch blocks and channel-width w, there exists a legal detailed route of the circuit on the chip conforming to the given global route if and only if G is w-colorable.*

The proof of this is straightforward, and it immediately follows that even with the global routes, finding an optimal detailed routing problem is NP-hard for this architecture. Studies have been done concerning the structures and colorability of conflict graphs, as well as using the conflict graphs to build detailed routers [62, 103]. However, we are the first to use the graph to allow detailed routing constraint to influence the placement and global routing phases.



(a) (b)

Figure 7.3: (a) The global routings of four nets on an FPGA. (b) The corresponding conflict graph. Note that even though the global routings require of channel-width of 2, the conflict graph has a chromatic number of 3 – hence the detailed routing requires a channel-width of 3.

Figure 7.3 depicts the global routing of some circuit and the corresponding conflict graph. As that graph has a chromatic number of 3, we can conclude from Theorem 7.1 that the optimal detailed route for that global routing requires a channel-width of three, even though the global route appears to only require a width of two.

In order to incorporate detailed routing into our Spiffy tool, we construct and maintain a conflict graph during the operation of the tool. As the global routing is constructed, the graph will be updated accordingly. Thus, at any we point can determine the current minimal channel-width required for the detailed routing by checking the chromatic number of the graph. (Though in practice, we would have to do this heuristically, as finding the chromatic number of a graph is NP-complete [45].) By allowing this graph structure to influence the stages of the Spiffy algorithm, we can force the tool to construct the placement and global route in such a way as to minimize the chromatic number of the graph.

7.1.2 Doglegs

While switch blocks do not allow all possible connections between incident channel-wires, connection blocks are more flexible. In the Xilinx 4000 FPGA series, any wires incident to a connection block may be connected to any pin of a neighboring logic block [105]. As a result it is frequently assumed that two wires incident to the connection block may be connected through a pin, thus providing an *input pin dogleg*. Such a feature allows nets to switch tracks, thus invalidating our theorem and reducing the usefulness of a conflict graph.

Most FPGA design automation tools in the literature assume that input pin doglegs are allowed [6, 23, 54, 53, 59, 61, 104]; Betz and Rose note that this is not correct [15]. In order to implement input pin doglegs, a connection block must consist of independent pass transistors, each connecting a pin to a wire. However, connection blocks are considerably smaller when implemented with a multiplexer, which only allows for the one wire-connection per pin. Because of this reduction in area, commercial FPGAs use the multiplexer technology.

In the results presented in Chapters 5 and 6, Upstart assumes the availability of input

pin doglegs, as that assumption is necessary to provide a fair comparison against other tools. Our implementation of Gambit assumes that these doglegs are not available. Thus while the results of Gambit cannot be compared to those of other tools, the routings are more consistent with the actual architecture of commercial FPGAs.

7.2 Graph Coloring

In the performance of the Gambit algorithm it is frequently necessary to produce a good or optimal coloring for the conflict graph. While the conflict graphs are large enough that using an exact (exponential) algorithm is infeasible, the algorithm will occasionally need to color a small subgraph of the conflict graph exactly. Hence we need both an exact algorithm and a good heuristic.

7.2.1 Exact Coloring Algorithms

For coloring small graphs exactly, we experimented with two methods: a branch-and-bound algorithm and a linear-programming solution. In each case, the runtime of the algorithm can be improved if we can provide a small upper bound $v(G)$ on the chromatic number of the graph G. Let $\chi(G)$ denote the chromatic number of a conflict graph G. We know $\chi(G) \leq n$, where n is the number of nets in the circuit. We could set $v(G) = n$. However, the following theorem will frequently give us a better bound:

Theorem 7.2 *Any graph G has at least $\chi(G)$ nodes of degree no smaller than $\chi(G) - 1$.*

The proof of the theorem is straight forward [83].

From this theorem we can create a simple algorithm to find a bound of $\chi(G)$ that will usually be better than n. Let $d_{\max}$ be the maximum degree of any node in G. We can sort the nodes by degree in $O(d_{\max}) = O(n)$ time. We then start with the nodes of highest degree, and search backwards until we find the first value k such that there are at least k

nodes of degree $k-1$ or greater – also requiring linear time. As this value k is the largest such number for G, so we know by Theorem 7.2 that $\chi(G) \leq k$, and we set $v(G) = k$.

7.2.1.1 Branch-and-Bound

The branch-and-bound search for the optimal coloring is a standard algorithm of that type. We perform a depth-first search on the solution tree, discontinuing a path whenever the solution currently under construction uses either more colors than $v(G)$ or more than the best solution seen so far. While it is difficult to derive a meaningful runtime bound on this (or any branch-and-bound) algorithm, we do know this will examine no more than $v(G)^{n-1}$ solutions.

7.2.1.2 Integer Programming

The integer programming technique, or more precisely the 0-1 programming technique, requires that for a given problem instance, we formulate:

- A linear function $g : \{0,1\}^\kappa \leftarrow \mathbb{R}$.
- A set of linear inequalities over $\{0,1\}^\kappa$.

We then find a vector $B \in \{0,1\}^\kappa$ consistent with the constraints such that $g(B)$ is minimized and use this B to induce a solution to the graph-coloring problem.

Given a graph G of n nodes with edge set R, we will require the following variables (elements of our vector):

- u_κ, $1 \leq \kappa \leq v(G)$: u_κ will be assigned a value of 1 if and only if there is some node assigned color κ
- $v_{i,\kappa}$, $1 \leq i \leq n$, $1 \leq \kappa \leq v(G)$: $v_{i,\kappa}$ will equal 1 if and only if node n_i is assigned color κ.

Note that given a coloring of a graph, $\sum_{i=1}^{v(G)} u_i$ will be the number of colors used – hence is the function we wish to minimize. To insure that the values assigned to the variables correctly conform to the coloring, we need the following constraints:

- $\forall i, \kappa\, u_\kappa \geq v_{i,\kappa}$.: This constraint ensures every color κ used in the graph is reflected in u_κ.
- $\forall i \sum_{\kappa=1}^{v(G)} v_{i,\kappa} = 1$: This constraint ensures that every node is assigned exactly one color.
- $\forall (i,j) \in R \forall \kappa \leq v(G) : v_{i,\kappa} + v_{j,\kappa} \leq 1$: This constraint ensures no two adjacent nodes have the same color.

Any assignment of 0 and 1 values to the variables that satisfy these constraints will induce a legitimate graph coloring, and any legitimate graph coloring will induce a solution satisfying these constraints. Hence any solution satisfying these constraints that minimizes $g(B) = \sum_{i=1}^{v(G)} u_i$ will induce an optimal coloring. By using the Berkelaar's LP_Solve tool for solving integer and linear equations [13], we can solve the program for an solution.

Though integer-programming is NP-hard, it tends to work quickly in practice. However, it does not work fast enough to be useful in finding colorings for full conflict graphs, which ranges in size from 79 nodes to over 400 nodes in our benchmarks. While there are $2^{v(G)(n+1)}$ possible variable assignments, it is difficult to calculate the size of the solution space because of the third constraint. It is also worth noting that even finding a element of the solution space is NP-Hard [51], making an asymptotic analysis very difficult.

7.2.2 Coloring Heuristics

Because the problem is NP-hard, when coloring a full conflict graph we use a coloring heuristic. There has been a significant amount of work on such heuristics for a variety of applications. We have chosen to implement and experiment with two particular heuristics.

7.2.2.1 Dsatur Heuristic

The Dsatur heuristic was introduced by Brelaz in 1979, but is generally regard of one of the better simple heuristics in the literature [17]. The heuristic is based on a greedy method oriented towards the *saturation degree*: the number of different colors to which a node is adjacent. We continually pick the node of greatest saturation degree, and in case of ties choose the candidate nodes with the highest edge-degree. Having picked a node, we assign it a color and continue. There are variations on which color we assign to the node. The three most common strategies are picking the legal color most used, picking the legal color least used, or picking the first legal color from an arbitrary ordering. The algorithm runs in linear time, and there is no known non-trivial bound on its error.

7.2.2.2 Interference Graph Coloring

Interference graph coloring is frequently used by compilers for register allocation [2, 11]. It is based on the principle that the machine in question has a limited number of registers c, and thus cannot use a graph color greater than c. If allocation to a register forces the estimated chromatic number of the graph to a value higher than c, alternate provisions have to be made.

The algorithm is similar to Dsatur, but only nodes of edge-degree less than c are admitted as candidates for coloring. As each node is colored it is removed from the graph, reducing the degree of all adjacent nodes. If at any time a sub-graph is reached in which every node has edge-degree c or more, steps will have to be taken in the original input to discard edges until some node can be selected. In terms of a compiler, these steps would involve reassigning data from a register to memory, thus freeing of the register for other uses.

The situation in detailed routing is similar to that of register allocation if the designer is confined to an FPGA of channel-width of some fixed w. In that case the chromatic number of the conflict graph cannot be allowed to increase past w. If in applying the heuristic we reach a point such that an edge needs to be removed, we would have to recalculate a

portion of some net's placement and global route to accomplish this task. We do not address this issue in this dissertation, except to observe that if the input pin doglegs are allowed, we have a ready-made way of eliminating conflict-graph edges. We foresee the possibility of incorporating doglegs at this point to split a net between tracks, hence splitting the corresponding node into two nodes of lesser degree. While this approach appears to be fairly complicated, it is worth future investigation if input pin doglegs are allowed in any architecture.

We have however implemented the interference graph coloring heuristic with the idea that specification of some value w may influence the general construction of the conflict graph towards a chromatic value of w. However, in those cases where we have to violate that limit, it is our practice to increase w as necessary – hence the heuristic is essentially a variation on Dsatur.

7.3 Gambit Implementation

Gambit is based primarily on the Spiffy algorithm, using the same basic steps and techniques, but modifying and expanding them where necessary. In this section we will review the basic phases of Spiffy, and explain where the changes incorporated into the Gambit tool.

7.3.1 Data Structures and Notation

Only one new data structure has been added to the Gambit tool: a weighted-graph object representing the conflict graph, implemented as a set-based adjacency-list. Nodes are persistent objects created for each net during the initialization of the algorithm, while edges can be added during the course of the algorithm. In order to facilitate coloring, we incorporated a constant-time mapping scheme between nodes and colors based on each node's uniquely assigned label. Edge weights are required to record the number of global-route connections between two nets, information necessary for net removal when performing template smoothing.

Some other data-structures from Spiffy require minor modification. Each net holds a record of its node in the conflict-graph, making its node and node color accessible in constant time. Blocks are assigned the task of maintaining the edge structure of the conflict graph. Any two nets assigned to the same side of a switch block must share a channel, hence their nodes must be adjacent in the graph. So blocks add edges, remove edges, and modify edge weights as nets are assigned or removed from the blocks. This does raise the processing cost of adding a net to a block, or removing one from a block, from constant time to time linear in the number of nets mapped to the side of that block – $O(n)$ in the worst case.

Notation will remain consistent with that used in Section 5.4. G will denote the conflict-graph and $\phi(G)$ will denote the number of colors currently used by G. For simplification, we will not differentiate between a net and its corresponding node in the conflict graph. When we refer to a "net's color," we are referring to the color of the net's node in the conflict graph.

7.3.2 Initialization

The only added initialization cost is that of creating the conflict graph, which will begin with n nodes and 0 edges. Since there are 0 edges, $\chi(G) = 1$, hence we will assign each net the same initial color. As individual nodes can be created in constant time in conjunction with the creation of the nets, the linear time to crate the graph is absorbed in the initialization time of Spiffy.

7.3.3 Partition

As the partition phase deals with routing only in a very general sense, we left this portion of the Spiffy algorithm untouched. During this phase there are no modifications to switch blocks or connection blocks, hence there are no modifications to the conflict graph.

7.3.4 Route Selection

In the Spiffy algorithm we selected a thumbnail for each placement by building a congestion vector, and assigning thumbnails in such a way as to minimize the variance over the elements of this vector. In Gambit, we still choose a thumbnail for each net, but also assign a new color to each, or choose to leave its color unchanged. We are still interested in minimizing the congestion of the congestion vector, but minimizing the number of colors used on the net is made a higher priority.

We begin by clearing the color of all nets that will need a thumbnail, and then count the "color availability" $\alpha(e, \kappa)$ of each thumbnail edge e and color κ. That is, consider thumbnail edge e and let $S(e)$ be the set of switch blocks lying on the corresponding partition line. If $\mu(\vartheta)$ represents the colors of all nets currently used by ϑ (or, more accurately, by the two sides of ϑ relevant to the thumbnail edge), then no net assigned this thumbnail will be able to use any color in the set $\cap_{\vartheta \in S(e)} \mu(\vartheta)$. We define $\alpha(e, \kappa)$ as the number of switch blocks in $S(e)$ that do not have a net of color κ assigned to them – thus the number of nets of color κ we can assign to a thumbnail using edge e before κ becomes a member of the set $\cap_{b \in S(e)} \mu(\vartheta)$.

As before, we use a greedy algorithm in which we begin with those nets that have the least number of thumbnail choices. However, where we would arbitrarily break ties in Spiffy, here we pick the longest nets. The more switch blocks a net must transverse, the more color-conflicts it is likely to incur. Hence the longer nets will tend to have less color choices.

Given a net, we consider each possible thumbnail, and the partition edges the thumbnail crosses. We consider each thumbnail for the net, and determine if there is some color κ that can legally be assigned to the net's conflict graph node such that $\alpha(e, \kappa) > 0$ for every edge e in the thumbnail. If there is no $\kappa \leq \phi(G)$ such that this is true, we eliminate the thumbnail from consideration. Only if we eliminate all thumbnails do we introduce a new color, increasing $\phi(G)$ but allowing us to reintroduce all of the net's thumbnails as

candidates.

From this restricted set we pick our thumbnail as before: so as to minimize the variance of the elements of the congestion vector for the partial solution. In addition, we assign the net ν a color. We pick a legal color κ that will maximize the value $\min_{e \in \tau\nu} \alpha(e, \kappa)$. We do this in order to avoid "using up" any color on any edge – thus heuristically leaving more color options for future nets considering the use of thumbnails sharing these edges. Having picked a color κ, we update the values of $\alpha(e, \kappa)$ for each $e \in \tau(\nu)$, and repeat the process for the next net. Upon termination of this phase, we have a thumbnail for each net and a color for each node in the conflict graph.

Where the asymptotic runtime of the route selection in Spiffy was linear in the number of nets, the extra work required to heuristically minimize $\chi(G)$ will raise that runtime. Recall from Chapter 5 that the number of nodes v in the partition graph G is not considered a parameter of the input. Nor is the size of the edge-set E. As a result, for any point set V', the number of minimum rectilinear Steiner arboresences $\Lambda(V')$ is also independent of the input, as is the bound on that number $\Lambda(G)$. The number of nets n and the number of blocks b are the only variables we are concerned with in terms of input size.

Initially we must calculate the values $\alpha(e, c)$ for every $e \in E$ and every color κ, noting that the number of colors is bounded by the number of nodes n. To calculate $\alpha(e, \kappa)$ for all e we will have to check every switch block lying on a partition edge to count the number that can use color κ. We check if a given block can use color κ in $\log n$ time, so this stage takes $O(b \log n)$ time.

In sorting the nets, no net may have more than $\Lambda(G)$ thumbnails, and no thumbnail can have more than $v - 1$ edges. With these bounds on the sorting keys, we can sort the nets in $O(n\Lambda(G)(v-1)) = O(n)$ time. For each net we must examine the $O(\Lambda(G))$ thumbnails to determine which colors each can be legally assigned. Using the α function this can be done for a thumbnail edge in constant time, so for the entire thumbnail in $O(v)$ time, since the thumbnail has nor more than $v - 1$ edges. For a net with k thumbnails the process

can be done in $O(\Lambda(G)cvk)$ time, which is linear in the size of the input. We then pick an admissible thumbnail as before, and pick a color in $O(\kappa) = O(n)$ time. Once we have picked the thumbnail and color, updating the α function will require constant time.

Complexity Analysis 7.1 *The worst-case time complexity for the route selection phase of Gambit is* $O(n^2)$.

As noted before, this is a very loose analysis. Because κ will generally be much less than n, it is likely the average-case bound on the heuristic is much better.

7.3.5 Virtual Terminal Assignment

The virtual terminal assignment for Gambit is very close in form to the virtual terminal assignment for Spiffy. The only additional constraint is that we must ensure each net gets assigned to a switch block that can accommodate the net's color, and that no two nets of the same color are assigned to the same switch block.

Recall from the Spiffy algorithm that the problem was solved with a minimum-cost perfect matching problem. We created a bipartite graph P with a node for each net, a set of nodes for each available switch block, and added edges between net-nodes and block-nodes with weights reflecting the distance between the net's terminals and the block. We ensured that no switch block is overloaded by limiting the number of nodes in P corresponding to any switch block.

In Gambit the limit on $\phi(G)$ will implicitly limit switch block overloading; there is no need to explicitly deal with the issue. Thus when creating nodes in P for each switch block, instead of limiting the number of nodes by the ratio, we create one node for each color that can be used by the block. As P must be a complete bipartite graph, we must add an edge from each of the net nodes to each of the switch block nodes. However, for any edge that connects a net of color κ with a switch block node of color other than κ we add an infinite penalty to the weight of the edge. From the manner in which we assigned our net colors, we

know that if there are k nets using color κ then there must be k switch blocks with color κ available, hence each net can be assigned to a switch block without incurring the penalty.

7.3.6 Graph Recoloring

While all the steps taken in the route selection and virtual terminal assignment phases are done with the aim of minimizing $\phi(G)$, there is no assurance that we have the best graph coloring possible. Hence there is some value in recoloring G with a more efficient heuristic after the virtual terminal assignment. In later recursions many of the nodes are unaffected by any of these stages as they may not exist with the portion of the chip under consideration. We have chosen to only recolor the subgraph of G affected by the previous steps on most recurrences. Given that we are performing a breadth-first search, we recolor the entire graph only when we starting a new level in the tree.

7.3.7 Base Case

While the base case can be handled in much the same way as in Spiffy, we do make modifications to the integer program used. Specifically, we combine Ganley's solution with the graph color technique presented in Section 7.2.

In creating the constraints of the integer program, we introduce all variables and constraints needed in Ganley's solution, as well as all variables and constraints used in our graph coloring solution. One additional set of constraints is needed to provide the correct interaction. In Ganley's formulation, he uses a series of 0/1 variables denoted $f_{k,i}$, which are assigned a value of 1 if and only if net k is assigned to channel i. We need to ensure in our solution that if $f_{k,i} = 1$ and $f_{m,i} = 1$, indicating that nets k and m have been assigned to the same channel, than they are given a different color. We enforce this with the constraint $f_{k,i} + f_{m,i} + v_{k,c} + v_{m,c} \leq 3$ for all $c \leq v(G)$. Thus if the two nets are assigned to the same channel they cannot both use color c, but otherwise are free to be the same color. We create a new minimization function for the linear program that is a linear combination of the

optimization functions from original each integer program, prioritizing color minimization over the minimization of net length.

In general we will not consider the entire conflict graph in the solution of a single block, as we do not want to take the time to do a exact graph coloring of the full graph. Instead we consider only the subgraph relevant to the portion being considered, and the color constraint imposed by nodes adjacent to this subgraph. However, the number of instance of our additional constraint is still fairly large. With k nets and, in Ganley's formulation, 8 channels, there will be $\left(\frac{k}{2}\right) 8 \cdot v(G)$ instances of this constraint. Even with small values of k the problem instance because quite large, causing the LP_Solve to take a considerable amount of time in deriving a solution for a single logic block.

7.4 Asymptotic Analysis

In the analysis of Gambit, there are only a few changes in terms of the worst-case asymptotic runtime. The route selection phase has changed from $O(n)$ to $O(n^2)$. Assigning the virtual terminals will require $O(n^{\frac{5}{2}})$ time as before. However, as we now require $O(n)$ time to assign a net to a virtual terminal, the processing time of each net is raised to $O(n^2)$. This is absorbed in the $O(n^{\frac{5}{2}})$ term, and will not affect our analysis. Finally, in each recursion we will have to recolor a portion, or all of, the graph. As this depends on the heuristic or algorithm used, we will donate the runtime of this action as $g(n)$.

Thus the only changes to the recurrence 5.1 are the addition of an $O(n^2)$ term for the base-case, and an $O(g(n))$ term for the graph-heuristics. As long as $g(n) = O(n^{\frac{5}{2}})$, which it is for our graph heuristics, these changes will make no different to the asymptotic runtime of the algorithm.

Complexity Analysis 7.2 *Taking the size of the partition graph as a constant and the net-size and chip-sizes as input parameters, and $g(n)$ as the run-time of the graph-coloring*

heuristic employed, the Gambit Algorithm has a worst-case runtime bound of:

$$O(bf(n) + (bn^2 \log b + nb^2 + b^3) \log n + b^2 + n^{\frac{5}{2}} b + bg(n))$$

where $f(n)$ is the runtime bound of the base-case.

7.5 Experimental Results

Gambit was implemented as a proof-of-concept. Raising the quality of Gambit's results to a competitive level will require further research. However, our results indicate that such research is worthwhile.

In Table 7.1 we present the results of using the Gambit algorithm with the Dsatur heuristic [17], and in Table 7.2 we present the results of using the algorithm with the graph-interference heuristic. Clearly the later heuristic is superior in practice, significantly increasing quality with very little increase in runtime.

Circuit	Average Channel-Width	Best Channel-Width	Runtime (seconds)
busc	20.47 ± 0.6	18	31.36 ± 1.1
dma	26.20 ± 0.57	24	76.37 ± 2.0
bnre	36.53 ± 0.6	33	180.90 ± 3.4
dfsm	32.13 ± 0.80	29	179.43 ± 4.8
9symml	21.43 ± 0.59	19	21.3 ± 1.4
term1	16.10 ± 1.1	12	11.13 ± 1.5
apex7	17.60 ± 0.9	14	19.83 ± 2.1
alu2	30.30 ± 0.78	27	63.57 ± 1.6
too_large	30.00 ± 0.7	27	76.97 ± 1.4
example2	24.60 ± 1.0	21	53.03 ± 3.0
vda	42.33 ± 0.7	39	141.7 ± 2.4
alu4	42.70 ± 0.8	38	149.40 ± 1.9

Table 7.1: The channel-width produced by Gambit for each benchmark (as specified by the chromatic number of the conflict graph), and the runtime of the algorithm in second when using the Dsatur Heuristic [17]. All averages are over thirty runs.

Circuit	Average Channel-Width	Best Channel-Width	Runtime (seconds)
busc	14.2 ± 0.58	12	32.46 ± 1.6
dma	17.60 ± 0.5	15	77.47 ± 2.0
bnre	23.63 ± 0.5	21	186.73 ± 3.6
dfsm	20.23 ± 0.6	18	191.63 ± 4.7
9symml	16.23 ± 0.6	14	21.2 ± 1.34
term1	11.2 ± 0.7	9	9.67 ± 1.3
apex7	12.97 ± 0.8	11	20.56 ± 2.3
alu2	21.93 ± 0.6	19	65.87 ± 2.0
too_large	20.03 ± 0.7	18	78.80 ± 2.12
example2	18.06 ± 0.9	15	56.10 ± 2.7
vda	28.47 ± 0.5	26	144.33 ± 2.0
alu4	29.16 ± 0.7	27	153.92 ± 2.90

Table 7.2: The channel-width produced by Gambit for each benchmark (as specified by the chromatic number of the conflict graph), and the runtime of the algorithm in second using the graph interference heuristic. All averages are over thirty runs.

In Table 7.3 we see Gambit compared to several of the tool suites discussed in Section 5.6.4. Note that this is not a fair comparison, as those tools allow input pin doglegs where Gambit does not, allowing for more compaction of nets into channels. Clearly, Gambit is not the superior tool. However, observe that Gambit does tie or beat each of these tool suites for at least once benchmark.

7.6 Conclusion

Although in implementing Gambit we have not produced a tool that is generally competitive with others in our tool suite, we have produced a proof of concept: conflict graphs can be used to integrate the placement, detailed routing and global routing into a simultaneous execution.

There are two specific areas of research worth following up in our goal to make Gambit a quality tool. The first is in the area of graph colorings. The best algorithms to color

Global R.	LocusRoute		GBP	OCG	Gambit
Detailed R.	CGA	SEGA			
9symml	9	9	9	9	14
alu2	12	10	11	9	19
alu4	15	13	14	12	27
apex7	**13**	**13**	**11**	10	11
example2	**18**	**17**	13	12	15
term1	**10**	**9**	**10**	**9**	9
too_large	13	11	12	11	18
vda	14	14	13	11	26
Total	104	96	93	83	139

Table 7.3: Comparison of channel-widths resulting from the Altor placement tool [80] paired with various route tools, compared against the Gambit tool with the interference graph heuristic. All scores shown are the best out of all trials. Tools for scores other than Gambit are taken from the paper on VPR [15].

classes of graphs exploit specific characteristics of those graphs. If we can find a common characteristic of conflict graphs, we may be able to devise a heuristic with a bound on solution quality, or even a polynomial time algorithm that finds exact solutions for graphs of the given characteristics. Coudert used such a technique to provide a polynomial algorithm for the exact coloring of *perfect graphs* [37]. His argument that all "real life" graphs have the characteristic of being perfect actually provided part of the initial thought process leading to Gambit. However, conflict graphs provide an example of a real life graph that is not perfect.

The second area of research is in the base case of the recursion, the so-called "miniature routing problem". While the integer programming technique works well for Spiffy, it does not lend itself to the added complexity produced by the conflict graph. In order to improve the results of Gambit, an alternative method must be found of routing around the base-case will minimizing $\chi(G)$.

8

Circuit Mappings for 3D-FPGAs

In Chapter 5 we discussed our implementation of the Spiffy algorithm and showed that when paired with the Upstart router, the tool suite produces quality circuit mappings for traditional FPGA architectures. However, both tools are also designed to process circuit instances for the 3D-FPGA architectures. With the ability to create mappings for the 3D architecture, we can now experimentally justify the arguments made in Chapter 3.

In Section 8.1 we discuss the conceptual challenges of generalizing each of the tools to the third dimension. In Section 8.2 we compare 2D mappings with 3D mappings, which shows that the move to the third dimension does result in an improvement in circuit quality. In Section 8.3 we examine the routings produced by different partition grids for 3D-FPGAs, and in Section 8.4 we explore the effects of template smoothing on the 3D mappings. Finally, in Section 8.5 we consider the cases where the 3D-FPGA is provided with only a subset of the possible vertical connections, cases Spiffy is also equipped to handle.

8.1 Spiffy and Upstart for 3D-FPGAs

While the implementation details of Spiffy for the 3D-FPGAs are not completely straightforward, the basis for the Spiffy algorithm remains unchanged. We place an $r \times p \times q$ partition grid on the chip, perform each stage of the algorithm as with the 2D architecture,

and recurse through the partitions. Template smoothing is implemented analogously, although the smoothing-classes vary with the different partition girds. We note that because Upstart is a graph-based router, it easily generalizes to the third dimension.

8.2 2D-FPGAs vs 3D-FPGAs

In this section we compare circuit mappings of the two architectures with the intent to show that 3D-FPGAs do lead to higher-quality results. Using the Spiffy/Upstart tool suite, we map traditional benchmarks to variations of 3D-FPGA architectures and compare the resulting channel widths and net lengths. However, it is very important that we conduct our experiments in such a way as to ensure that any gains in the 3D-architecture are due to the extra dimensions, and not because of extra resources or differences in the performance of the algorithm on that architecture.

In order to ensure that the mapping tool operates at the same level on each architecture, we must find a grid partition for the 3D architecture comparable to the $1 \times 3 \times 3$ partition used for the 2D architecture. Computationally, we are limited to using a grid of dimensions no larger than $3 \times 2 \times 2$. But by using a $3 \times 2 \times 2$ partition grid to map our 3D-circuits, while 2D circuits are mapped with a 3×3 partition grid, we leave ourselves open to the possibility that the greater number of partitions in the 3D architectures was responsible for any gain in circuit quality. Hence we will limit ourselves to a $2 \times 2 \times 2$ partition grid – giving the 2D architecture a slight advantage. Because we are dealing with benchmarks whose x and y dimensions are greater than the z dimension, we will eventually reach recursion levels where the block-portions consist of only one level. At this point, Spiffy will automatically switch back to the $1 \times 3 \times 3$ partition, thus slightly offsetting the advantage of the 2D mappings.

In order to compare the two architectures we must ensure that the amount of resources provided to each architecture is equal. If one architecture is provided with more blocks on which to map a benchmark than the other, that architecture will have an advantage that is not a product of its dimension: there is more room to place and route elements, leading to

better circuit mappings. If we are to argue that 3D-FPGAs are superior, we must ensure that they are not provided with more resources than on the 2D architecture.

In Section 8.2.1, we compare our mappings of standard benchmarks between 2D-FPGAs and 4-level 3D-FPGAs such that the two architectures are provided with exactly the same number of logic blocks. In many cases the number of logic blocks provided is larger than required by the benchmark, but as this will be equally true on both chips we believe neither architecture will receive an advantage.

In Section 8.2.2 we take our standard mappings for the 2D-FPGAs and compare them with mappings to 3D-FPGAs of 2 to 4 levels. Here we make our best effort to keep the resources the same, though the smallest cube with enough blocks to hold the circuit sometimes has slightly more logic blocks than its 2D counterpart. However, such experiments give us a sense of the gain resulting from the addition of levels to the FPGA.

8.2.1 Holding Chip Size Constant

In our first experiment, we take each benchmark and map it to two FPGA's of the same size: a standard FPGA that is the smallest square of even parameters that can hold the circuit, and a four-level 3D-FPGA with each side of each level cut in half. For a circuit requiring k logic blocks, let c be the smallest even integer such that $c \geq \sqrt{k}$. We then map the circuit to a standard FPGA of size $c \times c$ and to a 3D-FPGA of size $4 \times \frac{c}{2} \times \frac{c}{2}$.

In Table 8.1 we see the average results of this mapping. Moving to the third dimension proves beneficial for both channel-width and the average net length, with a 22.31% improvement in the first, and a 11.23% improvement in the second. However, since we are not trying to rate the effectiveness of our tool, but rather the effectiveness of our architecture, it is more important to consider the best mapping that can be achieved on the architecture. In Table 8.2 we see these statistics. On average the best circuit mapping over a 3D-FPGA improves the best channel-width on a 2D-FPGA by 18.97%, while the net length is improved by 9.31%.

Circuit	Size	Channel-Width			Net Length		
		2D	3D	%Δ	2D	3D	%Δ
busc	152	9.30 ± 0.66	7.50 ± 0.36	19.35%	20.56 ± 1.81	19.84 ± 1.11	3.49%
dma	261	8.83 ± 0.43	7.73 ± 0.34	12.45%	23.68 ± 1.01	21.67 ± 0.56	8.48%
bnre	390	10.77 ± 0.38	8.93 ± 0.37	17.03%	24.12 ± 0.29	22.65 ± 0.97	6.11%
dfsm	451	15.60 ± 1.70	10.33 ± 0.60	33.76%	37.28 ± 4.32	27.02 ± 2.39	27.51%
9symml	80	10.10 ± 0.38	7.47 ± 0.25	26.07%	24.36 ± 0.30	19.99 ± 0.24	17.93%
term1	86	8.80 ± 0.52	6.83 ± 0.33	22.35%	17.57 ± 1.09	15.01 ± 0.64	14.56%
apex7	113	8.80 ± 0.69	7.87 ± 0.45	10.61%	20.02 ± 1.82	19.55 ± 0.73	2.33%
alu2	159	10.53 ± 0.51	8.73 ± 0.39	17.09%	25.63 ± 1.45	24.46 ± 0.98	4.57%
too_large	189	12.80 ± 0.95	9.23 ± 0.35	27.86%	26.56 ± 1.88	22.95 ± 1.00	13.60%
example2	164	10.10 ± 0.85	7.33 ± 0.28	27.39%	18.25 ± 1.62	14.88 ± 0.81	18.46%
vda	262	12.80 ± 0.56	8.89 ± 0.28	30.56%	30.77 ± 1.89	25.16 ± 0.62	18.25%
alu4	264	11.40 ± 0.35	10.03 ± 0.50	11.99%	27.25 ± 0.87	28.92 ± 1.59	-6.09%
Average				22.31%			11.23%

Table 8.1: A comparison of the channel-widths and net lengths of 3D-FPGA mappings compared against the 2D-FPGA holding the FPGA size constant. An FPGA of size k was mapped to a 2D-FPGA of size $c \times c$ and a 3D-FPGA of size $4 \times \frac{c}{2} \times \frac{c}{2}$, where c is the smallest even integer such that $c \geq \sqrt{k}$. Each number is the average of 30 runs, given with a 95% confidence interval. Note that as there is no detailed router capable of optimizing net length for 3D-FPGAs, we instead use net length to estimate performance.

Note the added significance in the improvement in net length, as this is accomplished after the reduction of channel-width. While channel-width is our primary concern, the reduction of channel-width leads to a reduction of resources and should therefore lead to an increase in net lengths. With an 19% reduction in channel-width, we would expect net lengths to increase significantly. Hence our 9% reduction in net lengths further validated our expectations of 3D-FPGAs.

8.2.2 Varying the Layers

In our next set of experiments, we vary the number of layers of the 3D-FPGA. Mapping each benchmark to architectures of comparable resources and ranging from one to four levels, we compare the average and best results for each benchmark in terms of channel-width and net

	Channel-Width			Net Length		
Circuit	2D	3D	%Δ	2D	3D	%Δ
busc	7	6	14.29%	15.03	14.18	5.64%
dma	8	7	12.50%	21.71	20.04	7.70%
bnre	9	7	22.22%	22.78	20.53	9.88%
dfsm	9	7	22.22%	21.40	20.29	5.20%
9symml	8	6	25.00%	22.22	18.29	17.66%
term1	6	6	0.00%	11.81	11.39	3.56%
apex7	6	5	16.67%	14.46	13.54	6.37%
alu2	9	7	22.22%	21.89	19.38	11.47%
too_large	9	7	22.22%	19.42	17.44	10.21%
example2	8	6	25.00%	14.04	12.85	8.47%
vda	11	8	27.27%	27.23	23.12	15.10%
alu4	10	8	20.00%	25.60	22.69	11.34%
Average			18.97%			9.31%

Table 8.2: A comparisons of the runs from Table 8.1, using the best of 30 runs for each.

length. Here we use the standard size architecture for each benchmark, and a 3D-FPGAs of two to four levels with approximately the same amount of resources. The exact size of each architecture for each benchmark can be seen in Table 8.3.

Name	1 Layer	2 Layers	3 Layers	4 Layers
busc	$1 \times 13 \times 12$	$2 \times 9 \times 9$	$3 \times 8 \times 7$	$4 \times 7 \times 6$
dma	$1 \times 18 \times 16$	$2 \times 12 \times 12$	$3 \times 10 \times 10$	$4 \times 9 \times 8$
bnre	$1 \times 22 \times 21$	$2 \times 16 \times 15$	$3 \times 13 \times 12$	$4 \times 11 \times 11$
dfsm	$1 \times 23 \times 22$	$2 \times 16 \times 16$	$3 \times 13 \times 13$	$4 \times 12 \times 11$
9symml	$1 \times 11 \times 10$	$2 \times 8 \times 7$	$3 \times 7 \times 6$	$4 \times 6 \times 5$
term1	$1 \times 10 \times 9$	$2 \times 8 \times 6$	$3 \times 6 \times 5$	$4 \times 5 \times 5$
apex7	$1 \times 12 \times 10$	$2 \times 8 \times 8$	$3 \times 7 \times 6$	$4 \times 6 \times 5$
alu2	$1 \times 15 \times 13$	$2 \times 10 \times 10$	$3 \times 9 \times 8$	$4 \times 7 \times 7$
too_large	$1 \times 15 \times 14$	$2 \times 11 \times 10$	$3 \times 9 \times 8$	$4 \times 8 \times 7$
example2	$1 \times 14 \times 12$	$2 \times 10 \times 9$	$3 \times 8 \times 7$	$4 \times 7 \times 6$
vda	$1 \times 17 \times 16$	$2 \times 12 \times 12$	$3 \times 10 \times 9$	$4 \times 9 \times 8$
alu4	$1 \times 19 \times 17$	$2 \times 13 \times 13$	$3 \times 11 \times 10$	$4 \times 9 \times 8$

Table 8.3: The size of the architectures used for each benchmark.

Table 8.4 shows the average and best channel-width resulting from thirty mappings of each benchmark to each architecture, and Table 8.5 shows the corresponding percentage improvement for each 3D-FPGA benchmark mapping over the standard FPGA benchmark mapping.

	2 layers		3 layers		4 layers	
Circuit	Average	Best	Average	Best	Average	Best
busc	6.80 ± 0.45	6	7.30 ± 1.22	6	7.00 ± 0.75	6
dma	7.90 ± 0.79	6	7.40 ± 0.60	6	7.90 ± 1.14	6
bnre	9.80 ± 0.81	9	9.80 ± 0.66	9	9.20 ± 0.45	8
dfsm	9.00 ± 0.34	8	10.20 ± 1.38	8	8.50 ± 0.61	7
9symml	7.80 ± 0.56	7	6.90 ± 0.41	6	6.80 ± 0.45	6
term1	8.60 ± 0.90	6	7.80 ± 0.74	7	7.00 ± 0.83	5
apex7	6.50 ± 0.51	6	7.40 ± 0.90	5	7.40 ± 0.69	5
alu2	8.10 ± 0.63	7	8.70 ± 0.96	7	8.90 ± 1.04	7
too_large	9.10 ± 0.71	8	9.40 ± 0.97	7	7.80 ± 0.45	7
example2	8.40 ± 0.97	7	8.30 ± 0.76	7	8.30 ± 0.68	7
vda	9.80 ± 0.66	9	10.30 ± 1.17	8	8.70 ± 0.35	8
alu4	10.30 ± 0.48	9	10.30 ± 0.90	8	9.30 ± 0.76	8

Table 8.4: The average and best channel-widths produced when mapping each benchmark to architectures ranging from 2 to 4 levels. The statistics are taken over 30 runs for each benchmark, and intervals are at the 95% level of confidence.

Clearly the addition of each level results in improvements. While a few benchmarks do suffer with addition of a second level, the overall improvement is impressive. At four levels we achieve on average a 19% improvement in the best channel-width found.

In Table 8.6 and Table 8.7 we see the best net-lengths achieved for each benchmark. As the channel-width has been decreased we would expect net length to increase dramatically. However, net length increases only by a small percentage for the 2- and 3- level FPGAs, and actually decreases for the 4-level FPGAs.

Circuit	2 layers		3 layers		4 layers	
	Average	Best	Average	Best	Average	Best
busc	15.00%	14.29%	8.75%	14.29%	12.50%	14.29%
dma	9.92%	25.00%	15.62%	25.00%	9.92%	25.00%
bnre	9.01%	0.00%	9.01%	0.00%	14.58%	11.11%
dfsm	12.02%	11.11%	0.29%	11.11%	16.91%	22.22%
9symml	12.06%	12.50%	22.21%	25.00%	23.34%	25.00%
term1	-14.67%	0.00%	-4.00%	-16.67%	6.67%	16.67%
apex7	13.68%	0.00%	1.73%	16.67%	1.73%	16.67%
alu2	17.77%	22.22%	11.68%	22.22%	9.64%	22.22%
too_large	9.45%	11.11%	6.47%	22.22%	22.39%	22.22%
example2	11.30%	12.50%	12.35%	12.50%	12.35%	12.50%
vda	18.54%	10.00%	14.38%	20.00%	27.68%	20.00%
alu4	11.74%	10.00%	11.74%	20.00%	20.31%	20.00%
Average	10.48%	10.73%	9.19%	14.36%	14.83%	18.99%

Table 8.5: The percent improvement in channel width over the 2D architecture for each multi-layered architecture.

Circuit	2 layers		3 layers		4 layers	
	Average	Best	Average	Best	Average	Best
busc	15.89 ± 1.03	14.89	15.90 ± 1.88	14.19	15.74 ± 1.39	13.54
dma	22.95 ± 0.67	21.23	22.46 ± 0.52	21.66	21.32 ± 0.89	19.87
bnre	26.98 ± 1.03	25.44	25.42 ± 0.73	23.73	23.73 ± 0.50	22.80
dfsm	24.37 ± 0.61	23.29	23.02 ± 0.47	22.13	21.89 ± 0.66	20.37
9symml	21.06 ± 0.87	18.38	21.29 ± 0.43	20.39	20.71 ± 0.65	18.75
term1	17.88 ± 1.18	14.48	16.30 ± 0.57	14.64	13.80 ± 1.21	11.66
apex7	16.33 ± 1.58	14.50	17.35 ± 1.82	13.80	14.50 ± 1.13	13.09
alu2	22.62 ± 0.81	21.54	21.45 ± 0.47	20.18	22.45 ± 1.57	19.76
too_large	21.07 ± 1.36	19.81	22.40 ± 2.55	18.48	18.46 ± 0.40	17.74
example2	16.20 ± 1.49	14.36	15.48 ± 1.85	12.77	15.45 ± 1.83	12.85
vda	26.87 ± 0.61	25.56	27.34 ± 2.29	24.16	23.47 ± 0.41	22.91
alu4	27.26 ± 0.32	26.63	27.07 ± 2.18	25.01	24.55 ± 1.61	22.85

Table 8.6: The average and best net lengths produced when mapping each benchmark to architectures ranging from 1 to 4 levels. The statistics are taken over 30 runs for each benchmark, and intervals are at the 95% level of confidence.

	2 layers		3 layers		4 layers	
Circuit	Average	Best	Average	Best	Average	Best
busc	-1.86%	-3.62%	-1.92%	1.25%	-0.90%	5.78%
dma	-1.10%	1.03%	1.06%	-0.98%	6.08%	7.37%
bnre	-5.89%	-7.66%	0.24%	-0.42%	6.87%	3.51%
dfsm	-4.86%	-8.93%	0.95%	-3.51%	5.81%	4.72%
9symml	-2.53%	-1.66%	-3.65%	-12.78%	-0.83%	-3.71%
term1	-31.47%	-23.55%	-19.85%	-24.91%	-1.47%	0.51%
apex7	-8.43%	-6.07%	-15.21%	-0.95%	3.72%	4.24%
alu2	0.44%	-1.03%	5.59%	5.35%	1.19%	7.32%
too_large	-0.96%	-2.01%	-7.33%	4.84%	11.55%	8.65%
example2	-8.36%	-6.29%	-3.55%	5.48%	-3.34%	4.89%
vda	1.21%	0.47%	-0.51%	5.92%	13.71%	10.79%
alu4	-1.56%	-5.97%	-0.86%	0.48%	8.53%	9.07%
Average	-5.45%	-5.44%	-3.75%	-1.69%	4.24%	5.26%

Table 8.7: The percent improvement in net length over the standard architecture for each multi-layered architecture.

8.3 Changing the Partition Dimensions

As we discussed in Chapter 5, it is theoretically possible to increase the dimensions of the partition grid – presumably leading to superior results. However, in practice any grid with more that 12 partitions is infeasible, leave us only one alternative to the $2 \times \times 2 \times 2$ partition: a $3 \times 2 \times 2$ partition. Recall that when we reach a portion consisting of a plane, we revert back to a $1 \times 3 \times 3$ partition grid.

In Tables 8.8, 8.9 and 8.10 we see the results of mapping the benchmarks to architectures of varying levels using the $3 \times 2 \times 2$ partition grid. In all cases the average runtime is increased dramatically for very little (when any) gain in quality. As the percentage increase in runtime appears to be inversely proportional to the size of the circuit, we can attribute part of it to the large start-up cost of loading the additional Steiner-tree structures. Hence for larger benchmarks the large partition my be of use. In these examples it is not worthwhile.

Circuit	Channel-Width		Net Length		Runtime	
	Average	%Δ	Average	%Δ	Average	%Δ
busc	7.17 ± 0.34	-5.39%	16.17 ± 0.66	-7.69%	6.87 ± 0.73	-8.99%
dma	7.90 ± 0.27	0.00%	22.91 ± 0.27	-7.89%	13.17 ± 0.61	-17.56%
bnre	9.90 ± 0.36	-1.02%	27.45 ± 0.25	-6.08%	21.67 ± 0.42	-12.85%
dfsm	9.13 ± 0.36	-1.48%	24.33 ± 0.35	-0.84%	27.77 ± 0.83	-6.80%
9symml	7.57 ± 0.25	2.99%	20.41 ± 0.50	2.67%	3.00 ± 0.00	-50.00%
term1	7.30 ± 0.45	15.12%	16.21 ± 0.87	6.27%	3.00 ± 0.24	-76.47%
apex7	6.90 ± 0.33	-6.15%	16.69 ± 1.00	-9.16%	3.90 ± 0.34	-25.81%
alu2	8.30 ± 0.28	-2.47%	22.43 ± 0.52	-3.31%	6.27 ± 0.17	-27.89%
too_large	8.53 ± 0.29	6.23%	21.10 ± 0.54	-5.94%	7.77 ± 0.27	-27.32%
example2	8.33 ± 0.55	0.79%	15.54 ± 0.82	0.88%	8.77 ± 0.79	-34.87%
vda	9.77 ± 0.34	0.34%	27.37 ± 0.44	-7.41%	13.17 ± 0.54	3.19%
alu4	10.23 ± 0.39	0.65%	26.97 ± 0.29	-2.69%	13.07 ± 0.28	-12.64%
Average		0.80		-3.43		-24.83

Table 8.8: Results of using a $2 \times 3 \times 3$ partition with a target FPGA of 2 layers, and the improvement over the $2 \times 2 \times$ partition. Statistics are averaged over 30 runs, with the 95% confidence intervals indicated.

8.4 Template Smoothing for 3D-FPGAs

Template smoothing can be applied to 3D-FPGAs just as it can be applied to 2D-FPGAs. With the different Spiffy-portion configuration it becomes more difficult to characterize the templates, but is no more difficult to perform once the templates are specified.

When using a $1 \times 3 \times 3$ partition, we could easily specify a smoothing-portion by its center Spiffy-portion. When an employing a $2 \times 2 \times 2$ partition, we specify a partition by its corner. Consider the $4 \times 4 \times 4$ grid of level 2 Spiffy-portions, and number them from 0 to 3 along each axis. The smoothing portion consisting of the Spiffy-portions $(0,0,0)$, $(0,1,0)$, $(0,0,1)$, $(0,1,1)$, $(1,0,0)$, $(1,1,0)$, $(1,0,1)$ and $(1,1,1)$ would be denote by the corner of the least value for each coordinate, $(0,0,0)$. In this manner we can unique specify and Spiffy-portion, and use these specifications to unique define each possible smoothing-template.

In Tables 8.11 through 8.14 we see the results from using four progressively larger smoothing-templates for mapping our benchmarks to a four-layer architecture. For tem-

Circuit	Channel-Width		Net Length		Runtime	
	Average	%Δ	Average	%Δ	Average	%Δ
busc	6.93 ± 0.31	5.02%	15.00 ± 0.57	5.67%	6.57 ± 0.60	-49.24%
dma	7.57 ± 0.35	-2.25%	21.44 ± 0.29	4.54%	11.73 ± 0.47	-20.96%
bnre	9.00 ± 0.38	8.16%	23.25 ± 0.26	8.53%	20.73 ± 0.42	-12.07%
dfsm	8.53 ± 0.42	16.34%	20.83 ± 0.20	9.49%	28.80 ± 1.19	-20.50%
9symml	6.80 ± 0.23	1.45%	20.55 ± 0.52	3.50%	3.00 ± 0.00	-50.00%
term1	7.13 ± 0.51	8.55%	14.41 ± 0.62	11.58%	2.47 ± 0.29	-76.19%
apex7	6.77 ± 0.36	8.56%	14.67 ± 0.63	15.46%	4.40 ± 0.45	-76.00%
alu2	8.30 ± 0.37	4.60%	21.23 ± 0.37	1.04%	6.13 ± 0.19	-22.67%
too_large	8.27 ± 0.42	12.06%	19.16 ± 0.49	14.44%	8.13 ± 0.42	-27.08%
example2	7.93 ± 0.42	4.42%	13.48 ± 0.39	12.96%	12.53 ± 1.27	-42.42%
vda	9.10 ± 0.25	11.65%	24.13 ± 0.27	11.75%	20.03 ± 1.81	-41.08%
alu4	9.27 ± 0.39	10.03%	24.21 ± 0.25	10.57%	12.70 ± 0.30	-18.69%
Average		7.38		9.13		-38.08

Table 8.9: Results of using a $2 \times 3 \times 3$ partition with a target FPGA of 3 layers, and the improvement over the $2 \times 2 \times$ partition. Statistics are averaged over 30 runs, with the 95% confidence intervals indicated.

plates T_1, we include the smoothing-portions with corners at $(0,1,1)$, $(2,1,1)$ and $(1,1,1)$ – the center of each plane. For $T2$, we took each smoothing portion from the middle: $(1,1,0)$, $(1,1,2)$, $(1,0,1)$, $(1,2,1)$ and $(1,1,1)$. For T_3 we added to this the same portions at the bottom and top level (changing the z coordinate of T_2 to 0 and 2. For T_4 we used all possible smoothing portions $T_1 \cup T_2 \cup T_3 \cup T_4$. As with the 2D-FPGAs, the quality of the results increase with the size of the template, as does the runtime. Once again, with the largest template no benchmark requires more than 48 seconds and returns a significant improvement in both channel-width and net length.

8.5 Restrictions on Vertical Routing

In the initial proposal of the 3D-FPGA, we believed it likely that any commercial version of such a chip would only provide a minimal number of vertical interconnections. While the research of Lesser, Meleis, Vai, Chiricescu, Xu and Zavracky does not come to the

Circuit	Channel-Width		Net Length		Runtime	
	Average	%Δ	Average	%Δ	Average	%Δ
busc	7.07 ± 0.38	-0.95%	14.44 ± 0.40	8.25%	5.37 ± 0.44	-37.61%
dma	7.57 ± 0.23	4.22%	20.60 ± 0.45	3.38%	11.20 ± 0.46	-12.00%
bnre	9.33 ± 0.32	-1.45%	23.31 ± 0.27	1.79%	18.73 ± 0.31	-7.66%
dfsm	8.53 ± 0.29	-0.39%	20.88 ± 0.20	4.64%	23.07 ± 0.49	-6.30%
9symml	7.67 ± 0.34	-12.75%	19.70 ± 0.47	4.87%	2.93 ± 0.17	-62.96%
term1	7.00 ± 0.31	0.00%	13.76 ± 0.62	0.29%	2.20 ± 0.27	-100.00%
apex7	7.03 ± 0.43	4.96%	15.32 ± 0.73	-5.62%	4.27 ± 0.60	-25.49%
alu2	8.50 ± 0.38	4.49%	21.07 ± 0.67	6.15%	5.40 ± 0.19	-28.57%
too_large	8.30 ± 0.37	-6.41%	18.72 ± 0.35	-1.42%	6.97 ± 0.27	-22.22%
example2	7.87 ± 0.35	5.22%	14.65 ± 1.01	5.20%	9.77 ± 1.60	-23.63%
vda	9.17 ± 0.43	-5.36%	23.49 ± 0.23	-0.07%	12.97 ± 0.66	-7.16%
alu4	9.23 ± 0.35	0.72%	23.40 ± 0.21	4.69%	11.77 ± 0.27	-15.36%
Average		-0.64		2.68		-29.08

Table 8.10: Results of using a $2 \times 3 \times 3$ partition with a target FPGA of 4 layers, and the improvement over the $2 \times 2 \times$ partition. Statistics are averaged over 30 runs, with the 95% confidence intervals indicated.

same conclusion after constructing the actual chip [60, 63], we have still provided Spiffy the capacity to account for this restriction in flexibility.

The modifications to allow Spiffy to handle the case of missing vertical interconnections are straightforward. In order to prevent Spiffy from assigning a net to a non-existent connection, we merely remove the connections end-point switch blocks from consideration during the virtual-node assignment phase. If a net is not assigned to the switch blocks on either end of the connection, it will not use the connection. As Spiffy was already configured to avoid the use of switch blocks that are full, we merely needed to mark each relevant switch block as full (with respect to the vertical direction) in the initialization phase. Having done that, Spiffy will automatically avoid routing nets through the non-existent vertical interconnections.

There is one related issue which is more complex: the assignment of nets to partitions in such a way as to guarantee there is a feasible minimal route between them. Suppose

Circuit	Channel-Width		Net Length		Runtime	
	Average	%Δ	Average	%Δ	Average	%Δ
busc	7.10 ± 0.34	-1.43%	14.78 ± 0.62	4.05%	5.33 ± 0.57	-36.75%
dma	7.33 ± 0.25	7.17%	20.11 ± 0.36	3.06%	11.90 ± 0.67	-19.00%
bnre	8.73 ± 0.38	5.07%	23.17 ± 0.26	0.98%	21.03 ± 0.54	-20.88%
dfsm	8.27 ± 0.38	2.75%	21.45 ± 0.26	2.63%	25.27 ± 0.75	-16.44%
9symml	6.93 ± 0.28	-1.96%	20.88 ± 0.27	-0.31%	2.07 ± 0.09	-14.81%
term1	6.97 ± 0.41	0.48%	14.72 ± 0.66	-9.73%	1.63 ± 0.21	-48.48%
apex7	7.33 ± 0.43	0.90%	15.06 ± 0.73	-8.58%	3.13 ± 0.34	7.84%
alu2	7.97 ± 0.32	10.49%	20.49 ± 0.64	8.49%	5.83 ± 0.28	-38.89%
too_large	7.90 ± 0.45	-1.28%	18.20 ± 0.57	-0.51%	7.77 ± 0.36	-36.26%
example2	8.20 ± 0.44	1.20%	14.69 ± 0.90	4.54%	8.50 ± 1.35	-7.59%
vda	8.93 ± 0.35	-2.68%	23.21 ± 0.45	-0.75%	12.90 ± 0.68	-6.61%
alu4	9.23 ± 0.38	0.72%	23.33 ± 0.42	4.46%	11.53 ± 0.31	-13.07%
Average		1.79%		0.69%		-20.91%

Table 8.11: The results of using smoothing-template T_1 on a 4-layer FPGA, and the improvement over a mapping to a 4-layer FPGA without using template smoothing. All averages are taking over 30 runs, with the 95% confidence interval specified.

the simulated annealing assigned some net to two vertically adjacent partitions, and there were no vertical interconnections between those two partitions. The net would then not be routable, and the algorithm would report failure.

In order to avoid such a situation there were two options: ensuring that the simulated annealing would not allow such a situation arise, or ensuring that there will always be a route between any two partitions. As we will discuss next, the second solution is quite reasonable to implement, and there is no need to complicated our simulated annealing algorithm to implement the first solution.

Given the results of our experiments, it is reasonable to assume that the partition size will be of dimensions $z \times 2 \times 2$, where $z \in \{2, 3\}$. Further, in the benchmarks that we consider the architectures are limited to four levels. As a result, no chip portion past the second recursion will contain more than one level. Consider Figure 8.1, where we see one level of a 3D-FPGA, with the 2×2 partition placed on it to two levels of recursion.

Circuit	Channel-Width		Net Length		Runtime	
	Average	%Δ	Average	%Δ	Average	%Δ
busc	6.93 ± 0.29	0.95%	14.67 ± 0.58	4.76%	5.53 ± 0.46	-41.88%
dma	7.47 ± 0.31	5.49%	20.03 ± 0.49	3.45%	12.23 ± 0.57	-22.33%
bnre	8.83 ± 0.33	3.99%	22.89 ± 0.34	2.21%	23.07 ± 0.80	-32.57%
dfsm	8.33 ± 0.30	1.96%	21.33 ± 0.29	3.15%	27.00 ± 0.56	-24.42%
9symml	7.13 ± 0.32	-4.90%	20.40 ± 0.35	1.96%	1.97 ± 0.07	-9.26%
term1	6.60 ± 0.29	5.71%	14.07 ± 0.69	-4.83%	1.53 ± 0.19	-39.39%
apex7	7.03 ± 0.40	4.96%	15.52 ± 0.77	-11.89%	3.53 ± 0.41	-3.92%
alu2	7.87 ± 0.36	11.61%	20.44 ± 0.46	8.70%	6.53 ± 0.29	-55.56%
too_large	7.97 ± 0.27	-2.14%	18.40 ± 0.54	-1.61%	8.17 ± 0.34	-43.27%
example2	7.80 ± 0.36	6.02%	14.92 ± 0.85	3.07%	9.73 ± 1.70	-23.21%
vda	8.83 ± 0.26	-1.53%	23.90 ± 0.64	-3.71%	14.57 ± 1.06	-20.39%
alu4	9.03 ± 0.35	2.87%	23.14 ± 0.24	5.24%	13.20 ± 0.42	-29.41%
Average		2.92%		0.88%		-28.80%

Table 8.12: The results of using smoothing-template T_2 on a 4-layer FPGA, and the improvement over a mapping to a 4-layer FPGA without using template smoothing. All averages are taking over 30 runs, with the 95% confidence interval specified.

By providing the marked switch blocks with vertical inter-connections, we have assured that each partition has access to some vertical inter-connection, hence a net can be routed between any two partitions. Any further portions will consist of only one level, hence do not need access to vertical inter-connections. Thus only four vertical connections are required per level. In general, for a 3D-FPGA consisting of k levels, we would require $4^{\lceil \frac{k}{2} \rceil}$ vertical interconnections per level.

In Tables 8.15 through 8.17 we see the results of mapping our circuits to these restricted 3D architectures.

8.6 Conclusion

Having created a tool suite that can create circuit mappings to 3D-FPGAs, we have shown the advantages of constructing these devices. The addition of extra levels allows for circuit mappings that require less space and lead to better performance then their two-dimensional

Circuit	Channel-Width		Net Length		Runtime	
	Average	%Δ	Average	%Δ	Average	%Δ
busc	7.07 ± 0.31	-0.95%	14.72 ± 0.64	4.49%	6.90 ± 0.61	-76.92%
dma	7.23 ± 0.29	8.44%	19.39 ± 0.30	6.55%	14.50 ± 0.53	-45.00%
bnre	8.87 ± 0.40	3.62%	21.75 ± 0.27	7.09%	28.10 ± 0.84	-61.49%
dfsm	8.00 ± 0.37	5.88%	20.22 ± 0.19	8.23%	32.17 ± 0.76	-48.23%
9symml	7.10 ± 0.36	-4.41%	19.86 ± 0.32	4.60%	2.80 ± 0.15	-55.56%
term1	6.80 ± 0.37	2.86%	13.61 ± 0.64	-1.41%	1.80 ± 0.27	-63.64%
apex7	7.00 ± 0.33	5.41%	14.78 ± 0.64	-6.53%	4.67 ± 0.46	-37.26%
alu2	8.07 ± 0.32	9.36%	20.59 ± 0.61	8.02%	8.33 ± 0.38	-98.41%
too_large	7.63 ± 0.35	2.14%	17.51 ± 0.18	3.29%	10.87 ± 0.34	-90.64%
example2	8.00 ± 0.40	3.61%	14.09 ± 0.73	8.44%	11.30 ± 1.50	-43.04%
vda	8.83 ± 0.31	-1.53%	22.90 ± 0.57	0.61%	17.47 ± 0.91	-44.35%
alu4	9.30 ± 0.44	0.00%	22.85 ± 0.55	6.43%	15.47 ± 0.47	-51.63%
Average		2.87%		4.15%		-59.68%

Table 8.13: The results of using smoothing-template T_3 on a 4-layer FPGA, and the improvement over a mapping to a 4-layer FPGA without using template smoothing. All averages are taking over 30 runs, with the 95% confidence interval specified.

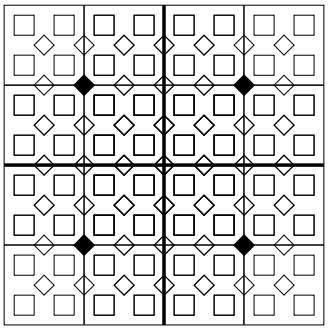


Figure 8.1: One level of a 3D-FPGA, with two recursion-levels of a $2 \times 2 \times 2$ partitioning displayed. If the four black switch blocks are used for vertical inter-connections, then every partition is incident to some vertical inter-connection.

counterparts without requiring additional resources. Even the addition of a single level leads to improvement, and a 3D-FPGA consisting of four levels leads to significant improvement.

In Chapter 3 we discussed the difficulties of constructing 3D-FPGAs, and noted that our work on the subject had motivated researchers at Northeastern University to construct the

Circuit	Channel-Width		Net Length		Runtime	
	Average	%Δ	Average	%Δ	Average	%Δ
busc	6.87 ± 0.29	1.90%	14.11 ± 0.43	8.42%	9.10 ± 0.50	-133.33%
dma	7.57 ± 0.31	4.22%	19.04 ± 0.30	8.21%	18.37 ± 1.14	-83.67%
bnre	8.03 ± 0.39	12.68%	19.75 ± 0.24	15.61%	37.73 ± 1.38	-116.86%
dfsm	7.17 ± 0.28	15.69%	18.39 ± 0.18	16.53%	43.07 ± 1.20	-98.46%
9symml	7.43 ± 0.34	-9.31%	19.55 ± 0.40	6.09%	3.70 ± 0.17	-105.56%
term1	6.43 ± 0.21	8.10%	13.59 ± 0.45	-1.31%	2.70 ± 0.30	-145.45%
apex7	6.83 ± 0.43	7.66%	14.02 ± 0.63	-1.05%	5.93 ± 0.38	-74.51%
alu2	7.53 ± 0.25	15.36%	20.02 ± 0.48	10.57%	11.17 ± 0.30	-165.87%
too_large	7.77 ± 0.36	0.43%	17.45 ± 0.47	3.64%	14.27 ± 0.54	-150.29%
example2	7.90 ± 0.37	4.82%	13.57 ± 0.55	11.84%	14.03 ± 1.27	-77.64%
vda	8.40 ± 0.39	3.45%	22.10 ± 0.58	4.09%	23.77 ± 1.53	-96.42%
alu4	8.57 ± 0.39	7.89%	21.79 ± 0.45	10.78%	20.83 ± 0.58	-104.25%
Average		6.07%		7.79%		-112.69%

Table 8.14: The results of using smoothing-template T_4 (the set of all possible smoothing-portions) on a 4-layer FPGA, and the improvement over a mapping to a 4-layer FPGA without using template smoothing. All averages are taking over 30 runs, with the 95% confidence interval specified.

Name	Channel-Width		Net Length		Runtime
	Average	Best	Average	Best	Average
busc	9.30 ± 1.12	8	15.83 ± 0.95	14.25	5.40 ± 1.44
dma	10.90 ± 1.70	9	22.69 ± 1.06	20.49	9.90 ± 1.24
bnre	15.00 ± 1.26	12	26.16 ± 0.77	23.85	19.40 ± 0.37
dfsm	12.50 ± 0.91	11	23.85 ± 0.47	23.18	25.50 ± 1.27
9symml	10.10 ± 0.86	9	20.62 ± 0.94	18.92	2.20 ± 0.30
term1	12.20 ± 1.25	10	17.53 ± 0.85	15.51	1.30 ± 0.35
apex7	10.20 ± 2.65	7	15.77 ± 1.88	13.78	3.00 ± 0.58
alu2	12.60 ± 0.97	11	22.16 ± 0.38	21.40	4.90 ± 0.23
too_large	13.30 ± 3.07	10	20.91 ± 1.14	19.39	6.60 ± 0.69
example2	12.70 ± 4.03	9	14.94 ± 1.43	13.55	7.80 ± 1.96
vda	22.10 ± 4.52	17	26.82 ± 1.41	25.29	11.70 ± 1.58
alu4	18.60 ± 1.27	16	26.43 ± 0.57	25.30	11.70 ± 0.48

Table 8.15: The results of using restricted vertical routes on a 2-layer 3D-FPGA. All averages are taking over 10 runs, with the 95% confidence interval specified.

Name	Channel-Width		Net Length		Runtime
	Average	Best	Average	Best	Average
busc	18.90 ± 2.51	14	18.30 ± 2.28	14.58	4.80 ± 0.56
9symml	13.70 ± 0.83	12	22.88 ± 0.94	20.56	2.50 ± 0.51
term1	13.50 ± 1.36	11	16.66 ± 1.07	13.26	1.50 ± 0.38
apex7	15.40 ± 1.94	11	19.11 ± 2.14	14.40	2.90 ± 0.71
alu2	25.60 ± 2.06	22	23.56 ± 2.06	21.07	5.60 ± 0.50
too_large	28.90 ± 1.95	24	21.08 ± 1.28	19.38	6.20 ± 0.66
example2	23.50 ± 3.39	17	17.38 ± 1.83	14.46	6.80 ± 3.05

Table 8.16: The results of restricted vertical routes on a 3-layer 3D-FPGA. All averages are taking over 10 runs, with the 95% confidence interval specified.

Name	Channel-Width		Net Length		Runtime
	Average	Best	Average	Best	Average
busc	13.90 ± 2.93	10	17.20 ± 1.81	14.97	4.70 ± 0.48
dma	20.60 ± 1.18	18	22.58 ± 0.38	21.84	10.70 ± 1.22
9symml	13.10 ± 0.92	11	22.23 ± 1.05	19.24	2.00 ± 0.48
term1	10.80 ± 1.61	8	15.71 ± 1.31	13.17	1.40 ± 0.50
apex7	12.90 ± 2.27	10	16.47 ± 1.59	14.17	3.40 ± 0.84
alu2	18.90 ± 2.37	14	24.93 ± 1.44	22.11	5.00 ± 0.34
too_large	19.30 ± 2.24	16	20.90 ± 0.78	19.66	6.40 ± 0.77
example2	23.10 ± 4.76	15	17.72 ± 2.06	14.27	7.10 ± 2.47
alu4	31.60 ± 5.17	24	25.69 ± 1.38	24.51	11.20 ± 0.45

Table 8.17: The results of restricted vertical routes on a 4-layer 3D-FPGA. All averages are taking over 10 runs, with the 95% confidence interval specified.

chips, demonstrating that the architecture is feasible [60, 63]. At this point no manufacturer has produced a commercial 3D-FPGA line, but given these results we strongly believe that it will eventually happen. We have laid the ground work for such a product, and developed the tools necessary to make the product useful.

9

Summary, Conclusion and Future Work

In this chapter we review the accomplishments of this work and discuss the future of the research in this dissertation. In Section 9.1 we outline the completed research, and discuss the results and their significance. In Section 9.2 we will discuss the potential directions our research could take, and the idea we feel are worth pursuing.

9.1 Summary of Results

The accomplishments of this dissertation can be classified into four area: the creation of Spiffy, the implementation of template smoothing, the creation of Gambit, and the work done on three-dimensional field programmable gate arrays.

9.1.1 Spiffy

We first created the tool Spiffy. Spiffy performs simultaneous placement and global routing for FPGAs, and is the first tool to perform either task for 3D-FPGAs. Based on geometric partitioning, Spiffy divides the chip area into a set of disjoint grid-portions, then simultaneously places logic blocks into these portions and routes nets amongst them. Given this course placement and global route, the algorithm is then applied to each grid-portion,

refining the placement and global route within each partition.

Spiffy was implemented in C++ in slightly more than 10,000 lines of code, and provides the user with a number options. The user may specify the use of either Steiner trees or Steiner arboresences to model performance, and may specify a number of the operational parameters. The simulated annealing algorithm used for partitioning may be replaced with an exact branch-and-bound algorithm if the user specifies, or such a substitution can occur when the chip area reaches a sufficiently small size, also specified by the user. In the case of 3D-FPGAs, the user has the flexibility to add a penalty for vertical interconnections, or to disallow the use of large numbers of those interconnections.

In comparing Spiffy against its predecessor, we found Spiffy to be a superior tool. Spiffy was tested over a set of benchmarks frequently used as test-cases for FPGA design automation tools, and compared against the results of Mondrian using the same benchmarks and resources. On average, Spiffy reduces the runtime by 38%, with the runtime over the largest benchmark being reduced from 3.4 hours to 14 minutes. Further, the circuit mappings produced by the Spiffy/Upstart pair are of a substantially better quality, lowering the channel-width by an average of 13% and the net size by an average of 10%. In comparing it to tools produced since Mondrian we see that the Spiffy/Upstart tool suite produces some of the best results in the literature. Only VPR achieves better results, and those are derived from superior technology mappings, giving the tool an additional advantage.

9.1.2 Template Smoothing

We presented the technique of template smoothing. Template smoothing augments the geometric partitioning methodology and leads to superior circuit mappings. When used with the Spiffy algorithm, template smoothing intersperses invocations of the algorithm into the recursive breath-first search. Where Spiffy divides the chip area into partitions and deals with each partitions independently, template smoothing also considers partition subproblems that overlap the original subproblems. As a result, Spiffy has the opportunity

to correct and improve solutions that suffered from the loss of information incurred from dealing with each original partition independently.

Template smoothing is an option to the Spiffy tool, with the user free to specify the number and exact locations of extra partitions. The number of extra partitions will increase the overall runtime, with experimental results showing the correlation between this number and the percentage increase in runtime is essentially linear. With the largest number of partitions in our experiments, we achieved an 6% improvement in channel-width and 7% improvement in path-length when compared to mappings produces without using template smoothing. Using smaller numbers of partitions, we achieved smaller, but still significant, increases in quality with a lesser increase in runtime.

Note that while Spiffy and the template smoothing augmentation were designed for 3D-FPGAs, the techniques are by no means limited to that chip architecture. These algorithms are equally applicable to many semi-custom design styles, and with some variations could be applied to full-custom chips. For each style a new tool would have to be coded, but the theoretical challenges in doing so would be minimal.

9.1.3 Gambit

We created Gambit. Gambit is the first tool to perform placement, global routing and detailed routing simultaneously. Gambit is designed for FPGAs (2D and 3D), though the concept could be generalized to many channel-based architectures. Based on the constraint imposed by many commercial FPGAs, Gambit maintains a structure known as a conflict graph, a graph whose coloring will induce a feasible detailed route. By configuring Spiffy to make its decision in such a way as to heuristically minimize the chromatic number of the conflict graph, the final algorithm produces a detailed routing in addition to the placement and global routing already created by Spiffy.

Gambit is significant in that it is a proof of concept, showing that it is possible to perform all three phases together. We do not claim the results to be competitive with other

tools in the field. We do claim the tool has the potential to produce quality mapping. We have argued the merits of integrating phases, and have shown that a small increase in the quality of the coloring algorithm brings about a large increase in quality. By using more sophisticated coloring techniques, we believe the quality of the results will become competitive to those of other tools. In Section 9.2 we will discuss possible improvements to the algorithm, which we believe will lead to a tool competitive with other tool suits in the literature.

9.1.4 3D-FPGAs

We proposed and justified the idea of a 3D-FPGA. By placing FPGAs in layers to form a three-dimensional configuration, we argued that the same amount of resources could be used to construct circuit mappings that would require less space and have a superior performance. Initially we made a theoretical justification for this argument, which motivated researchers at Northeastern to implement such an architecture [60, 63].

Having developed the first tools for the new architecture, we were able to create the first circuit mappings, and thus the first experimental results justifying their use. In performing experiments, we found that when mapping circuits to 2D-FPGAs with Spiffy, and comparing the results to four-layered 3D-FPGAs of equivalent resources, the 3D-FPGA reduced channel-width by an average of 22%, and reduced net length by an average of 11%. We continued to map the benchmarks to FPGAs by varying the number of levels, and established that in practice the addition of each new level resulted in improvement.

9.2 Future Work

While within each category we achieved the goals listed, there are a number of research paths we believe are still worth pursuing. To start, a number of aspects of the Spiffy algorithm leave room for improvement:

- Steiner Tree Storage: Spiffy relies on the quick computation of minimum rectilinear Steiner trees, and accomplishes this calculation by storing all possible such trees over the partition graph. As a result, we are limited by space to fairly small partition sizes. However, as nets are generally small, it seems likely that Steiner trees with a large number of terminals could be eliminated from the list, thus allowing us to use larger partition graphs without the memory penalty incurred by those larger Steiner tree lists.

- Sharp Partitioning: In performing the sharp partitioning, blocks are placed within partitions so as to minimize the "spread" of the net, leaving the route-selection phase to attempt to minimize congestion between nets given the placement. Thus Spiffy is implicitly prioritizing path length minimization over channel-width minimization, though the later is generally regarded as more important. We believe it is worth an effort to incorporate a more direct channel-width minimization technique into the partition phase. It is conceivable that during the simulated annealing phase a penalty could be incorporated into a solution score to reflect that a solution would lead to a larger channel-width, but the ability to calculate such a penalty quickly would be vital to the performance of the simulated annealing.

- Floating Virtual Terminals: Once a virtual terminal has been assigned to a switch block, it is fixed to that block (save by removal during template smoothing). However, it could prove advantageous during a later partitioning phase to move that virtual terminal to another switch block along the relevant chip portion's boarder. Doing such would then effect the placement within other independent portions, creating some difficult challenges with respect to the data structures and recursion. It may be worth a programmer's time to incorporate the ability to adjust virtual terminals during the partition phase, if this can be done without an undo cost in runtime.

- Miniature Routing: Currently, Spiffy uses as its base case a chip portion containing

exactly one logic block. An examination should be conducted that considers expanding the base to route larger portions, thus providing the tool with more information as to the net's path when making the final connections. However, the integer program currently used cannot be easily generalized to these larger base cases, and it is likely a new method would need to be developed.

While Spiffy could benefit from research along these lines, Gambit clearly requires significant work in order to become a quality tool. The first step in doing such will require a deeper investigation into graph-coloring heuristics. Two specific techniques worth exploring is the sequential graph color algorithm developed by Sarma and Banyopadhyay [83], and the semidefinite method of Karger, Motwani and Sudan [52]. Further, a more sophisticated method of choosing net colors and routes is required then the simple greedy algorithm currently employed.

However, the most important area of research involving Gambit is that of the miniature routing. While we have adopted the Spiffy integer programming solution method to handle the conflict graph, the solution is unsatisfactory. Because of the complexity of the graph-coloring problem and the size of the conflict graph, it is infeasible to have the integer program minimize the coloring over the entire graph for each base case. Only a small portion of the graph – the portion relevant to the base-case in question – is considered. It is our belief that an approach should be developed that would heuristically route the block and color the entire graph, as opposed to exactly coloring a smaller portion without consideration of the remainder of the graph.

Bibliography

[1] Aggarwal, A. and Lewis, D. Routing Architectures for Hierarchical Field-Programmable Gate Arrays. *IEEE Proceedings of the Custom Integrated Circuits Conference*, pages 475–478, 1994.

[2] Aho, A.V., Sethi, R., and Jeffery, U.D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[3] Michael Alexander. *High-Performance Routing for FPGAs.* PhD thesis, University of Virginia, Department of Computer Science, Charlottesville, VA, 1996.

[4] Alexander, M.J., Cohoon, J.P., Colflesh, J.L., Karro, J., Peters, E.L., and Robins, G. Physical Layout for Three-Dimensional FPGAs. *Fifth ACM/SIGDA Physical Design Workshop*, pages 142–149, April 1996.

[5] Alexander, M.J., Cohoon, J.P., Colflesh, J.L., Karro, J., Peters, E.L., and Robins, G. Placement and Routing for Three-Dimensional FPGAs. *Fourth Canadian Workshop on Field Programmable Devices*, pages 11–18, 1996.

[6] Alexander, M.J., Cohoon, J.P., Colflesh, J.L., Karro, J., and Robins, G. Three-Dimensional Field-Programmable Gate Arrays. *IEEE ASIC Conference*, pages 253–256, September 1995.

[7] Alexander, M.J., Cohoon, J.P., Ganley, J.L., and Robins, G. Performance-oriented

placement and routing for field-programmable gate arrays. *Proceedings of the European Design Automation Conference*, pages 80–85, 1995.

[8] Alexander, M.J. and Robins, G. A New Approach to FPGA Routing Basses on Multi-Weight Graphs. *Proceedings of the ACM/SIGDA International Workshop on Field-Programmable Gate Arrays*, February 1994.

[9] Alexander, M.J. and Robins, G. New Performance-Driven FPGA Routing Algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1505–1517, December 1996.

[10] L. Ammeraal. *STL for C++ Programmers.* John Wiley, 1996.

[11] A.W. Appel. *Modern compiler implementation in ML.* Cambridge University Press, 1997.

[12] Bapat, S. and Cohoon, J.P. A parallel VLSI circuit layout methodology. *Proceedings of Sixth International Conference on VLSI Design*, pages 236–241, 1993.

[13] M.R.C.M. Berkelaar. LP_SOLVE user's manual (version 1.5), 1994.

[14] Betz, V. and Rose, J. Directional Bias and Non-Uniformity in FPGA Global Routing Architectures. *Proceedings of the International Conference on Computer Aided Design*, pages 642–659, 1996.

[15] Betz, V. and Rose, J. VPR: A New Packing, Placement and Routing Tool for FPGA Research. *International Workshop on Field Programmable Logic and Applications*, pages 213–222, 1997.

[16] Borriello, G., Ebeling, C., Hauck, S., and Burns, S. The Triptych FPGA Architecture. *IEEE Transactions on VLSI Systems*, 3(4):473–482, 1995.

[17] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, pages 251–256, 1979.

[18] M. Breuer. A class of min-cut placement algorithms. *Proceedings of the 14th ACM/IEEE Design Automation Conference*, pages 284–290, 1977.

[19] M. Breuer. Min-cut placement. *Journal of Design Automation and Fault-Tolerant Computing*, 1:343–362, 1977.

[20] Brown, S.D. FPGA Architectural Research: A Survey. *IEEE Design and Test of Computers*, 13(4):9–15, Winter 1996.

[21] Brown, S.D., Francis, R.J., Rose, J., and Vranesic, Z.G. *Field-Programmable Gate Arrays*. Kluwer Academic Publications, Boston, MA, 1992.

[22] Brown, S.D. and Rose, J. FPGA and CPLD Architectures: A Tutorial. *IEEE Design and Test of Computers*, 13(2):42–57, Summer 1996.

[23] Brown, S.D., Rose, J., and Vranesic, Z.G. A Detailed Router for Field-Programmable Gate Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(5):620–628, May 1992.

[24] Brown, S.D., Rose, J.S., and Vranesic, Z.G. A detailed router for field-programmable gate arrays. *Proceedings of the International Conference on Computer-Aided Design*, pages 382–385, 1990.

[25] Cahill, C., Compagno, A., O'Donovan, J., Slattery, O., Mathuna, S.C., Barrett, J., Sethelon, I., Val, C., Tigneres, J.P., Stern, J., Ivey, P., Masgrangeas, M., and Coello-Vera, A. Thermal Characterization of Vertical Multichip Modules MCM-V. *IEEE Transactions on Components, Packaging, and Manufacturing Technology*, 18(4):765–771, December 1995.

[26] Carter, W.S., Duong, K., Freeman, R.H., Hsier, H.C., Ja, J.Y., Mahoney, J.E., Ngo, L.T., and Sze, S.L. A User Programmable Reconfiguration Gate Array. *Proceedings of the Custom Integrated Circuits Conference*, pages 233–235, May 1986.

[27] Chang, Y.W., Thakur, S., Zhu, K., and Wong, D.F. A new global routing algorithm for FPGAs. *Proceedings of the International Conference on Computer Aided Design*, pages 380–385, 1994.

[28] Chang, Y.W. and Wong, D.F. Universal Switch Modules for FPGA Design. *ACM Transactions on Design Automation of Electronic Systems*, 1(1):80–101, January 1996.

[29] Chang, Y.W., Wong, D.F., and Wong, C.K. FPGA Global Routing Based on a New Congestion Metric. *Proceedings of the International Conference on Computer Aided Design*, 1995.

[30] Chang, Y.W., Wong, D.F., and Wong, C.K. Universal Switch-Module Design for Symmetric-Array-Based FPGAs. *FPGA*, pages 80–86, 1996.

[31] Chen, C., Tsay, T., Hwang, A., Wu, H., and Lin, Y. Combining technology mapping and placement for delay-optimization in FPGA designs. *Proceedings of the International Conference on Computer Aided Design*, pages 123–127, 1993.

[32] Chung, K. Using Hierarchical Logic Blocks to Improve the Speed of Field-Programmable Gate Arrays. *Proceedings of the International Workshop on Field Programmable Logic and Applications*, 1991.

[33] Chung, K. and Rose, J. TEMPT: Technology Mapping for the Exploration of FPGA Architectures with Hard-Wired Connections. *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 361–367, 1992.

[34] Cong, J. and Ding, Y. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Transactions on Computer-Aided Design*, 13(1):1–12, February 1993.

[35] Cong, J., Hwang, Y., and Xu, S. Technology Mapping for FPGAs with Nonuniform

Pin Delays and Fast Interconnections. *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 273–278, 1999.

[36] Cong, J. and Xu, S. Delay-Optimal Technology Mapping for FPGAs with Heterogeneous LUTs. *Proceedings of the 35th ACM/IEEE Design Automation Conference*, pages 704–707, 1998.

[37] Coudert, O. Exact Coloring of Real-Life Graphs is Easy. *Proceedings of the 34th IEEE/ACM Design Automation Conference*, pages 121–125, 1997.

[38] Dunlop, A. and Kernighan, B. A procedure for placement of standard VLSI circuits. *IEEE Transactions on Computer-Aided Design*, 4:92–98, 1985.

[39] Feuer, M. VLSI Design Automation: An Introduction. *Proceedings of the IEEE*, 71(1):1–9, January 1983.

[40] Francis, R., Rose, J., and Chung, K. Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays. *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 613–619, 1990.

[41] Frank, A., Nishizeki, T., Saito, H., and Tardos, E. Algorithms for routing around a rectangle. *Discrete Applied Mathematics*, 40:363–378, 1992.

[42] Frankle, J. Iterative and Adaptive Slack Allocation for Performance-Driven Layout and FPGA Routing. *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 536–542, 1992.

[43] J. Ganley. *Geometric Interconnections and Placement Algorithms.* PhD thesis, University of Virginia, Department of Computer Science, Charlottesville, VA, 1995.

[44] Ganley, J. and Cohoon, J. Improved Computation of Optimal Rectilinear Steiner Tree Minimal Trees. *International Journal of Computational Geometry and Applications*, 7:457–472, 1997.

[45] Garey, M.R. and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[46] T. Günther. Die räumliche Anordnung von Einheiten mit Wechselbeziehungen. *Elektronische Datenverarbeitung*, 11:209–211, 1969.

[47] Hauck, S., Borriello, G., and Ebeling, C. TRIPTYCH: An FPGA for Synchronous and Asynchronous Circuits. *Brown/MIT Conference on Field-Programmable Gate Arrays*, February 1992.

[48] Hopcraft, J.E. and Karp, R.M. An $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs. *Siam Journal on Computing*, 2:225–231, 1973.

[49] Hsieh, H. Third-Generation Architecture Boots Speed and Density of Field-Programmable Gate Arrays. *IEEE Proceedings of the Custom Integrated Circuits*, pages 31.2.1–31.2.7, 1990.

[50] Hwang, F.K., Richards, D.S., and Winter, P. *The Steiner Tree Problem.* Elsevier Science Publications, 1992.

[51] Ignizio, J.P. and Cavalier, T.M. *Liner Programming.* Prentice Hall, Inc., 1994.

[52] Karger, D., Motwani, R., and Sudan, M. Approximate Graph Coloring by Semidefinite Programming. *Journal of the ACM*, 45(2):246–265, March 1998.

[53] Karro, J. and Cohoon, J. A Spiffy Tool for the Simultaneous Placement and Global Routing of Three-Dimensional Field Programmable Gate Arrays. *Ninth Great Lakes Symposium on VLSI*, pages 226–227, March 1999.

[54] Karro, J. and Cohoon, J. An Approach to the Physical Design Problems of 3D-FPGAs. *International Symposium on Circuits and Systems*, July 1999.

[55] Kirkpatric, S., Gelatt Jr., C.D., and Vecchi, M.P. Optimization by Simulated Annealing. *Science*, 220:671–679, 1983.

[56] Johannes, F. Kleinhans, J., Sigl, F. and Anbtreich, K. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE Transactions on Computer-Aided Design*, 10:356–365, 1991.

[57] Kramer, M.R. and van Leeuwen, J. *Wire-routing is NP-complete.* Technical Report RUU-CS-82-4, University of Utrecht, Netherlands, 1992.

[58] U. Lauther. A min-cut placement algorithm for general cell assemblies based on a graph representation. *Journal of Digital Systems*, 4:21–34, 1979.

[59] Lee, Y. and Wu, A. A performance and routability driven router for FPGAs considering path delays. *Proceedings of the 32 IEEE/ACM Conference on Design Automation*, pages 557–561, 1995.

[60] Leeser, M., Meleis, W.M., Vai, M.M., Chiricescu, S., Xu, W., and Zavracky, P.M. Rothko: A Three-Dimensional FPGA. *IEEE Design and Test of Computers*, 15(1):16–23, January - March 1998.

[61] Lemieux, G. and Brown, S. A detailed routing algorithm for allocating wire segments in field-programmable gate arrays. *Proceedings of the Fourth Physical Design Workshop*, pages 215–226, 1993.

[62] Lemiux, G., Brown, S., and Vranesic, D. On Two-Step Routing for FPGAs. *International Symposium on Physical Design*, pages 60–66, 1997.

[63] Lesser, M., Meleis, W.M., Vai, M.M., and Zavracky, P.M. Rothko: A Three Dimensional FPGA Architecture, Its Fabrication, and Design Tools. *Field-Programmable Logic and Applications*, 1997.

[64] J. Lienig. A Parallel Genetic Algorithm for Performance-Driven VLSI Routing. *IEEE Transactions on Evolutionary Computation*, 1(1):29–39, 1997.

[65] Mak, W-K and Wong, D.F. Performance-driven board-level routing for FPGA-based logic emulation. *Proceedings of the International Conference on Computer Design, VLSI in Computers and Processors*, pages 199–201, 1998.

[66] Marquardt, V., Betz, V., and Rose, J. Timing-Driven Placement for FPGAs. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 175–184, February 2000.

[67] Mayrhofer, S. and Lauther, U. Congestion-driven placement using a new multi-partitioning heuristic. *Proceedings of the International Conference on Computer Aided Design*, pages 332–335, 1990.

[68] Meleis, W., Leeser, M., Zavracky, P., and Vai, M. Architectural Design of a Three Dimensional FPGA. *IEEE Seventeenth Conference on Advanced Research in VLSI*, pages 256–268, 1997.

[69] Nag, S.K. and Rutenbar, R.A. Performance-driven simultaneous place and route for row-based FPGAs. *Proceedings of the ACM/IEEE 31st Design Automation Conference*, pages 301–307, 1994.

[70] Nag, S.K. and Rutenbar, R.A. Performance-driven simultaneous place and route for island-style FPGAs. *IEEE/ACM International Conference on Computer Aided Design*, pages 332–338, 1995.

[71] Nag, S.K. and Rutenbar, R.A. Performance-driven simultaneous place and route for FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(5):499–518, June 1998.

[72] Nakatake, S., Sakanushi, K., Kajitani, Y., and Kawakita, M. The channeled-BSG: a universal floorplan for simultaneous place/route with IC applications. *IEEE/ACM International Conference on Computer-Aided Design*, pages 418–25, 1998.

[73] J.M. Rabaey. *Digital Integrated Circuits.* Prentice-Hall, Inc., Upper Saddle River, New Jersey, 1996.

[74] Radunovic, B. and Milutinovic, V. A survey of reconfigurable computing architectures. *Field-Programmable Logic and Applications, 8th International Workshop*, pages 376–385, 1998.

[75] Raghavan, R., Cohoon, J.P., and Sahni, S. Single bend wiring. *Journal of Algorithms*, 7:232–257, 1986.

[76] C. Rim. A performance driven FPGA routing method. *Journal of the Korea Information Science Society*, 25(6):626–35, June 1998.

[77] Rose, J. Parallel Global Routing for Standard Cells. *IEEE Transactions on Computer Aided Design*, pages 1085–1095, October 1990.

[78] Rose, J. and Brown, S. Flexibility of Interconnection Structures for Field-Programmable Gate Arrays. *Journal of Solid-State Circuits*, 26(3):277–281, March 1991.

[79] Rose, J., Francis, R.J, Lewis, D., and Chow, P. Architecture of Field-Programmable Gate Arrays: The Effects of Logic Block Functionality on Area Efficiency. *IEEE Journal of Solid-State Circuits*, 27(5):1217–1225, October 1990.

[80] Rose, J., Snelgrove, W., and Vranesic, Z. ALTOR: An automatic standard cell layout program. *Canadian Conference on VLSI*, pages 169–173, 1985.

[81] Roy, S., Belkhale, K., and Banerjee, P. An α-approximate algorithm for delay-constraint technology mapping. *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 367–372, 1999.

[82] M.J. Saltzman. *Code for Minimum-Cost Perfect Matching.* Clemson University, Clemson SC, 1992.

[83] Sarma, S.S. and Bandyopadhyay, S.K. Some sequential graph coloring algorithms. *Electronics*, 67(2):187–199, 1989.

[84] Sarrafzadeh, M. and Wong, C.K. *An Introduction to VLSI Physical Design*. McGraw-Hill Companies, Inc., New York, NY, 1996.

[85] Schlag, M., Kong, J., and Chan, P. Routability-Driven Technology Mapping for Lookup Table-Based FPGA's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):13–26, January 1994.

[86] Sechen, C. and Sangiovanni-Vincentelli, A. The Timberwolf Placement and Routing Package. *IEEE Journal of Solid-State Circuits*, 20(2), April 1985.

[87] Seed, L., Meacham, R., Zawads, A., and Thorne, P. *TriMorph: 3D Computational Structures*. http://www.shef.ac .uk/uni/academic/D-H/eee/esg/research/trimorph.html.

[88] Sherwani, N. *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publications, Norwell, MA, 1999.

[89] Shi, W. and Su, Chen. *The Rectilinear Steiner Arborescence is NP-Complete*. University of Texas Technical Report #N-99-001, July 1999.

[90] Singh, S. The Effect of Logic Block Architecture on FPGA Performance. *IEEE Journal of Solid-State Circuits*, 27(3):281–287, March 1992.

[91] Suaris, P. and Kedem, G. A Quadrisection Based on Combines Place and Route Scheme for Standard Cells. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(3):234–244, March 1989.

[92] Suaris, P. and Kedem, G. A quadrisection-based place and route scheme for standard cells. *IEEE Transactions on Computer Aided Design*, 8:234–244, 1989.

[93] Thakuar, S., Chang, Y., Wong, D., and Muthukrishnan, S. Algorithms for an FPGA Switch Module Routing Problem with Applications to Global Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(1):43–46, January 1997.

[94] Thakur, S., Wong, D.F., and Muthukrishnan, S. Algorithms for FPGA switch module routing. *Proceedings of the European Design Automation Conference*, pages 265–270, 1994.

[95] Togawa, N., Sato, M., and Ohtsuki, T. A Simultaneous Placement and Global Routing Algorithm for FPGAs. *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 483–483, 1994.

[96] Togawa, N., Sato, M., and Ohtsuki, T. Maple-opt: a simultaneous technology mapping, placement, and global routing algorithm for FPGAs with performance optimization. *Proceedings of the IEEE/ACM Asian and South Pacific Design Automation Conference*, pages 319–327, 1995.

[97] Togawa, N., Sato, M., and Ohtsuki, T. A simultaneous technology mapping, placement, and global routing algorithm for FPGAs with path delay constraints. *EICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, E79-A(3), March 1996.

[98] Togawa, N., Yanagisawa M, and Ohtsuki, T. Maple-opt: A performance-oriented simultaneous technology mapping, placement and global routing algorithm for FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(9):803–18, September 1998.

[99] S. Trimberger. *Field-Programmable Gate Array Technology.* Kluwer Academic Press, 1994.

[100] S. Trimberger. Effects of FPGA Architecture on FPGA Routing. *Proceedings of the ACM/IEEE Design Automation Conference*, pages 574–578, 1995.

[101] Wu, Y. and Marek-Sadowaka, M. An efficient router for 2-D field programmable gate arrays. *Proceedings of the European Design and Test Conference*, pages 412–416, 1994.

[102] Wu, Y.L. and Chang, D. On the NP-completeness of Regular 2-D FPGA Routing Architectures and a Novel Solution. *Proceedings of the European Design Automation Conference*, pages 271–275, 1994.

[103] Wu, Y.L. and Marek-Sadowska, M. Graph Based Analysis of FPGA Routing. *Proceedings of the European Design Automation Conference*, pages 104–109, 1993.

[104] Wu, Y.L. and Marek-Sadowska, M. Orthogonal Greedy Coupling – A New Optimization Approach to 2-D FPGA Routing. *Proceedings of the ACM/IEEE Design Automation Conference*, pages 568–573, 1995.

[105] *The Programmable Logic Data Book*. 1994.

[106] *Xilinx Home Page*. http://www.xilinx.com/.

[107] Zavracky, P., Zavracky, M., VU, D.,, and Dingle, B. *Three Dimensional Processor using Transferred Thin Film Circuits*, January 1997. US Patent Application #08-531-177.